
Diofant Documentation

Release 0.8.0

Diofant Development Team

November 07, 2016

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	8
2.4	Printing	9
2.5	Simplification	11
2.6	Calculus	18
2.7	Solvers	23
2.8	Matrices	25
2.9	Expression Trees	32
3	Modules Reference	37
3.1	Diofant Core	37
3.2	Combinatorics	158
3.3	Number Theory	250
3.4	Concrete Mathematics	279
3.5	Numerical evaluation	296
3.6	Numeric Computation	303
3.7	Mathematical Functions	305
3.8	Geometry	429
3.9	Symbolic Integrals	525
3.10	Numeric Integrals	559
3.11	Logic	566
3.12	Matrices	577
3.13	Polynomials Manipulation	660
3.14	Printing System	864
3.15	Interactive	885
3.16	Plotting	886
3.17	Term rewriting	900
3.18	Series Expansions	902
3.19	Sets	908
3.20	Simplify	922
3.21	Details on the Hypergeometric Function Expansion	946
3.22	Stats	956
3.23	ODE	994
3.24	PDE	1045

3.25 Solvers	1054
3.26 Diophantine	1072
3.27 Inequality Solvers	1093
3.28 Tensors	1096
3.29 Utilities	1121
3.30 Parsing input	1167
3.31 Calculus	1170
3.32 Differential Geometry	1177
3.33 Vector Module	1192
4 Developer's Guide	1229
4.1 Guidelines for Contributing	1229
4.2 Rosetta Stone	1230
4.3 Versioning and Release Procedure	1230
4.4 How to Build Documentation	1231
5 About	1233
5.1 License	1233
5.2 SymPy Development Team	1234
5.3 Brief History	1245
6 Release Notes	1247
6.1 Diofant 0.8	1247
Bibliography	1257

INSTALLATION

The Diofant can be installed on virtually any computer with Python 3.4 or above. Diofant requires [setuptools](#). You can install latest release with pip:

```
$ pip install diofant
```

or to install also extra dependencies:

```
$ pip install diofant[gmpy,plot]
```

Tip: You could use [pyenv](#) to create isolated Python environment first, instead of installing everything system-wide.

1.1 From Sources

If you are a developer or like to get the latest updates as they come, be sure to install from git:

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

1.2 Run Diofant

To verify that your freshly-installed Diofant works, please start up the Python interpreter and execute some simple statements like the ones below:

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

However, we recommend using [IPython](#) for working interactively. Use input transformations from the module [diofant.interactive](#) (page 885), that could reduce boilerplate while interacting with Diofant due to the Python language syntax.

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

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 1229).

TUTORIAL

Warning: It is assumed that the reader already knows the Python programming language. If you do not, please start from the [official 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. 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, we must use `symbols()` (page 83), that takes a string of symbol names separated by spaces or commas, and creates *Symbol* (page 81) instances out of them.

Note that we wrote $x + 2*y$, 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 $x + 2*y$. Now let us play around with it:

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

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

Tip: Use `evaluate()` (page 58) 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
```



```
>>> 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 plotting, printing (like 2D pretty printed output of math formulas, or \LaTeX), code generation, statistics, 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 *
>>> x, y, z = symbols('x y z')
>>> init_printing(pretty_print=True, use_unicode=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)
x      x
e ·sin(x) + e ·cos(x)
```

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

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

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

```
>>> integrate(sin(x**2), (x, -oo, oo))
√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)
[
[-√2, √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)), f(x))
f(x) = ex · C2 + e-x · C1
```

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

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

Rewrite the Bessel function $J_n(z)$ in terms of the spherical Bessel function $j_n(z)$.

```
>>> n = symbols('n')
>>> besselj(n, z).rewrite(jn)
sqrt(2) * sqrt(z) * jn(n - 1/2, z)
-----
sqrt(pi)
```

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

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

2.2 Basics

Here we discuss some of the most basic operations needed for expression manipulation in Diofant.

2.2.1 Substitution

One of the most common things you might want to do with a mathematical expression is substitution with `subs()` (page 54) 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.

To perform several substitutions in one shot, you can provide `Iterable` sequence of pairs.

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

$$\begin{pmatrix} x \\ \begin{pmatrix} x \\ x \end{pmatrix} \\ x \end{pmatrix}$$

```

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
```

2.2.2 Numerics

To evaluate a numerical expression into a floating point number with arbitrary precision, use `evalf()` (page 147). 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
```

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()
-0.e-124
>>> (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.

The easiest way to convert a Diofant expression to an expression that can be numerically evaluated with libraries like `numpy` — use the `lambdify()` (page 1163) function. It acts like a `lambda` form, except it converts the Diofant names to the names of the given numerical library.

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

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

```
>>> f = lambdify(x, expr, "math")
>>> f(0.1)
0.09983341664682815
```

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 91) instead of `int`. In most cases, Python numeric types will be correctly coerced to Diofant numbers during expression construction.

```
>>> 3 + x**2
2
x + 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)
1.5
x
```

Tip: While working in the IPython console, you could use `IntegerWrapper` (page 886) AST transformer to wrap all integer literals with `Integer` (page 91) automatically.

The universal solution is using correct Diofant numeric class to construct numbers explicitly. For example, `Rational` (page 89) in the above example

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

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 110), which can be used to create a symbolic equation

```
>>> Eq((x + 1)**2 - x**2, 2*x + 1)
      2      2
- x  + (x + 1) = 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 317), `acos()` (page 318) 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 10)
- `Repr` (page 10)
- `2D Pretty Printer` (page 10) (Unicode or ASCII)
- `LaTeX` (page 11)
- `Dot` (page 11)

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 885) function.

Lets take this simple expression

```
>>> expr = Integral(sqrt(1/x), x)
```

and try several available printers.

2.4.1 Str

To get a string form of an expression, use `str`. This is also the form that is produced by `print()`. String forms are designed to be easy to read and mostly to be in a form that is a correct Python syntax so that it can be copied and pasted.

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

2.4.2 Repr

The `repr` form of an expression is designed to show the exact form of an expression, it would yield an object with the same value when passed to `eval()`. To get it, use `repr()`.

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

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

2.4.3 2D Pretty Printer

A two-dimensional (2D) textual representation of the expression can be obtained with `pretty()` (page 867).

```
>>> pretty(expr)
┌ ──── \n|    / 1    \n|    / - dx\n|  /  x    \n|    '
>>> print(_)
┌ ────
|  /  1
|  - dx
|  /  x
```

You can pass `use_unicode=False` to use ASCII symbols.

```
>>> print(pretty(expr, use_unicode=False))
```

```
┌
|  /
|  /  1
|  - dx
|  /  x
|
└
```

`pprint()` (page 867) prints the output to the screen.

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

```
┌
|  / 1
```

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

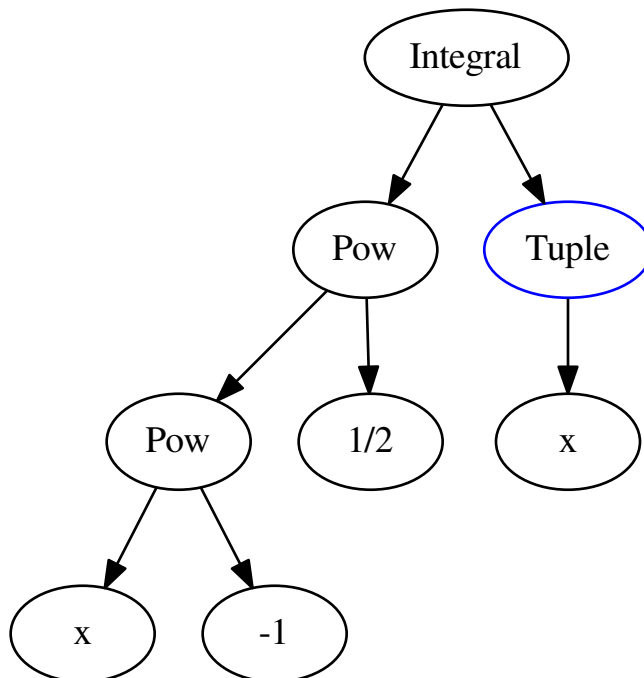
2.4.4 LaTeX

To get the \LaTeX form of an expression, use `latex()` (page 875).

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

2.4.5 Dot

`dotprint()` (page 884) 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 922) 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 685) function, given a polynomial with rational coefficients in several variables, is guaranteed to produce a factorization into irreducible factors.

The `simplify()` (page 922) 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 922) lets the user specify the Python function used to determine how “simple” an expression is. The default is `count_ops()` (page 142), 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 922) 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)
2
x  + 2·x + 1
>>> factor(_)
      2
(x + 1)
```

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

The function `simplify()` (page 922) 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 922) 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 137) 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)
2
x  + 2·x + 1
>>> expand((x + 2)*(x - 3))
2
x  - x - 6
```

Given a polynomial, `expand()` (page 137) will put it into a canonical form of a sum of monomials with help of more directed expansion methods, namely `expand_multinomial()` (page 145) and `expand_mul()` (page 143).

`expand()` (page 137) 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 137) on it due to cancellation.

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

Function `factor()` (page 685) takes a multivariate polynomial with rational coefficients and factors it into irreducible factors.

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

For polynomials, `factor()` (page 685) is the opposite of `expand()` (page 137).

Note: The input to `factor()` (page 685) and `expand()` (page 137) 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)
      2      2
sin(x) + 2·sin(x)·cos(x) + cos(x)
>>> factor(_)
      2
(sin(x) + cos(x))
```

`collect()` (page 930) collects common powers of a term in an expression.

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

`collect()` (page 930) is particularly useful in conjunction with the `coeff()` (page 68) method.

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

`cancel()` (page 688) 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      1
----- + -
 x - 4   x
>>> cancel(_)
      2
```

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

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

Note: Since `factor()` (page 685) will completely factorize both the numerator and the denominator of an expression, it can also be used to do the same thing:

```
>>> factor(expr)
      2
(y - z)
-----
      x - 1
```

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

`apart()` (page 737) 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)
      3      2
4*x + 21*x + 10*x + 12
-----
      4      3      2
x + 5*x + 5*x + 4*x
>>> apart(_)
      2      1      3
----- + ----- + -----
      2      x + 4      x
x + x + 1
```

2.5.2 Trigonometric Functions

To simplify expressions using trigonometric identities, use `trigsimp()` (page 936) function.

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

```
>>> trigsimp(sin(x)*tan(x)/sec(x))
      2
sin (x)
```

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 922) function, *trigsimp()* (page 936) 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 144) function.

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

2.5.3 Assumptions

By default, Diofant performs all calculations assuming that symbols are commutative with respect to the multiplication. 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 81) 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 50).

```
>>> 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
```

2.5.4 Powers and Logarithms

powdenest() (page 938) function applies identity $(x^a)^b = x^{ab}$, from left to right, if assumptions allow.

```
>>> a, b = symbols('a b', extended_real=True)
>>> powdenest((p**a)**b)
a·b
p
```

powsimp() (page 937) function reduces expression by combining powers with similar bases and exponent.

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

Again, as for *powdenest()* (page 938) 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)
a
(p·q)
```

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

expand_power_exp() (page 145) and *expand_power_base()* (page 145) functions do reverse of *powsimp()* (page 937).

```
>>> expand_power_exp(x**(y + z))
y z
x ·x
>>> expand_power_base((p*q)**a)
a a
p ·q
```

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 143). 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 143), use `logcombine()` (page 928) function.

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

2.5.5 Special Functions

Diofant implements dozens of *special functions* (page 307), ranging from functions in combinatorics to mathematical physics.

To expand special functions in terms of some identities, use `expand_func()` (page 144). For example the `gamma` function `gamma` (page 358) can be expanded as

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

This method also can help if you would like to rewrite the generalized hypergeometric function `hyper` (page 404) or the Meijer G-function `meijerg` (page 406) in terms of more standard functions.

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

To simplify combinatorial expressions, involving `factorial` (page 345), `binomial` (page 341) or `gamma` (page 358) — use `combsimp()` (page 940) 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)
```

2.6 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 132) function.

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

`diff()` (page 132) can take multiple derivatives at once. To take multiple derivatives, pass the variable as many times as you wish to differentiate, or pass a number after the variable. For example, both of the following find the third derivative of x^4 .

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

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

```
>>> expr = exp(x*y*z)
>>> diff(expr, x, y, y, z, z, z, z)
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 132) can also be called as a method `diff()` (page 70). The two ways of calling `diff()` (page 132) 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 129) class. It has the same syntax as *diff()* (page 132).

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

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 45) 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 546) function. There are two kinds of integrals, definite and indefinite. To compute an indefinite integral, do

```
>>> integrate(cos(x), 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 546) is unable to compute an integral, it returns an unevaluated `Integral` (page 553) object.

```
>>> integrate(x**x, x)
|
| x
| x dx
|
>>> print(_)
Integral(x**x, x)
```

As with `Derivative` (page 129), you can create an unevaluated integral directly. To later evaluate this integral, call `doit()` (page 554).

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

`integrate()` (page 546) 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)), x)
|
| x ( x 2      x      x 4 2      )
| e · (e · x  - 2·e · x  - e  + x  - x  - 2·x)
| ----- dx
|      ( x      )      2      2
|      (e  + 1) · (x - 1) · (x + 1)
|
>>> _.doit()
x
e
----- + log(e + 1)
2
x - 1
```

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), x)
|
| sin(x2) dx
|
>>> _.doit()
_____ ( |√2 · x| )
```


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

```
>>> Integral(x**y*exp(-x), (x, 0, oo))
∫
  -x y
  e  ·x  dx
0
>>> _.doit()
Γ(y + 1)    for -re(y) < 1
∫
  -x y
  e  ·x  dx    otherwise
0
```

This last example returned a *Piecewise* (page 332) expression because the integral does not converge unless $\Re(y) > 1$.

2.6.3 Sums and Products

Much like integrals, there are *summation()* (page 293) and *product()* (page 294) to compute sums and products respectively.

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

Unevaluated sums and products are represented by *Sum* (page 280) and *Product* (page 284) classes.

```
>>> Sum(1, (x, 1, z))
z
  \
  \  1
  /
  /
x = 1
>>> _.doit()
z
```

2.6.4 Limits

Diofant can compute symbolic limits with the `limit()` (page 902) function. To compute a directional limit

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

do

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

`limit()` (page 902) should be used instead of `subs()` (page 54) 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 129) and *Integral* (page 553), `limit()` (page 902) has an unevaluated counterpart, `Limit` (page 902). To evaluate it, use `doit()` (page 902).

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

To change side, pass '-' as a third argument to `limit()` (page 902). For example, to compute

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

do

```
>>> limit(1/x, x, 0, dir='-')
-∞
```

You can also evaluate bidirectional limits

```
>>> limit(sin(x)/x, x, 0, dir='real')
1
>>> limit(1/x, x, 0, dir='real')
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.

```
>>> series(exp(sin(x)), x, 0, 4)
      2
      x  ( 4)
1 + x + -- + 0(x )
      2
```

The $O(x^4)$ term, an instance of O (page 903) 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 903).

```
>>> x + x**3 + x**6 + O(x**4)
      3  ( 4)
x + x  + 0(x )
>>> x*O(1)
0(x)
```

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

```
>>> series(exp(x), x, 0, 3).removeO()
      2
      x
-- + x + 1
      2
```

The O (page 903) notation supports arbitrary limit points (other than 0):

```
>>> series(exp(x - 1), x, x0=1)
      2      3      4      5
(x - 1) (x - 1) (x - 1) (x - 1)
----- + ----- + ----- + ----- + x + 0((x - 1) ; x → 1)
      2          6          24         120
```

2.7 Solvers

This section covers equations solving.

Note: Any expression in input, that not in an `Eq` (page 110) is automatically assumed to equal 0 by the solving functions.

2.7.1 Algebraic Equations

The main function for solving algebraic equations is `solve()` (page 1054).

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

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

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

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

`solve()` (page 1054) can also solve systems of equations.

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

If you want a dictionary output, pass `dict=True`.

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

`solve()` (page 1054) reports each solution only once.

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

To get the solutions of a polynomial including multiplicity use `roots()` (page 734).

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

2.7.2 Differential Equations

To solve differential equations, use `dsolve()` (page 994). First, create an undefined function by passing `cls=Function` to the `symbols()` (page 83) function.

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

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

```
>>> f(x).diff(x)
d
--(f(x))
dx
```

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

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

To solve the ODE, pass it and the function to solve for to `dsolve()` (page 994).

```
>>> dsolve(_, f(x))
      x          cos(x)
f(x) = e · (C$_1$ + C$_2$·x) + -----
                2
```

dsolve() (page 994) returns an instance of *Eq* (page 110). This is because in general, solutions to differential equations cannot be solved explicitly for the function.

```
>>> dsolve(f(x).diff(x)*(1 - sin(f(x))), f(x))
f(x) + cos(f(x)) = C$_1$
```

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

dsolve() (page 994) can also solve systems of equations, like *solve()* (page 1054).

```
>>> dsolve([f(x).diff(x) - g(x), g(x).diff(x) - f(x)], [f(x), g(x)])
[
  ( x  -x)   ( x  -x)   ( x  -x)   ( x  -x)
  | e   e |   | e   e |   | e   e |   | e   e |
f(x) = C$_1$·| -- + --- | + C$_2$·| -- - --- |, g(x) = C$_1$·| -- - --- | + C$_2$·| -- + ---
  ||
  | (2  2 )   (2  2 )   (2  2 )   (2  2 ) |
```

2.8 Matrices

To make a matrix in Diofant, use the *Matrix* (page 577) object. A matrix is constructed by providing a list of row vectors that make up the matrix. For example, to construct the matrix

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

use

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

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

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

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 653) in places that require immutability, such as inside other Diofant expressions or as keys to dictionaries.

2.8.1 Indexing

Diofant matrices support subscription of matrix elements with pair of integers or `slice` instances. In last case, new *Matrix* (page 577) instances will be returned.

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

To get an individual row or column of a matrix, you could also use methods `row()` (page 632) or `col()` (page 631).

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

2.8.2 Reshape and Rearrange

To get the shape of a matrix use `shape` (page 617) 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 methods `row_del()` (page 635) or `col_del()` (page 633).

```

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

```

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

To insert rows or columns, use methods `row_insert()` (page 616) or `col_insert()` (page 594).

```

>>> 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]

```

To swap two given rows or columns, use methods `row_swap()` (page 636) or `col_swap()` (page 633).

```

>>> M.row_swap(0, 1)
>>> M
[-2 0 4]
[1 2 3]
>>> M.col_swap(1, 2)
>>> M
[-2 4 0]
[1 3 2]

```

To take the transpose of a Matrix, use `T` (page 590) property.

```

>>> M.T
[-2 1]
[4 3]
[0 2]

```

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

2.8.4 Special Matrices

Several constructors exist for creating common matrices. To create an identity matrix, use `eye()` (page 622) 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 622) function.


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

Similarly, function `ones()` (page 622) creates a matrix of ones.

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

To create diagonal matrices, use function `diag()` (page 622). 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]
```

2.8.5 Advanced Methods

To compute the determinant of a matrix, use `det()` (page 595) method.

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

To put a matrix into reduced row echelon form, use method `rref()` (page 616). 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.

```

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

To find the nullspace of a matrix, use method *nullspace()* (page 612). It returns a list of column vectors that span the nullspace of the matrix.

```

>>> M = Matrix([[1, 2, 3, 0, 0], [4, 10, 0, 0, 1]])
>>> M
[1  2  3  0  0]
[4 10  0  0  1]
>>> M.nullspace()
[[-15], [0], [ 1]]
[ 6, [0], [-1/2]]
[ 1, [0], [ 0]]
[ 0, [1], [ 0]]
[ 0, [0], [ 1]]
```

To find the eigenvalues of a matrix, use method *eigenvals()* (page 597). It returns a dictionary of roots including its multiplicity (similar to the output of *roots()* (page 734) function).

```

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

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

Matrices can have symbolic elements.

```

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



```
>>> M = Matrix([[ -2, 4], [1, 3]])
>>> P, J = M.jordan_form()
>>> J
```

$$\begin{bmatrix} 1 & \frac{\sqrt{41}}{2} & & \\ - & + & \frac{\sqrt{41}}{2} & 0 \\ 2 & & & \\ & & \frac{\sqrt{41}}{2} & 1 \\ 0 & - & \frac{\sqrt{41}}{2} & + \\ & & 2 & 2 \end{bmatrix}$$

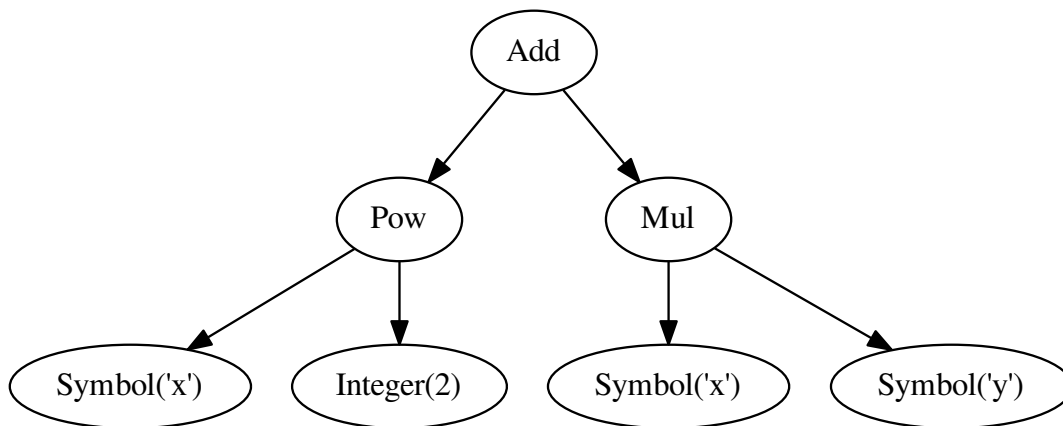
2.9 Expression Trees

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

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

```
>>> repr(x*y + x**2)
"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 81) class and the Diofant version of integers, instance of the *Integer* (page 91) 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 103).

```
>>> 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 100) class instance.

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

We could have created the same object by calling

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

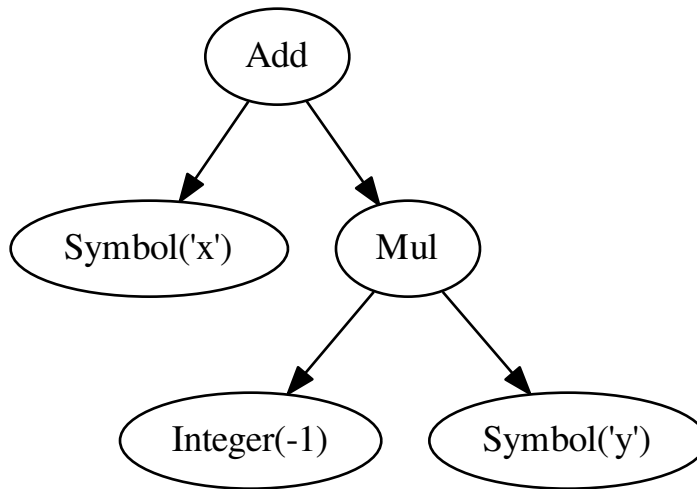
Now we get to our final expression, $x*y + x^{**2}$. This is the addition of our last two objects. The Diofant class for addition is *Add* (page 107), so, as you might expect, to create this object, we could use

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

Note: You may have noticed that the order we entered our expression and the order that it came out from printers like `repr()` or in the graph were different. This because in Diofant, the arguments of *Add* (page 107) and the commutative arguments of *Mul* (page 103) are stored in an arbitrary (but consistent!) order, which is independent of the order inputted.

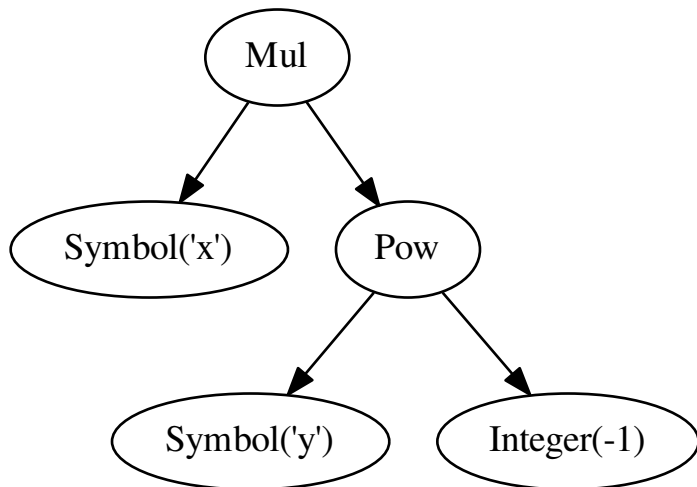
There is no subtraction class in Diofant. $x - y$ is represented as $x + (-1)*y$

```
>>> 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.

2.9.1 Walking the Tree

Let's look at how to dig our way through an expression tree. For this every object in Diofant has a very generic interface — two important attributes, *func* (page 46), and *args* (page 42).

The head of the object is encoded in the *func* (page 46) attribute. Usually it is the same as the class of the object, but not always.

```
>>> expr = 2 + x*y
>>> expr
x·y + 2
>>> expr.func
<class 'diofant.core.add.Add'>
>>> type(expr)
<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 make heavy use of 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 42) attribute.

```
>>> expr.args
(2, x·y)
```

From this, we can see *expr* can be completely reconstructed from its *func* (page 46) and its *args* (page 42).

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

Note: Every well-formed Diofant expression must either have empty *args* (page 42) or satisfy invariant

```
>>> expr == expr.func(*expr.args)
True
```

In Diofant, empty *args* (page 42) signal that we have hit a leaf of the expression tree.

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

This interface allows us to write simple algorithms that walk expression trees.

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

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

Let's test this by printing all nodes of the expression at each level.

```
>>> expr = x*y + 2
>>> for term in pre(expr):
...     print(term)
x*y + 2
2
x*y
x
y
```

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

Such traversals are so common in Diofant that the generator functions `preorder_traversal()` (page 57) and `postorder_traversal()` (page 1157) are provided to make such traversals easy. We could have also written our test as

```
>>> for arg in preorder_traversal(expr):
...     print(arg)
x*y + 2
2
x*y
x
y
```


MODULES REFERENCE

Because every feature of Diofant must have a test case, when you are not sure how to use something, just look into the `tests/` directories, find that feature and read the tests for it, that will tell you everything you need to know.

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

Click the "modules" (modindex) link in the top right corner to easily access any Diofant module, or use this contents:

3.1 Diofant Core

3.1.1 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.

```
>>> from diofant import sympify
```

```
>>> sympify(2).is_integer
True
```

```
>>> sympify(2).is_extended_real
True
```

```
>>> sympify(2.0).is_extended_real
True
>>> sympify("2.0").is_extended_real
True
>>> sympify("2e-45").is_extended_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 `bitcount` function is treated as a symbol and the `0` is interpreted as the `Order` object (used with series) and it raises an error when used improperly:

```
>>> s = 'bitcount(42)'
>>> sympify(s)
bitcount(42)
>>> sympify("0(x)")
0(x)
>>> sympify("0 + 1")
Traceback (most recent call last):
...
TypeError: unbound method...
```

In order to have `bitcount` be recognized it can be imported into a namespace dictionary and passed as `locals`:

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

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

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

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

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

```
>>> from diofant.abc import x
>>> -2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1
-1
>>> s = '-2*(-(-x + 1/x)/(x*(x - 1/x)**2) - 1/(x*(x - 1/x))) - 1'
>>> sympify(s)
-1
>>> 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.

```
>>> from diofant import Matrix
>>> 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
...         return
...     def __getitem__(self, i): return list(self)[i]
>>> from diofant.core.sympify import converter
>>> converter[MyList2] = lambda x: Matrix(x)
>>> sympify(MyList2())
Matrix([
[1],
[2]])
```

3.1.2 assumptions

This module contains the machinery handling assumptions.

All symbolic objects have assumption attributes that can be accessed via `.is_<assumption name>` attribute, i.e. `is_integer` (page 49). Full set of defined assumption names are accessible as `Basic.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):

```
>>> from diofant import I
>>> I.is_algebraic
True
>>> I.is_real
False
>>> I.is_prime
False
```

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

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

See Also

See also:

ImaginaryUnit (page 98) *is_algebraic* (page 47) *is_real* (page 50) *is_prime* (page 50)

Notes

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

3.1.3 cache

cacheit

`diofant.core.cache.cacheit(f)`

Caching decorator.

The result of cached function must be *immutable*.

Examples

```
>>> from diofant.core.cache import cacheit
>>> from diofant.abc import x, y
```

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

```
>>> f(x, y)
x y
x + y
>>> f(x, y)
x + y
```

3.1.4 basic

Base class for all the objects in Diofant

class `diofant.core.basic.Atom`

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 45)

matches(*expr*, *repl_dict*={})

Helper method for `match()`.

See also:

[Basic.matches](#) (page 51)

sort_key(*order=None*)

Return a sort key.

xreplace(*rule, hack2=False*)

Replace occurrences of objects within the expression.

See also:

[Basic.xreplace](#) (page 56)

class `diofant.core.basic.Basic`

Base class for all objects in Diofant.

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

Examples

```
>>> from diofant import cot
>>> from diofant.abc import x, y
```

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

```
>>> cot(x).args[0]
x
```

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

```
>>> (x*y).args[1]
y
```

args

Returns a tuple of arguments of 'self'.

Notes

Only use `_args` in `__new__` when creating a new function. Don't override `.args()` from `Basic` (so that it's easy to change the interface in the future if needed).

Examples

```
>>> from diofant import cot
>>> from diofant.abc import x, y
```

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

```
>>> cot(x).args[0]
x
```

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

```
>>> (x*y).args[1]
y
```

as_content_primitive(*radical=False*)

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

See also:

[diofant.core.expr.Expr.as_content_primitive](#) (page 62)

as_poly(*gens, **args)

Converts self to a polynomial or returns None.

Examples

```
>>> from diofant import sin
>>> from diofant.abc import x, y
```

```
>>> (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
```

assumptions0

Return object *type* assumptions.

For example:

```
Symbol('x', real=True) Symbol('x', integer=True)
```

are different objects. In other words, besides Python type (Symbol in this case), the initial assumptions are also forming their typeinfo.

Examples

```
>>> from diofant import Symbol
>>> from diofant.abc import x
>>> x.assumptions0
{'commutative': True}
>>> x = Symbol("x", positive=True)
>>> x.assumptions0
{'commutative': True, 'extended_real': True, 'imaginary': False,
 'negative': False, 'nonnegative': True, 'nonpositive': False,
 'nonzero': True, 'positive': True, 'zero': False}
```

atoms(*types)

Returns the atoms that form the current object.

By default, only objects that are truly atomic and can't be divided into smaller pieces are returned: symbols, numbers, and number symbols like I and π . It is possible to request atoms of any type, however, as demonstrated below.

Examples

```
>>> from diofant import I, pi, sin
>>> from diofant.abc import x, y
>>> (1 + x + 2*sin(y + I*pi)).atoms()
{1, 2, I, pi, x, y}
```

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

```
>>> from diofant import Number, NumberSymbol, Symbol
>>> (1 + x + 2*sin(y + I*pi)).atoms(Symbol)
{x, y}
```

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

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

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

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

The type can be given implicitly, too:

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

```
>>> from diofant import S, Integer
>>> (1 + x + 2*sin(y + I*pi)).atoms(Integer(1))
{1}
```

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

Finally, arguments to `atoms()` can select more than atomic atoms: any diofant type (loaded in `core/__init__.py`) can be listed as an argument and those types of "atoms" as found in scanning the arguments of the expression recursively:

```
>>> from diofant import Function, Mul
>>> from diofant.core.function import AppliedUndef
>>> f = Function('f')
>>> (1 + f(x) + 2*sin(y + I*pi)).atoms(Function)
{f(x), sin(y + I*pi)}
```



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

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

canonical_variables

Return a dictionary mapping any variable defined in `self.variables` as underscore-suffixed numbers corresponding to their position in `self.variables`. Enough underscores are added to ensure that there will be no clash with existing free symbols.

Examples

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

classmethod class_key()

Nice order of classes.

compare(*other*)

Return -1, 0, 1 if the object is smaller, equal, or greater than *other*.

Not in the mathematical sense. If the object is of a different type from the “other” then their classes are ordered according to the `sorted_classes` list.

Examples

```
>>> from diofant.abc import x, y
>>> x.compare(y)
-1
>>> x.compare(x)
0
>>> y.compare(x)
1
```

copy()

Return shallow 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

```
>>> from diofant import Integral
>>> from diofant.abc import x
```

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

```
>>> (2*Integral(x, x)).doit()
x**2
```

```
>>> (2*Integral(x, x)).doit(deep=False)
2*Integral(x, x)
```

dummy_eq(*other*, *symbol=None*)

Compare two expressions and handle dummy symbols.

Examples

```
>>> from diofant import Dummy
>>> from diofant.abc import x, y
```

```
>>> u = Dummy('u')
```

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

```
>>> (u**2 + y).dummy_eq(x**2 + y, x)
True
>>> (u**2 + y).dummy_eq(x**2 + y, y)
False
```

find(*query*, *group=False*)

Find all subexpressions matching a query.

free_symbols

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

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

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

func

The top-level function in an expression.

The following should hold for all objects:

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

Examples

```
>>> from diofant.abc import x
>>> 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 whether any subexpression matches any of the patterns.

Examples

```
>>> from diofant import sin
>>> from diofant.abc import x, y, z
>>> (x**2 + sin(x*y)).has(z)
False
>>> (x**2 + sin(x*y)).has(x, y, z)
True
>>> x.has(x)
True
```

Note that `expr.has(*patterns)` is exactly equivalent to `any(expr.has(p) for p in patterns)`. In particular, `False` is returned when the list of patterns is empty.

```
>>> x.has()
False
```

is_algebraic

Test if self can have only values from the set of algebraic numbers [\[R31\]](#) (page 1257).

References

[\[R31\]](#) (page 1257)

is_antihermitian

Test if self belongs to the field of antihermitian operators.

is_commutative

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

is_comparable

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

Examples

```
>>> from diofant import exp_polar, pi, I
>>> (I*exp_polar(I*pi/2)).is_comparable
True
>>> (I*exp_polar(I*pi*2)).is_comparable
False
```

is_complex

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

See also:

[is_real](#) (page 50)

is_composite

Test if self is a positive integer that has at least one positive divisor other than 1 or the number itself. See [\[R32\]](#) (page 1257).

References

[\[R32\]](#) (page 1257)

is_even

Test if self can have only values from the set of even integers [\[R33\]](#) (page 1257).

See also:

[is_odd](#) (page 50)

References

[\[R33\]](#) (page 1257)

is_extended_real

Test if self can have only values on the extended real number line [\[R34\]](#) (page 1257).

See also:

[is_real](#) (page 50)

References

[\[R34\]](#) (page 1257)

is_finite

Test if self absolute value is bounded. See [\[R35\]](#) (page 1257).

References

[\[R35\]](#) (page 1257)

is_hermitian

Test if self belongs to the field of hermitian operators.

is_hypergeometric(*k*)

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

See also:

diofant.simplify.simplify.hypersimp (page 926)

is_imaginary

Test if self is an imaginary number [\[R36\]](#) (page 1257).

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

References

[\[R36\]](#) (page 1257)

is_infinite

Test if self absolute value can be arbitrarily large. See [\[R37\]](#) (page 1257), [\[R38\]](#) (page 1257).

References

[\[R37\]](#) (page 1257), [\[R38\]](#) (page 1257)

is_integer

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

is_irrational

Test if self value cannot be represented exactly by Rational, see [\[R39\]](#) (page 1257).

References

[\[R39\]](#) (page 1257)

is_negative

Test if self can have only negative values [\[R40\]](#) (page 1257).

References

[\[R40\]](#) (page 1257)

is_noninteger

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

is_nonnegative

Test if self can have only nonnegative values [\[R41\]](#) (page 1257).

See also:

is_negative (page 49)

References

[\[R41\]](#) (page 1257)

is_nonpositive

Test if self can have only nonpositive values.

is_nonzero

Test if self is nonzero.

See also:

[is_zero](#) (page 51)

is_odd

Test if self can have only values from the set of odd integers [\[R42\]](#) (page 1257).

See also:

[is_even](#) (page 48)

References

[\[R42\]](#) (page 1257)

is_polar

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

See also:

[diofant.functions.elementary.exponential.exp_polar](#) (page 330)

is_positive

Test if self can have only positive values.

is_prime

Test if self is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [\[R43\]](#) (page 1257).

References

[\[R43\]](#) (page 1257)

is_rational

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

is_real

Test if self can have only values from the set of real numbers [\[R44\]](#) (page 1257).

See also:

[is_complex](#) (page 48)

References

[\[R44\]](#) (page 1257)

is_transcendental

Test if self can have only values from the set of transcendental numbers [\[R45\]](#) (page 1257).

References

[R45] (page 1257)

`is_zero`

Test if self is zero.

See also:

`is_nonzero` (page 50)

`match(pattern)`

Pattern matching.

Wild symbols match all.

Return None when expression (self) does not match with pattern. Otherwise return a dictionary such that:

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

Examples

```
>>> from diofant import Wild
>>> from diofant.abc import x, y
>>> p = Wild("p")
>>> q = Wild("q")
>>> r = Wild("r")
>>> e = (x+y)**(x+y)
>>> e.match(p**p)
{p_: x + y}
>>> e.match(p**q)
{p_: x + y, q_: x + y}
>>> e = (2*x)**2
>>> e.match(p*q**r)
{p_: 4, q_: x, r_: 2}
>>> (p*q**r).xreplace(e.match(p*q**r))
4*x**2
```

`matches(expr, repl_dict={})`

Helper method for `match()` that looks for a match between Wild symbols in self and expressions in expr.

Examples

```
>>> from diofant import symbols, Wild, Basic
>>> a, b, c = symbols('a b c')
>>> x = Wild('x')
>>> Basic(a + x, x).matches(Basic(a + b, c)) is None
True
>>> Basic(a + x, x).matches(Basic(a + b + c, b + c))
{x_: b + c}
```

`rcall(*args)`

Apply on the argument recursively through the expression tree.

This method is used to simulate a common abuse of notation for operators. For instance in Diofant the the following will not work:

```
(x+Lambda(y,2*y))(z) == x+2*z,
```

however you can use

```
>>> from diofant import Lambda
>>> from diofant.abc import x, y, z
>>> (x + Lambda(y, 2*y)).rcall(z)
x + 2*z
```

replace(*query, value, map=False, simultaneous=True, exact=False*)

Replace matching subexpressions of self with value.

If `map = True` then also return the mapping `{old: new}` where `old` was a sub-expression found with `query` and `new` is the replacement value for it. If the expression itself doesn't match the query, then the returned value will be `self.xreplace(map)` otherwise it should be `self.subs(ordered(map.items()))`.

Traverses an expression tree and performs replacement of matching subexpressions from the bottom to the top of the tree. The default approach is to do the replacement in a simultaneous fashion so changes made are targeted only once. If this is not desired or causes problems, `simultaneous` can be set to `False`. In addition, if an expression containing more than one Wild symbol is being used to match subexpressions and the `exact` flag is `True`, then the match will only succeed if non-zero values are received for each Wild that appears in the match pattern.

The list of possible combinations of queries and replacement values is listed below:

See also:

subs (page 54) substitution of subexpressions as defined by the objects themselves.

xreplace (page 56) exact node replacement in expr tree; also capable of using matching rules

Examples

Initial setup

```
>>> from diofant import log, sin, cos, tan, Wild, Mul, Add
>>> from diofant.abc import x, y
>>> f = log(sin(x)) + tan(sin(x**2))
```

1.1. type -> type `obj.replace(type, newtype)`

When object of type `type` is found, replace it with the result of passing its argument(s) to `newtype`.

```
>>> f.replace(sin, cos)
log(cos(x)) + tan(cos(x**2))
>>> sin(x).replace(sin, cos, map=True)
(cos(x), {sin(x): cos(x)})
>>> (x*y).replace(Mul, Add)
x + y
```


1.2. type -> func obj.replace(type, func)

When object of type `type` is found, apply `func` to its argument(s). `func` must be written to handle the number of arguments of `type`.

```
>>> f.replace(sin, lambda arg: sin(2*arg))
log(sin(2*x)) + tan(sin(2*x**2))
>>> (x*y).replace(Mul, lambda *args: sin(2*Mul(*args)))
sin(2*x*y)
```

2.1. pattern -> expr obj.replace(pattern(wild), expr(wild))

Replace subexpressions matching `pattern` with the expression written in terms of the Wild symbols in `pattern`.

```
>>> a = Wild('a')
>>> f.replace(sin(a), tan(a))
log(tan(x)) + tan(tan(x**2))
>>> f.replace(sin(a), tan(a/2))
log(tan(x/2)) + tan(tan(x**2/2))
>>> f.replace(sin(a), a)
log(x) + tan(x**2)
>>> (x*y).replace(a*x, a)
y
```

When the default value of `False` is used with patterns that have more than one Wild symbol, non-intuitive results may be obtained:

```
>>> b = Wild('b')
>>> (2*x).replace(a*x + b, b - a)
2/x
```

For this reason, the `exact` option can be used to make the replacement only when the match gives non-zero values for all Wild symbols:

```
>>> (2*x + y).replace(a*x + b, b - a, exact=True)
y - 2
>>> (2*x).replace(a*x + b, b - a, exact=True)
2*x
```

2.2. pattern -> func obj.replace(pattern(wild), lambda wild: expr(wild))

All behavior is the same as in 2.1 but now a function in terms of pattern variables is used rather than an expression:

```
>>> f.replace(sin(a), lambda a: sin(2*a))
log(sin(2*x)) + tan(sin(2*x**2))
```

3.1. func -> func obj.replace(filter, func)

Replace subexpression `e` with `func(e)` if `filter(e)` is `True`.

```
>>> g = 2*sin(x**3)
>>> g.replace(lambda expr: expr.is_Number, lambda expr: expr**2)
4*sin(x**9)
```

The expression itself is also targeted by the query but is done in such a fashion that changes are not made twice.

```
>>> e = x*(x*y + 1)
>>> e.replace(lambda x: x.is_Mul, lambda x: 2*x)
2*x*(2*x*y + 1)
```

rewrite(*args, **hints)

Rewrite functions in terms of other functions.

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

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

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

Examples

```
>>> from diofant import sin, exp
>>> from diofant.abc import x
```

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

```
>>> from diofant.core import S, I, Rational, Symbol
```

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

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

subs(*args, **kwargs)

Substitutes old for new in an expression after sympifying args.

args is either:

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

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

See also:

[replace](#) (page 52) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

[xreplace](#) (page 56) exact node replacement in expr tree; also capable of using matching rules

[diofant.core.evalf.EvalfMixin.evalf](#) (page 147) calculates the given formula to a desired level of precision

Examples

```
>>> from diofant import pi, exp, limit, oo
>>> from diofant.abc import x, y
>>> (1 + x*y).subs(x, pi)
pi*y + 1
>>> (1 + x*y).subs({x:pi, y:2})
1 + 2*pi
>>> (1 + x*y).subs([(x, pi), (y, 2)])
1 + 2*pi
>>> reps = [(y, x**2), (x, 2)]
>>> (x + y).subs(reps)
6
>>> (x + y).subs(reversed(reps))
x**2 + 2
```

```
>>> (x**2 + x**4).subs(x**2, y)
y**2 + y
```

To replace only the `x**2` but not the `x**4`, use `xreplace`:

```
>>> (x**2 + x**4).xreplace({x**2: y})
x**4 + y
```

To delay evaluation until all substitutions have been made, set the keyword `simultaneous` to `True`:

```
>>> (x/y).subs([(x, 0), (y, 0)])
0
>>> (x/y).subs([(x, 0), (y, 0)], simultaneous=True)
nan
```

This has the added feature of not allowing subsequent substitutions to affect those already made:

```
>>> ((x + y)/y).subs({x + y: y, y: x + y})
1
>>> ((x + y)/y).subs({x + y: y, y: x + y}, simultaneous=True)
y/(x + y)
```

In order to obtain a canonical result, unordered iterables are sorted by `count_op` length, number of arguments and by the `default_sort_key` to break any ties. All other iterables are left unsorted.

```
>>> from diofant import sqrt, sin, cos
>>> from diofant.abc import a, b, c, d, e
```

```
>>> expr = sqrt(sin(2*x))*sin(exp(x)*x)*cos(2*x) + sin(2*x)
```

```
>>> expr.subs({sqrt(sin(2*x)): a, sin(2*x): b,
...          cos(2*x): c, x: d, exp(x): e})
a*c*sin(d*e) + b
```

The resulting expression represents a literal replacement of the old arguments with the new arguments. This may not reflect the limiting behavior of the expression:

```
>>> (x**3 - 3*x).subs({x: oo})
nan
```

```
>>> limit(x**3 - 3*x, x, oo)
oo
```

If the substitution will be followed by numerical evaluation, it is better to pass the substitution to `evalf` as

```
>>> (1/x).evalf(subs={x: 3.0}, n=21)
0.3333333333333333333333333333333333
```

rather than

```
>>> (1/x).subs({x: 3.0}).evalf(21)
0.333333333333333333333333333314830
```

as the former will ensure that the desired level of precision is obtained.

xreplace(rule)

Replace occurrences of objects within the expression.

Parameters rule : dict-like

Expresses a replacement rule

Returns xreplace : the result of the replacement

See also:

replace (page 52) replacement capable of doing wildcard-like matching, parsing of match, and conditional replacements

subs (page 54) substitution of subexpressions as defined by the objects themselves.

Examples

```
>>> from diofant import symbols, pi, exp
>>> x, y, z = symbols('x y z')
>>> (1 + x*y).xreplace({x: pi})
pi*y + 1
>>> (1 + x*y).xreplace({x: pi, y: 2})
1 + 2*pi
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> (x*y + z).xreplace({x*y: pi})
z + pi
>>> (x*y*z).xreplace({x*y: pi})
x*y*z
>>> (2*x).xreplace({2*x: y, x: z})
y
>>> (2*2*x).xreplace({2*x: y, x: z})
4*z
>>> (x + y + 2).xreplace({x + y: 2})
x + y + 2
>>> (x + 2 + exp(x + 2)).xreplace({x + 2: y})
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:

```
>>> from diofant import Integral
>>> Integral(x, (x, 1, 2*x)).xreplace({x: y})
Integral(y, (y, 1, 2*y))
```

Trying to replace x with an expression raises an error:

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

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

```
>>> from diofant import symbols
>>> from diofant.core.basic import preorder_traversal
>>> x, y, z = symbols('x y z')
```

The nodes are returned in the order that they are encountered unless key is given; simply passing key=True will guarantee that the traversal is unique.

```
>>> list(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

```
>>> from diofant.core import symbols
>>> from diofant.core.basic import preorder_traversal
>>> x, y, z = symbols('x y z')
>>> 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
```

3.1.5 core

3.1.6 singleton

S

diofant.core.singleton.S

3.1.7 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

```
>>> from diofant.abc import x
>>> from diofant.core.evaluate import evaluate
>>> print(x + x)
2*x
>>> with evaluate(False):
...     print(x + x)
x + x
```

3.1.8 expr

3.1.9 Expr

class diofant.core.expr.Expr

Base class for algebraic expressions.

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

See also:

[diofant.core.basic.Basic](#) (page 42)

adjoint()

Compute conjugate transpose or Hermite conjugation.

See also:

[diofant.functions.elementary.complexes.adjoint](#) (page 310)

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.

```
>>> from diofant import symbols, oo
>>> A, B = symbols('A B', commutative=0)
>>> x, y = symbols('x y')
>>> (-2*x*y).args_cnc()
[[-1, 2, x, y], []]
>>> (-2.5*x).args_cnc()
[[-1, 2.5, x], []]
>>> (-2*x*A*B*y).args_cnc()
```

```

[[-1, 2, x, y], [A, B]]
>>> (-2*x*A*B*y).args_cnc(split_1=False)
[[-2, x, y], [A, B]]
>>> (-2*x*y).args_cnc(cset=True)
[[-1, 2, x, y], []]

```

The arg is always treated as a Mul:

```

>>> (-2 + x + A).args_cnc()
[[], [x - 2 + A]]
>>> (-oo).args_cnc() # -oo is a singleton
[[-1, oo], []]

```

as_base_exp()

Return base and exp of self.

See also:

[diofant.core.power.Pow.as_base_exp](#) (page 101)

as_coeff_Add()

Efficiently extract the coefficient of a summation.

as_coeff_Mul(rational=False)

Efficiently extract the coefficient of a product.

as_coeff_add(*deps)

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

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

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

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

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

```

>>> from diofant import S
>>> from diofant.abc import x, y
>>> (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)

$c*x**e \rightarrow c, e$ 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)`

```
>>> from diofant import S
>>> from diofant.abc import x, y
>>> (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.

See also:

`coeff` (page 68) return sum of terms have a given factor

`as_coeff_Add` (page 60) separate the additive constant from an expression

`as_coeff_Mul` (page 60) separate the multiplicative constant from an expression

`as_independent` (page 63) separate x-dependent terms/factors from others

`diofant.polys.polytools.Poly.coeff_monomial` (page 695) efficiently find the single coefficient of a monomial in Poly

`diofant.polys.polytools.Poly.nth` (page 715) like `coeff_monomial` but powers of monomial terms are used

Examples

```
>>> from diofant import E, pi, sin, I, Poly
>>> from diofant.abc import x
```

```
>>> E.as_coefficient(E)
1
>>> (2*E).as_coefficient(E)
2
>>> (2*sin(E)*E).as_coefficient(E)
```

Two terms have E in them so a sum is returned. (If one were desiring the coefficient of the term exactly matching E then the constant from the returned expression could be selected. Or, for greater precision, a method of Poly can be used to indicate the desired term from which the coefficient is desired.)

```
>>> (2*E + x*E).as_coefficient(E)
x + 2
>>> _.args[0] # just want the exact match
2
>>> p = Poly(2*E + x*E); p
Poly(x*E + 2*E, x, E, domain='ZZ')
>>> p.coeff_monomial(E)
2
>>> p.nth(0, 1)
2
```

Since the following cannot be written as a product containing E as a factor, None is returned. (If the coefficient 2*x is desired then the coeff method should be used.)

```
>>> (2*E*x + x).as_coefficient(E)
>>> (2*E*x + x).coeff(E)
2*x
```

```
>>> (E*(x + 1) + x).as_coefficient(E)
```

```
>>> (2*pi*I).as_coefficient(pi*I)
2
>>> (2*I).as_coefficient(pi*I)
```

as_coefficients_dict()

Return a dictionary mapping terms to their Rational coefficient. Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```
>>> from diofant.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

as_content_primitive(radical=False)

This method should recursively remove a Rational from all arguments and return that (content) and the new self (primitive). The content should always be positive and `Mul(*foo.as_content_primitive()) == foo`. The primitive need no be in canonical form and should try to preserve the underlying structure if possible (i.e. `expand_mul` should not be applied to self).

Examples

```
>>> from diofant import sqrt
>>> from diofant.abc import x, y, z
```

```
>>> eq = 2 + 2*x + 2*y*(3 + 3*y)
```

The `as_content_primitive` function is recursive and retains structure:

```
>>> eq.as_content_primitive()
(2, x + 3*y*(y + 1) + 1)
```

Integer powers will have Rationals extracted from the base:

```
>>> ((2 + 6*x)**2).as_content_primitive()
(4, (3*x + 1)**2)
>>> ((2 + 6*x)**(2*y)).as_content_primitive()
(1, (2*(3*x + 1))**(2*y))
```

Terms may end up joining once their `as_content_primitives` are added:

```
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y)).as_content_primitive()
(11, x*(y + 1))
>>> ((3*(x*(1 + y)) + 2*x*(3 + 3*y)).as_content_primitive()
(9, x*(y + 1))
>>> ((3*(z*(1 + y)) + 2.0*x*(3 + 3*y)).as_content_primitive()
(1, 6.0*x*(y + 1) + 3*z*(y + 1))
>>> ((5*(x*(1 + y)) + 2*x*(3 + 3*y))**2).as_content_primitive()
(121, x**2*(y + 1)**2)
>>> ((5*(x*(1 + y)) + 2.0*x*(3 + 3*y))**2).as_content_primitive()
(1, 121.0*x**2*(y + 1)**2)
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

`as_expr(*gens)`

Convert a polynomial to a Diofant expression.

Examples

```
>>> from diofant import sin
>>> from diofant.abc import x, y
```

```
>>> f = (x**2 + x*y).as_poly(x, y)
>>> f.as_expr()
x**2 + x*y
```

```
>>> sin(x).as_expr()
sin(x)
```

`as_independent(*deps, **hint)`

A mostly naive separation of a Mul or Add into arguments that are not 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 $\text{self} = i + d$
- if self is a Mul then $\text{self} = i*d$
- if self is anything else, either tuple (self, S.One) or (S.One, self) is returned.

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

See also:

[diofant.simplify.simplify.separatevars](#) (page 924), [expand](#) (page 70), [diofant.core.add.Add.as_two_terms](#) (page 108), [diofant.core.mul.Mul.as_two_terms](#) (page 104), [as_coeff_add](#) (page 60), [as_coeff_mul](#) (page 60)

Examples

- self is an Add

```
>>> from diofant import sin, cos, exp
>>> from diofant.abc import x, y, z
```

```
>>> (x + x*y).as_independent(x)
(0, x*y + x)
>>> (x + x*y).as_independent(y)
(x, x*y)
>>> (2*x*sin(x) + y + x + z).as_independent(x)
(y + z, 2*x*sin(x) + x)
>>> (2*x*sin(x) + y + x + z).as_independent(x, y)
(z, 2*x*sin(x) + x + y)
```

- self is a Mul

```
>>> (x*sin(x)*cos(y)).as_independent(x)
(cos(y), x*sin(x))
```

non-commutative terms cannot always be separated out when self is a Mul

```
>>> from diofant import symbols
>>> n1, n2, n3 = symbols('n1 n2 n3', commutative=False)
>>> (n1 + n1*n2).as_independent(n2)
(n1, n1*n2)
>>> (n2*n1 + n1*n2).as_independent(n2)
(0, n1*n2 + n2*n1)
```

```
>>> (n1*n2*n3).as_independent(n1)
(1, n1*n2*n3)
>>> (n1*n2*n3).as_independent(n2)
(n1, n2*n3)
>>> ((x-n1)*(x-y)).as_independent(x)
(1, (x - y)*(x - n1))
```

- self is anything else:

```
>>> (sin(x)).as_independent(x)
(1, sin(x))
>>> (sin(x)).as_independent(y)
(sin(x), 1)
>>> exp(x+y).as_independent(x)
(1, 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

```
>>> from diofant import Integral
>>> I = Integral(x, (x, 1, 2))
>>> I.has(x)
True
>>> x in I.free_symbols
False
>>> I.as_independent(x) == (I, 1)
True
>>> (I + x).as_independent(x) == (I, x)
True
```

Note: when trying to get independent terms, a separation method might need to be used first. In this case, it is important to keep track of what you send to this routine so you know how to interpret the returned values

```
>>> from diofant import separatevars, log
>>> separatevars(exp(x+y)).as_independent(x)
(E**y, E**x)
>>> (x + x*y).as_independent(y)
```

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

as_leading_term(*symbols)

Returns the leading (nonzero) term of the series expansion of self.

The `_eval_as_leading_term` routines are used to do this, and they must always return a non-zero value.

Examples

```
>>> from diofant.abc import x
>>> (1 + x + x**2).as_leading_term(x)
1
>>> (1/x**2 + x + x**2).as_leading_term(x)
x**(-2)
```

as_numer_denom()

expression -> a/b -> a, b

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

See also:

normal (page 77) return a/b instead of a, b

as_ordered_factors(order=None)

Return list of ordered factors (if Mul) else [self].

as_ordered_terms(order=None, data=False)

Transform an expression to an ordered list of terms.

Examples

```
>>> from diofant import sin, cos
>>> from diofant.abc import x
```

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

as_powers_dict()

Return self as a dictionary of factors with each factor being treated as a power. The keys are the bases of the factors and the values, the corresponding exponents. The resulting dictionary should be used with caution if the expression is a Mul and contains non-commutative factors since the order that they appeared will be lost in the dictionary.

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

Performs complex expansion on 'self' and returns a tuple containing collected both real and imaginary parts. This method can't be confused with `re()` and `im()` functions, which does not perform complex expansion at evaluation.

However it is possible to expand both `re()` and `im()` functions and get exactly the same results as with a single call to this function.

```
>>> from diofant import symbols, I
```

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

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

```
>>> from diofant.abc import z, w
```

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

as_terms()

Transform an expression to a list of terms.

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

Returns asymptotic expansion for "self". See [\[R48\]](#) (page 1257)

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 normalised representation of the `mrv` set and rewrites `f` using this normalised representation. Use the `bound` parameter to give limit on rewriting coefficients in its normalised 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`.

Notes

This algorithm is directly induced from the limit computational algorithm provided by Gruntz. It majorly uses the `mrv` 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

[\[R46\]](#) (page 1257), [\[R47\]](#) (page 1257), [\[R48\]](#) (page 1257)

Examples

```
>>> from diofant import sin, exp
>>> from diofant.abc import x, y
>>> e = sin(1/x + exp(-x)) - sin(1/x)
>>> e.aseeries(x)
E**(-x)*(1/(24*x**4) - 1/(2*x**2) + 1 + 0(x**(-6), (x, oo)))
>>> e.aseeries(x, n=3, hir=True)
-E**(-2*x)*sin(1/x)/2 + E**(-x)*cos(1/x) + 0(E**(-3*x), (x, oo))
```

```
>>> e = exp(exp(x)/(1 - 1/x))
>>> e.aseeries(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.aseeries(x)
E**(E**x/(1 - 1/x))
```

cancel(*gens, **args)

See the cancel function in diofant.polys

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

Returns the coefficient from the term(s) containing x^n or None. If n is zero then all terms independent of x will be returned.

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

See also:

as_coefficient (page 61) separate the expression into a coefficient and factor

as_coeff_Add (page 60) separate the additive constant from an expression

as_coeff_Mul (page 60) separate the multiplicative constant from an expression

as_independent (page 63) separate x-dependent terms/factors from others

diofant.polys.polytools.Poly.coeff_monomial (page 695) efficiently find the single coefficient of a monomial in Poly

diofant.polys.polytools.Poly.nth (page 715) like coeff_monomial but powers of monomial terms are used

Examples

```
>>> from diofant import symbols
>>> from diofant.abc import x, y, z
```

You can select terms that have an explicit negative in front of them:

```
>>> (-x + 2*y).coeff(-1)
x
>>> (x - 2*y).coeff(-1)
2*y
```

You can select terms with no Rational coefficient:


```
>>> (x + 2*y).coeff(1)
x
>>> (3 + 2*x + 4*x**2).coeff(1)
0
```

You can select terms independent of x by making $n=0$; in this case `expr.as_independent(x)[0]` is returned (and 0 will be returned instead of None):

```
>>> (3 + 2*x + 4*x**2).coeff(x, 0)
3
>>> eq = ((x + 1)**3).expand() + 1
>>> eq
x**3 + 3*x**2 + 3*x + 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 2]
>>> eq -= 2
>>> [eq.coeff(x, i) for i in reversed(range(4))]
[1, 3, 3, 0]
```

You can select terms that have a numerical term in front of them:

```
>>> (-x - 2*y).coeff(2)
-y
>>> from diofant import sqrt
>>> (x + sqrt(2)*x).coeff(sqrt(2))
x
```

The matching is exact:

```
>>> (3 + 2*x + 4*x**2).coeff(x)
2
>>> (3 + 2*x + 4*x**2).coeff(x**2)
4
>>> (3 + 2*x + 4*x**2).coeff(x**3)
0
>>> (z*(x + y)**2).coeff((x + y)**2)
z
>>> (z*(x + y)**2).coeff(x + y)
0
```

In addition, no factoring is done, so $1 + z*(1 + y)$ is not obtained from the following:

```
>>> (x + z*(x + x*y)).coeff(x)
1
```

If such factoring is desired, `factor_terms` can be used first:

```
>>> from diofant import factor_terms
>>> factor_terms(x + z*(x + x*y)).coeff(x)
z*(y + 1) + 1
```

```
>>> n, m, o = symbols('n m o', commutative=False)
>>> n.coeff(n)
1
>>> (3*n).coeff(n)
3
>>> (n*m + m*n*m).coeff(n) # = (1 + m)*n*m
1 + m
```

```
>>> (n*m + m*n*m).coeff(n, right=True) # = (1 + m)*n*m
m
```

If there is more than one possible coefficient 0 is returned:

```
>>> (n*m + m*n).coeff(n)
0
```

If there is only one possible coefficient, it is returned:

```
>>> (n*m + x*m*n).coeff(m*n)
x
>>> (n*m + x*m*n).coeff(m*n, right=1)
1
```

collect(*syms*, *func=None*, *evaluate=True*, *exact=False*, *dis-tribute_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 310)

could_extract_minus_sign()

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

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

```
>>> from diofant.abc import x, y
>>> (x-y).could_extract_minus_sign() != (y-x).could_extract_minus_sign()
True
```

count_ops(*visual=None*)

wrapper for count_ops that returns the operation count.

diff(**symbols*, ***assumptions*)

Alias for [diff\(\)](#) (page 132).

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 137)

extract_additively(*c*)

Return self - c if it's possible to subtract c from self and make all matching coefficients move towards zero, else return None.

See also:

[extract_multiplicatively](#) (page 72), [coeff](#) (page 68), [as_coefficient](#) (page 61)

Examples

```
>>> from diofant.abc import x, y
>>> e = 2*x + 3
>>> e.extract_additively(x + 1)
x + 2
>>> e.extract_additively(3*x)
>>> e.extract_additively(4)
>>> (y*(x + 1)).extract_additively(x + 1)
>>> ((x + 1)*(x + 2*y + 1) + 3).extract_additively(x + 1)
(x + 1)*(x + 2*y) + 3
```

Sometimes auto-expansion will return a less simplified result than desired; `gcd_terms` might be used in such cases:

```
>>> from diofant import gcd_terms
>>> (4*x*(y + 1) + y).extract_additively(x)
4*x*(y + 1) + x*(4*y + 3) - x*(4*y + 4) + y
>>> gcd_terms(_)
x*(4*y + 3) + y
```

extract_branch_factor(*allow_half=False*)

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

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

If `allow_half` is True, also extract `exp_polar(I*pi)`:

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

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

extract_multiplicatively(*c*)

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

```
>>> from diofant import symbols, Rational
```

```
>>> x, y = symbols('x,y', extended_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

get0()

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

getn()

Returns the order of the expression.

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

Examples

```
>>> from diofant import 0
>>> from diofant.abc import x
>>> (1 + x + 0(x**2)).getn()
2
>>> (1 + x).getn()
```

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

See the integrate function in diofant.integrals

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” [R49] (page 1257) with symbolic exponents. This is a simple extension to the `is_rational_function`, including rational exponentiation.

See also:

[is_rational_function](#) (page 75)

References

[R49] (page 1257)

Examples

```
>>> from diofant import Symbol, sqrt
>>> x = Symbol('x', extended_real=True)
>>> sqrt(1 + x).is_rational_function()
False
>>> sqrt(1 + x).is_algebraic_expr()
True
```

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

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

is_constant(*wrt, **flags)

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

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

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

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

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

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

Examples

```

>>> from diofant import cos, sin, Sum, S, pi
>>> from diofant.abc import a, n, x, y
>>> 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

```

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

```

>>> from diofant import log, Integral
>>> from diofant.abc import x

```

```

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

```

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

`is_polynomial(*syms)`

Return True if self is a polynomial in syms and False otherwise.

This checks if self is an exact polynomial in syms. This function returns False for expressions that are “polynomials” with symbolic exponents. Thus, you should be able to apply polynomial algorithms to expressions for which this returns True, and `Poly(expr, *syms)` should work if and only if `expr.is_polynomial(*syms)` returns True. The polynomial does not have to be in expanded form. If no symbols are given, all free symbols in the expression will be used.

This is not part of the assumptions system. You cannot do `Symbol('z', polynomial=True)`.

See also:

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

Examples

```
>>> from diofant import Symbol
>>> x = Symbol('x')
>>> ((x**2 + 1)**4).is_polynomial(x)
True
>>> ((x**2 + 1)**4).is_polynomial()
True
>>> (2**x + 1).is_polynomial(x)
False
```

```
>>> n = Symbol('n', nonnegative=True, integer=True)
>>> (x**n + 1).is_polynomial(x)
False
```

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

```
>>> from diofant import sqrt, factor, cancel
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)
>>> a.is_polynomial(y)
False
>>> factor(a)
y + 1
>>> factor(a).is_polynomial(y)
True
```

```
>>> b = (y**2 + 2*y + 1)/(y + 1)
>>> b.is_polynomial(y)
False
>>> cancel(b)
y + 1
>>> cancel(b).is_polynomial(y)
True
```

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)`.

See also:

[is_algebraic_expr](#) (page 72)

Examples

```
>>> from diofant import Symbol, sin
>>> from diofant.abc import x, y
```

```
>>> (x/y).is_rational_function()
True
```

```
>>> (x**2).is_rational_function()
True
```

```
>>> (x/sin(y)).is_rational_function(y)
False
```

```
>>> n = Symbol('n', integer=True)
>>> (x**n + 1).is_rational_function(x)
False
```

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

```
>>> from diofant import sqrt, factor
>>> y = Symbol('y', positive=True)
>>> a = sqrt(y**2 + 2*y + 1)/y
>>> a.is_rational_function(y)
False
>>> factor(a)
(y + 1)/y
>>> factor(a).is_rational_function(y)
True
```

limit(x, xlim, dir='+')

Compute limit $x \rightarrow xlim$.

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

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

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


```
for term in sin(x).lseries(x):
    print term
```

The advantage of `lseries()` over `nseries()` is that many times you are just interested in the next term in the series (i.e. the first term for example), but you don't know how many you should ask for in `nseries()` using the "n" parameter.

See also:

[nseries](#) (page 77)

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), [lseries](#) (page 76)

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 grunz algorithm for determining limits. `_eval_nseries` has to return a generalized power series with coefficients in $C(\log(x), \log)$:

$$c_0 * x^{e_0} + \dots \text{ (finitely many terms)}$$

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`.)

Examples

```
>>> from diofant import sin, log, Symbol
>>> from diofant.abc import x
>>> 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 $-\infty$ (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)
```

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

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

powsimp(*deep=False*, *combine='all'*)

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

```
>>> from diofant.abc import x
>>> (3*(x + 1)**2).primitive()
(3, (x + 1)**2)
>>> a = (6*x + 2); a.primitive()
(2, 3*x + 1)
>>> b = (x/2 + 3); b.primitive()
(1/2, x + 6)
>>> (a*b).primitive() == (1, a*b)
True
```

radsimp()

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

ratsimp()

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

remove0()

Removes the additive $O(\dots)$ symbol if there is one

round(*p=0*)

Return `x` rounded to the given decimal place.

If a complex number would results, apply round to the real and imaginary components of the number.

Notes

Do not confuse the Python builtin function, round, with the 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
```

Examples

```
>>> from diofant import pi, E, I, Add, Mul, Number, Float
>>> 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
```

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

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

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

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

```
>>> from diofant import cos, exp
>>> from diofant.abc import x, y
>>> cos(x).series()
1 - x**2/2 + x**4/24 + O(x**6)
>>> cos(x).series(n=4)
1 - x**2/2 + O(x**4)
```

```
>>> cos(x).series(x, x0=1, n=2)
cos(1) - (x - 1)*sin(1) + 0((x - 1)**2, (x, 1))
>>> e = cos(x + exp(y))
>>> e.series(y, n=2)
cos(x + 1) - y*sin(x + 1) + 0(y**2)
>>> e.series(x, n=2)
cos(E**y) - x*sin(E**y) + 0(x**2)
```

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

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

For `dir=+` (default) the series is calculated from the right and for `dir=-` the series from the left. For smooth functions this flag will not alter the results.

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

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 312)

trigsimp(***args*)

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

3.1.10 AtomicExpr

class `diofant.core.expr.AtomicExpr`

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

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

3.1.11 symbol

Symbol

class diofant.core.symbol.Symbol

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.

See also:

[diofant.core.assumptions](#) (page 40)

Examples

```
>>> from diofant import symbols
>>> 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) == True # multiplication by scalars is commutative
True
```

Wild

class diofant.core.symbol.Wild

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

See also:

[Symbol](#) (page 81)

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:

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

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

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

Exclude also helps remove ambiguity from matches.

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

Examples

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

matches(*expr*, *repl_dict*={})
Helper method for match().

See also:

[*diofant.core.basic.Basic.matches*](#) (page 51)

Dummy

class `diofant.core.symbol.Dummy`

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

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

If a name is not supplied then a string value of the count index will be used. This is useful when a temporary variable is needed and the name of the variable used in the expression is not important.

```
>>> Dummy()
_Dummy_10
```

See also:

[Symbol](#) (page 81)

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 81) class.

`symbols()` (page 83) function returns a sequence of symbols with names taken from *names* argument, which can be a comma or whitespace delimited string, or a sequence of strings:

```
>>> from diofant import symbols, Function

>>> x, y, z = symbols('x,y,z')
>>> a, b, c = symbols('a b c')
```

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

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

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

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

To reduce typing, range syntax is supported to create indexed symbols. Ranges are indicated by a colon and the type of range is determined by the character to the right of

the colon. If the character is a digit then all contiguous digits to the left are taken as the nonnegative starting value (or 0 if there is no digit left of the colon) and all contiguous digits to the right are taken as 1 greater than the ending value:

```
>>> symbols('x:10')
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)

>>> symbols('x5:10')
(x5, x6, x7, x8, x9)
>>> symbols('x5(:2)')
(x50, x51)

>>> symbols('x5:10,y:5')
(x5, x6, x7, x8, x9, y0, y1, y2, y3, y4)

>>> symbols(('x5:10', 'y:5'))
((x5, x6, x7, x8, x9), (y0, y1, y2, y3, y4))
```

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

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

>>> symbols(':c')
(a, b, c)

>>> symbols('a:d, x:z')
(a, b, c, d, x, y, z)

>>> symbols(('a:d', 'x:z'))
((a, b, c, d), (x, y, z))
```

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

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

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

```
>>> symbols('x((a:b))')
(x(a), x(b))
>>> symbols(r'x(:1\, :2)') # or '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', extended_real=True)
>>> x.is_extended_real and y.is_extended_real and z.is_extended_real
True
```

Despite its name, `symbols()` (page 83) 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 83) with the same arguments and puts the results into the *global* namespace. It's recommended not to use `var()` (page 85) in library code, where `symbols()` (page 83) has to be used.

Examples

```
>>> from diofant import var
```

```
>>> var('x')
x
>>> x
x
```

```
>>> var('a,ab,abc')
(a, ab, abc)
>>> abc
abc
```

```
>>> var('x,y', extended_real=True)
(x, y)
>>> x.is_extended_real and y.is_extended_real
True
```

See `symbols()` (page 83) documentation for more details on what kinds of arguments can be passed to `var()` (page 85).

3.1.12 numbers

Number

class `diofant.core.numbers.Number`

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.

If you want to represent, for example, $1+\sqrt{2}$, then you need to do:

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

as_coeff_Add()

Efficiently extract the coefficient of a summation.

as_coeff_Mul(*rational=False*)

Efficiently extract the coefficient of a product.

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 60)

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 60)

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*.

is_constant(**wrt*, ***flags*)

Return True if self is constant.

See also:

[diofant.core.expr.Expr.is_constant](#) (page 73)

lcm(*other*)

Compute LCM of *self* and *other*.

sort_key(*order=None*)

Return a sort key.

Float

class `diofant.core.numbers.Float`

Represent a floating-point number of arbitrary precision.

Notes

Floats are inexact by their nature unless their value is a binary-exact value.

```
>>> approx, exact = Float(.1, 1), Float(.125, 1)
```

For calculation purposes, evalf needs to be able to change the precision but this will not increase the accuracy of the inexact value. The following is the most accurate 5-digit approximation of a value of 0.1 that had only 1 digit of precision:

```
>>> approx.evalf(5)
0.099609
```

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

```
>>> Float(exact, 5)
0.12500
>>> exact.evalf(20)
0.12500000000000000000
```

Trying to make a high-precision Float from a float is not disallowed, but one must keep in mind that the *underlying float* (not the apparent decimal value) is being obtained with high precision. For example, 0.3 does not have a finite binary representation. The closest rational is the fraction $5404319552844595/2^{54}$. So if you try to obtain a Float of 0.3 to 20 digits of precision you will not see the same thing as 0.3 followed by 19 zeros:

```
>>> Float(0.3, 20)
0.299999999999999998890
```

If you want a 20-digit value of the decimal 0.3 (not the floating point approximation of 0.3) you should send the 0.3 as a string. The underlying representation is still binary but a higher precision than Python's float is used:

```
>>> Float('0.3', 20)
0.30000000000000000000
```

Although you can increase the precision of an existing Float using Float it will not increase the accuracy - the underlying value is not changed:

```
>>> def show(f): # binary rep of Float
...     from diofant import Mul, Pow
...     s, m, e, b = f._mpf_
...     v = Mul(int(m), Pow(2, int(e), evaluate=False), evaluate=False)
...     print('%s at prec=%s' % (v, f._prec))
...
>>> t = Float('0.3', 3)
>>> show(t)
4915/2**14 at prec=13
>>> show(Float(t, 20)) # higher prec, not higher accuracy
4915/2**14 at prec=70
>>> show(Float(t, 2)) # lower prec
307/2**10 at prec=10
```

The same thing happens when evalf is used on a Float:

```
>>> show(t.evalf(20))
4915/2**14 at prec=70
>>> show(t.evalf(2))
307/2**10 at prec=10
```

Finally, Floats can be instantiated with an mpf tuple (n, c, p) to produce the number $(-1)**n*c*2**p$:

```
>>> n, c, p = 1, 5, 0
>>> (-1)**n*c*2**p
-5
>>> Float((1, 5, 0))
-5.000000000000000
```

An actual mpf tuple also contains the number of bits in c as the last element of the tuple:

```
>>> _mpf_
(1, 5, 0, 3)
```

This is not needed for instantiation and is not the same thing as the precision. The mpf tuple and the precision are two separate quantities that Float tracks.

Examples

```
>>> from diofant import Float
>>> Float(3.5)
3.500000000000000
>>> Float(3)
3.000000000000000
```

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)
1.000000000000000
>>> Float(10**20)
10000000000000000000.
>>> Float('1e20')
10000000000000000000.
```

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

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

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

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

The desired number of digits can also be specified:

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

Float can automatically count significant figures if a null string is sent for the precision; space are also allowed in the string. (Auto-counting is only allowed for strings and ints).

```
>>> Float('123 456 789 . 123 456', '')
123456789.123456
>>> Float('12e-3', '')
0.012
>>> Float(3, '')
3.
```

If a number is written in scientific notation, only the digits before the exponent are considered significant if a decimal appears, otherwise the “e” signifies only how to move the decimal:

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

ceiling()

Compute ceiling of self.

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

Test approximate equality.

floor()

Compute floor of self.

num

Return mpmath representation of self.

Rational

class diofant.core.numbers.**Rational**

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

See also:

[diofant.core.sympify.sympify](#) (page 37), [diofant.simplify.simplify.nsimpify](#) (page 926)

Examples

```
>>> from diofant import Rational, nsimplify, sympify, pi
>>> Rational(3)
3
>>> Rational(1, 2)
1/2
```

Rational is unprejudiced in accepting input. If a float is passed, the underlying value of the binary representation will be returned:

```
>>> Rational(.5)
1/2
```

```
>>> Rational(.2)
3602879701896397/18014398509481984
```

If the simpler representation of the float is desired then consider limiting the denominator to the desired value or convert the float to a string (which is roughly equivalent to limiting the denominator to 10^{**12}):

```
>>> Rational(str(.2))
1/5
>>> Rational(.2).limit_denominator(10**12)
1/5
```

An arbitrarily precise Rational is obtained when a string literal is passed:

```
>>> Rational("1.23")
123/100
>>> Rational('1e-2')
1/100
>>> Rational(".1")
1/10
```

The conversion of other types of strings can be handled by the `sympify()` function, and conversion of floats to expressions or simple fractions can be handled with `nsimplify`:

```
>>> sympify('3**2/10') # general expressions
9/10
>>> nsimplify(.3) # numbers that have a simple form
3/10
```

But if the input does not reduce to a literal Rational, an error will be raised:

```
>>> Rational(pi)
Traceback (most recent call last):
...
TypeError: invalid input: pi
```

Low-level access numerator and denominator as `.p` and `.q`:

```
>>> r = Rational(3, 4)
>>> r
3/4
>>> r.p
3
>>> r.q
4
```

Note that `p` and `q` return integers (not Diofant Integers) so some care is needed when using them in expressions:

```
>>> r.p/r.q
0.75
```

as_content_primitive(*radical=False*)

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

See also:

[diofant.core.expr.Expr.as_content_primitive](#) (page 62)

Examples

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

as_numer_denom()
expression -> a/b -> a, b

See also:

[diofant.core.expr.Expr.as_numer_denom](#) (page 66)

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*.

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

Integer

class diofant.core.numbers.Integer

as_numer_denom()
expression -> a/b -> a, b

See also:

[diofant.core.expr.Expr.as_numer_denom](#) (page 66)

is_composite
Test if self is a positive integer that has at least one positive divisor other than 1 or the number itself. See [\[R50\]](#) (page 1257).

References

[\[R50\]](#) (page 1257)

is_even
Test if self can have only values from the set of even integers [\[R51\]](#) (page 1257).

See also:

[is_odd](#) (page 92)

References

[\[R51\]](#) (page 1257)

is_imaginary

Test if self is an imaginary number [\[R52\]](#) (page 1257).

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

References

[\[R52\]](#) (page 1257)

is_nonzero

Test if self is nonzero.

See also:

[is_zero](#) (page 92)

is_odd

Test if self can have only values from the set of odd integers [\[R53\]](#) (page 1257).

See also:

[is_even](#) (page 91)

References

[\[R53\]](#) (page 1257)

is_prime

Test if self is a natural number greater than 1 that has no positive divisors other than 1 and itself. See [\[R54\]](#) (page 1257).

References

[\[R54\]](#) (page 1257)

is_zero

Test if self is zero.

See also:

[is_nonzero](#) (page 92)

NumberSymbol

class diofant.core.numbers.NumberSymbol

approximation_interval(*number_cls*)

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

RealNumber

`diofant.core.numbers.RealNumber`
alias of `Float` (page 86)

igcd

`diofant.core.numbers.igcd(*args)`
Computes positive integer greatest common divisor.

Examples

```
>>> from diofant.core.numbers import igcd
>>> igcd(2, 4)
2
>>> igcd(5, 10, 15)
5
```

ilcm

`diofant.core.numbers.ilcm(*args)`
Computes integer least common multiple.

Examples

```
>>> from diofant.core.numbers import ilcm
>>> ilcm(5, 10)
10
>>> ilcm(7, 3)
21
>>> ilcm(5, 10, 15)
30
```

seterr

`diofant.core.numbers.seterr(divide=False)`
Should diofant raise an exception on 0/0 or return a nan?
divide == True raise an exception divide == False ... return nan

Zero

`class diofant.core.numbers.Zero`
The number zero.
Zero is a singleton, and can be accessed by `S.Zero`

References

[R55] (page 1257)

Examples

```
>>> from diofant import S, Integer, zoo
>>> Integer(0) is S.Zero
True
>>> 1/S.Zero
zoo
```

One

class diofant.core.numbers.**One**

The number one.

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

References

[R56] (page 1257)

Examples

```
>>> from diofant import S, Integer
>>> Integer(1) is S.One
True
```

NegativeOne

class diofant.core.numbers.**NegativeOne**

The number negative one.

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

See also:

One (page 94)

References

[R57] (page 1257)

Examples

```
>>> from diofant import S, Integer
>>> Integer(-1) is S.NegativeOne
True
```

Half

class diofant.core.numbers.Half

The rational number 1/2.

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

References

[R58] (page 1257)

Examples

```
>>> from diofant import S, Rational
>>> Rational(1, 2) is S.Half
True
```

NaN

class diofant.core.numbers.NaN

Not a Number.

This serves as a place holder for numeric values that are indeterminate. Most operations on NaN, produce another NaN. Most indeterminate forms, such as $0/0$ or $00 - 00$ produce NaN. Two exceptions are $0^{**}0$ and $00^{**}0$, which all produce 1 (this is consistent with Python's float).

NaN is loosely related to floating point nan, which is defined in the IEEE 754 floating point standard, and corresponds to the Python `float('nan')`. Differences are noted below.

NaN is mathematically not equal to anything else, even NaN itself. This explains the initially counter-intuitive results with `Eq` and `==` in the examples below.

NaN is not comparable so inequalities raise a `TypeError`. This is in contrast with floating point nan where all inequalities are false.

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

References

[R59] (page 1257)

Examples

```
>>> from diofant import nan, S, oo, Eq
>>> nan is S.NaN
True
>>> oo - oo
nan
>>> nan + 1
nan
>>> Eq(nan, nan) # mathematical equality
false
>>> nan == nan # structural equality
True
```

Infinity

class diofant.core.numbers.Infinity

Positive infinite quantity.

In real analysis the symbol ∞ denotes an unbounded limit: $x \rightarrow \infty$ means that x grows without bound.

Infinity is often used not only to define a limit but as a value in the affinely extended real number system. Points labeled $+\infty$ and $-\infty$ can be added to the topological space of the real numbers, producing the two-point compactification of the real numbers. Adding algebraic properties to this gives us the extended real numbers.

Infinity is a singleton, and can be accessed by `S.Infinity`, or can be imported as `oo`.

See also:

[NegativeInfinity](#) (page 96), [NaN](#) (page 95)

References

[\[R60\]](#) (page 1258)

Examples

```
>>> from diofant import oo, exp, limit, Symbol
>>> 1 + oo
oo
>>> 42/oo
0
>>> x = Symbol('x')
>>> limit(exp(x), x, oo)
oo
```

NegativeInfinity

class diofant.core.numbers.NegativeInfinity

Negative infinite quantity.

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

See also:

Infinity (page 96)

ComplexInfinity

class diofant.core.numbers.ComplexInfinity

Complex infinity.

In complex analysis the symbol ∞ , called “complex infinity”, represents a quantity with infinite magnitude, but undetermined complex phase.

ComplexInfinity is a singleton, and can be accessed by S.ComplexInfinity, or can be imported as zoo.

See also:

Infinity (page 96)

Examples

```
>>> from diofant import zoo, oo
>>> zoo + 42
zoo
>>> 42/zoo
0
>>> zoo + zoo
nan
>>> zoo*zoo
zoo
```

Exp1

class diofant.core.numbers.Exp1

The e constant.

The transcendental number $e = 2.718281828\dots$ is the base of the natural logarithm and of the exponential function, $e = \exp(1)$. Sometimes called Euler’s number or Napier’s constant.

Exp1 is a singleton, and can be accessed by S.Exp1, or can be imported as E.

References

[R61] (page 1258)

Examples

```
>>> from diofant import exp, log, E
>>> E is exp(1)
True
>>> log(E)
1
```

ImaginaryUnit

class diofant.core.numbers.ImaginaryUnit

The imaginary unit, $i = \sqrt{-1}$.

I is a singleton, and can be accessed by S.I, or can be imported as I.

References

[R62] (page 1258)

Examples

```
>>> from diofant import I, sqrt
>>> sqrt(-1)
I
>>> I*I
-1
>>> 1/I
-I
```

Pi

class diofant.core.numbers.Pi

The π constant.

The transcendental number $\pi = 3.141592654\dots$ represents the ratio of a circle's circumference to its diameter, the area of the unit circle, the half-period of trigonometric functions, and many other things in mathematics.

Pi is a singleton, and can be accessed by S.Pi, or can be imported as pi.

References

[R63] (page 1258)

Examples

```
>>> from diofant import S, pi, oo, sin, exp, integrate, Symbol
>>> S.Pi
pi
>>> pi > 3
```

```

true
>>> pi.is_irrational
True
>>> x = Symbol('x')
>>> sin(x + 2*pi)
sin(x)
>>> integrate(exp(-x**2), (x, -oo, oo))
sqrt(pi)

```

EulerGamma

class diofant.core.numbers.EulerGamma

The Euler-Mascheroni constant.

$\gamma = 0.5772157\dots$ (also called Euler's constant) is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k} - \ln n \right)$$

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

References

[R64] (page 1258)

Examples

```

>>> from diofant import S
>>> S.EulerGamma.is_irrational
>>> S.EulerGamma > 0
true
>>> S.EulerGamma > 1
false

```

Catalan

class diofant.core.numbers.Catalan

Catalan's constant.

$K = 0.91596559\dots$ is given by the infinite series

$$K = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)^2}$$

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

References

[R65] (page 1258)

Examples

```
>>> from diofant import S
>>> S.Catalan.is_irrational
>>> S.Catalan > 0
true
>>> S.Catalan > 1
false
```

GoldenRatio

class diofant.core.numbers.GoldenRatio

The golden ratio, ϕ .

$\phi = \frac{1+\sqrt{5}}{2}$ is algebraic number. Two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities, i.e. their maximum.

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

References

[R66] (page 1258)

Examples

```
>>> from diofant import S
>>> S.GoldenRatio > 1
true
>>> S.GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
>>> S.GoldenRatio.is_irrational
True
```

3.1.13 power

Pow

class diofant.core.power.Pow

Defines the expression $x^{**}y$ as “x raised to a power y”

Singleton definitions involving (0, 1, -1, oo, -oo, I, -I):

expr	value	reason
z^{**0}	1	Although arguments over 0^{**0} exist, see [2].
z^{**1}	z	
$(-oo)^{**(-1)}$	0	
$(-1)^{**-1}$	-1	
S.Zero $^{**-1}$	zoo	This is not strictly true, as 0^{**-1} may be undefined, but is convenient in some contexts where the base is assumed to be positive.
1^{**-1}	1	
oo^{**-1}	0	
0^{**oo}	0	Because for all complex numbers z near 0, $z^{**oo} \rightarrow 0$.
0^{**-oo}	zoo	This is not strictly true, as 0^{**oo} may be oscillating between positive and negative values or rotating in the complex plane. It is convenient, however, when the base is positive.
1^{**oo} 1^{**-oo} 1^{**zoo}	nan	Because there are various cases where $\lim(x(t),t)=1$, $\lim(y(t),t)=oo$ (or $-oo$), but $\lim(x(t)^{**y(t)}, t) \neq 1$. See [3].
$(-1)^{**oo}$ $(-1)^{**(-oo)}$	nan	Because of oscillations in the limit.
oo^{**oo}	oo	
oo^{**-oo}	0	
$(-oo)^{**oo}$ $(-oo)^{**-oo}$	nan	
oo^{**I} $(-oo)^{**I}$	nan	oo^{**e} could probably be best thought of as the limit of x^{**e} for real x as x tends to oo. If e is I, then the limit does not exist and nan is used to indicate that.
$oo^{**(1+I)}$ $(-oo)^{**(1+I)}$	zoo	If the real part of e is positive, then the limit of $\text{abs}(x^{**e})$ is oo. So the limit value is zoo.
$oo^{**(-1+I)}$ $(-oo)^{**(-1+I)}$	0	If the real part of e is negative, then the limit is 0.

Because symbolic computations are more flexible than floating point calculations and we prefer to never return an incorrect answer, we choose not to conform to all IEEE 754 conventions. This helps us avoid extra test-case code in the calculation of limits.

See also:

[diofant.core.numbers.Infinity](#) (page 96), [diofant.core.numbers.NegativeInfinity](#) (page 96), [diofant.core.numbers.NaN](#) (page 95)

References

[R67] (page 1258), [R68] (page 1258), [R69] (page 1258)

as_base_exp()

Return base and exp of self.

If base is 1/Integer, then return Integer, -exp. If this extra processing is not needed, the base and exp properties will give the raw arguments

Examples

```
>>> from diofant import Pow, S
>>> p = Pow(S.Half, 2, evaluate=False)
>>> p.as_base_exp()
(2, -2)
>>> p.args
(1/2, 2)
```

as_content_primitive(*radical=False*)

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

See also:

[diofant.core.expr.Expr.as_content_primitive](#) (page 62)

Examples

```
>>> from diofant import sqrt
>>> sqrt(4 + 4*sqrt(2)).as_content_primitive()
(2, sqrt(1 + sqrt(2)))
>>> sqrt(3 + 3*sqrt(2)).as_content_primitive()
(1, sqrt(3)*sqrt(1 + sqrt(2)))
```

```
>>> from diofant import expand_power_base, powsimp, Mul
>>> from diofant.abc import x, y
```

```
>>> ((2*x + 2)**2).as_content_primitive()
(4, (x + 1)**2)
>>> (4**((1 + y)/2)).as_content_primitive()
(2, 4**(y/2))
>>> (3**((1 + y)/2)).as_content_primitive()
(1, 3**((y + 1)/2))
>>> (3**((5 + y)/2)).as_content_primitive()
(9, 3**((y + 1)/2))
>>> eq = 3**(2 + 2*x)
>>> powsimp(eq) == eq
True
>>> eq.as_content_primitive()
(9, 3**(2*x))
>>> powsimp(Mul(*_))
3**(2*x + 2)
```

```
>>> eq = (2 + 2*x)**y
>>> s = expand_power_base(eq); s.is_Mul, s
(False, (2*x + 2)**y)
>>> eq.as_content_primitive()
(1, (2*(x + 1))**y)
>>> s = expand_power_base(_[1]); s.is_Mul, s
(True, 2**y*(x + 1)**y)
```

as_numer_denom()

expression -> a/b -> a, b

See also:

[diofant.core.expr.Expr.as_numer_denom](#) (page 66)

as_real_imag(*deep=True, **hints*)
Returns real and imaginary parts of self

See also:
[diofant.core.expr.Expr.as_real_imag](#) (page 67)

base
Returns base of the power expression.

classmethod class_key()
Nice order of classes.

exp
Returns exponent of the power expression.

is_constant(**wrt, **flags*)
Return True if self is constant, False if not, or None.

See also:
[diofant.core.expr.Expr.is_constant](#) (page 73)

matches(*expr, repl_dict={}*)
Helper method for match().

See also:
[diofant.core.basic.Basic.matches](#) (page 51)

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 $x^n == y$).

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

3.1.14 mul

Mul

class `diofant.core.mul.Mul`

as_base_exp()
Return base and exp of self.

See also:
[diofant.core.expr.Expr.as_base_exp](#) (page 60)

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 60)

as_content_primitive(radical=False)

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

See also:

[diofant.core.expr.Expr.as_content_primitive](#) (page 62)

Examples

```
>>> from diofant import sqrt
>>> (-3*sqrt(2)*(2 - 2*sqrt(2))).as_content_primitive()
(6, -sqrt(2)*(-sqrt(2) + 1))
```

as_numer_denom()

expression -> a/b -> a, b

See also:

[diofant.core.expr.Expr.as_numer_denom](#) (page 66)

as_ordered_factors(order=None)

Transform an expression into an ordered list of factors.

Examples

```
>>> from diofant import sin, cos
>>> from diofant.abc import x, y
```

```
>>> (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 66)

as_real_imag(deep=True, **hints)

Returns real and imaginary parts of self

See also:

[diofant.core.expr.Expr.as_real_imag](#) (page 67)

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use self.args[0];
- if you want to process the arguments of the tail then use self.as_coef_mul() which gives the head and a tuple containing the arguments of the tail when treated as a Mul.

- if you want the coefficient when self is treated as an Add then use `self.as_coeff_add()[0]`

```
>>> from diofant.abc import x, y
>>> (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. <https://github.com/sympy/sympy/issues/4596>}

```
>>> from diofant import Mul, sqrt
>>> from diofant.abc import x, y, z
>>> 2*(x + 1) # this is the 2-arg Mul behavior
2*x + 2
>>> y*(x + 1)*2
2*y*(x + 1)
>>> 2*(x + 1)*y # 2-arg result will be obtained first
y*(2*x + 2)
>>> Mul(2, x + 1, y) # all 3 args simultaneously processed
2*y*(x + 1)
>>> 2*((x + 1)*y) # parentheses can control this behavior
2*y*(x + 1)
```

Powers with compound bases may not find a single base to combine with unless all arguments are processed at once. Post-processing may be necessary in such cases. {c.f. <https://github.com/sympy/sympy/issues/5728>}

```
>>> a = sqrt(x*sqrt(y))
>>> a**3
(x*sqrt(y))**(3/2)
>>> Mul(a,a,a)
(x*sqrt(y))**(3/2)
>>> a*a*a
x*sqrt(y)*sqrt(x*sqrt(y))
>>> _.subs(a.base, z).subs(z, a.base)
(x*sqrt(y))**(3/2)
```

- If more than two terms are being multiplied then all the previous terms will be re-processed for each new argument. So if each of `a`, `b` and `c` were `Mul` (page 103) expression, then `a*b*c` (or building up the product with `*=`) will process all the arguments of `a` and `b` twice: once when `a*b` is computed and again when `c` is multiplied.

Using `Mul(a,b,c)` will process all arguments once.

- The results of Mul are cached according to arguments, so flatten will only be called once for `Mul(a, b, c)`. If you can structure a calculation so the arguments are most likely to be repeats then this can save time in computing the answer. For example, say you had a Mul, `M`, that you wished to divide by `d[i]` and multiply by `n[i]` and you suspect there are many repeats in `n`. It would be better to compute `M*n[i]/d[i]` rather than `M/d[i]*n[i]` since every time `n[i]` is a repeat, the product, `M*n[i]` will be returned without flattening - the cached value will be returned. If you divide by the `d[i]` first (and those are more unique than the `n[i]`) then that will create a new Mul, `M/d[i]` the args of which will be traversed again when it is multiplied by `n[i]`.

{c.f. <https://github.com/sympy/sympy/issues/5706>}

This consideration is moot if the cache is turned off.

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.

matches(*expr*, *repl_dict*={})
Helper method for match().

See also:

diofant.core.basic.Basic.matches (page 51)

prod

`diofant.core.mul.prod(a, start=1)`

Return product of elements of a. Start with int 1 so if only ints are included then an int result is returned.

Examples

```
>>> from diofant import prod, Integer
>>> prod(range(3))
0
>>> type(_) is int
True
>>> prod([Integer(2), 3])
6
>>> _.is_Integer
True
```

You can start the product at something other than 1:

```
>>> prod([1, 2], 3)
6
```

3.1.15 add

Add

`class diofant.core.add.Add`

`as_coeff_Add()`

Efficiently extract the coefficient of a summation.

`as_coeff_add(*deps)`

Returns a tuple (coeff, args) where self is treated as an Add and coeff is the Number term and args is a tuple of all other terms.

Examples

```
>>> from diofant.abc import x
>>> (7 + 3*x).as_coeff_add()
(7, (3*x,))
>>> (7*x).as_coeff_add()
(0, (7*x,))
```

`as_coefficients_dict()`

Return a dictionary mapping terms to their Rational coefficient.

Since the dictionary is a defaultdict, inquiries about terms which were not present will return a coefficient of 0. If an expression is not an Add it is considered to have a single term.

Examples

```
>>> from diofant.abc import a, x
>>> (3*x + a*x + 4).as_coefficients_dict()
{1: 4, x: 3, a*x: 1}
>>> _[a]
0
>>> (3*a*x).as_coefficients_dict()
{a*x: 3}
```

`as_content_primitive(radical=False)`

Return the tuple (R, self/R) where R is the positive Rational extracted from self. If radical is True (default is False) then common radicals will be removed and included as a factor of the primitive expression.

See also:

[diofant.core.expr.Expr.as_content_primitive](#) (page 62)

Examples

```
>>> from diofant import sqrt
>>> (3 + 3*sqrt(2)).as_content_primitive()
(3, 1 + sqrt(2))
```

Radical content can also be factored out of the primitive:

```
>>> (2*sqrt(2) + 4*sqrt(10)).as_content_primitive(radical=True)
(2, sqrt(2)*(1 + 2*sqrt(5)))
```

as_numer_denom()

expression -> a/b -> a, b

See also:

[diofant.core.expr.Expr.as_numer_denom](#) (page 66)

as_real_imag(deep=True, **hints)

returns a tuple representing a complex number

Examples

```
>>> from diofant import I
>>> (7 + 9*I).as_real_imag()
(7, 9)
>>> ((1 + I)/(1 - I)).as_real_imag()
(0, 1)
>>> ((1 + 2*I)*(1 + 3*I)).as_real_imag()
(-5, 5)
```

as_two_terms()

Return head and tail of self.

This is the most efficient way to get the head and tail of an expression.

- if you want only the head, use `self.args[0]`;
- if you want to process the arguments of the tail then use `self.as_coef_add()` which gives the head and a tuple containing the arguments of the tail when treated as an Add.
- if you want the coefficient when self is treated as a Mul then use `self.as_coeff_mul()[0]`

```
>>> from diofant.abc import x, y
>>> (3*x*y).as_two_terms()
(3, x*y)
```

classmethod class_key()

Nice order of classes

extract_leading_order(symbols)

Returns the leading term and its order.

Examples

```
>>> from diofant.abc import x
>>> (x + 1 + 1/x**5).extract_leading_order(x)
((x**(-5), 0(x**(-5))),)
>>> (1 + x).extract_leading_order(x)
((1, 0(1)),)
>>> (x + x**2).extract_leading_order(x)
((x, 0(x)),)
```


classmethod flatten(*seq*)

Takes the sequence “seq” of nested Adds and returns a flatten list.

Returns: (commutative_part, noncommutative_part, order_symbols)

Applies associativity, all terms are commutable with respect to addition.

See also:

[diofant.core.mul.Mul.flatten](#) (page 105)

get0()

Returns the additive O(..) symbol.

See also:

[diofant.core.expr.Expr.get0](#) (page 72)

matches(*expr*, *repl_dict*={})

Helper method for match().

See also:

[diofant.core.basic.Basic.matches](#) (page 51)

primitive()

Return (R, self/R) where R` is the Rational GCD of self`.

R is collected only from the leading coefficient of each term.

See also:

[diofant.polys.polytools.primitive](#) (page 683)

Examples

```
>>> from diofant.abc import x, y
```

```
>>> (2*x + 4*y).primitive()
(2, x + 2*y)
```

```
>>> (2*x/3 + 4*y/9).primitive()
(2/9, 3*x + 2*y)
```

```
>>> (2*x/3 + 4.2*y).primitive()
(1/3, 2*x + 12.6*y)
```

No subprocessing of term factors is performed:

```
>>> ((2 + 2*x)*x + 2).primitive()
(1, x*(2*x + 2) + 2)
```

Recursive subprocessing can be done with the `as_content_primitive()` method:

```
>>> ((2 + 2*x)*x + 2).as_content_primitive()
(2, x*(x + 1) + 1)
```

remove0()

Removes the additive O(..) symbol.

See also:

[diofant.core.expr.Expr.remove0](#) (page 78)

3.1.16 mod

Mod

class diofant.core.mod.**Mod**

Represents a modulo operation on symbolic expressions.

Receives two arguments, dividend p and divisor q .

The convention used is the same as Python's: the remainder always has the same sign as the divisor.

Examples

```
>>> from diofant.abc import x, y
>>> x**2 % y
x**2%y
>>> _.subs({x: 5, y: 6})
1
```

3.1.17 relational

Rel

diofant.core.relational.**Rel**

alias of *Relational* (page 111)

Eq

diofant.core.relational.**Eq**

alias of *Equality* (page 112)

Ne

diofant.core.relational.**Ne**

alias of *Unequality* (page 120)

Lt

diofant.core.relational.**Lt**

alias of *StrictLessThan* (page 123)

Le

diofant.core.relational.**Le**

alias of *LessThan* (page 116)

Gt

`diofant.core.relational.Gt`
alias of *StrictGreaterThan* (page 120)

Ge

`diofant.core.relational.Ge`
alias of *GreaterThan* (page 113)

Relational

class `diofant.core.relational.Relational`

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

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

`as_set()`

Rewrites univariate inequality in terms of real sets

Examples

```
>>> from diofant import Symbol, Eq
>>> x = Symbol('x', extended_real=True)
>>> (x>0).as_set()
(0, oo)
>>> Eq(x, 0).as_set()
{0}
```

`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.

lhs

The left-hand side of the relation.

reversed

Return the relationship with sides (and sign) reversed.

Examples

```
>>> from diofant import Eq
>>> from diofant.abc import x
>>> Eq(x, 1)
Eq(x, 1)
>>> _.reversed
Eq(1, x)
>>> x < 1
x < 1
>>> _.reversed
1 > x
```

rhs

The right-hand side of the relation.

Equality

class `diofant.core.relational.Equality`

An equal relation between two objects.

Represents that two objects are equal. If they can be easily shown to be definitively equal (or unequal), this will reduce to True (or False). Otherwise, the relation is maintained as an unevaluated Equality object. Use the `simplify` function on this object for more nontrivial evaluation of the equality relation.

As usual, the keyword argument `evaluate=False` can be used to prevent any evaluation.

See also:

[*diofant.logic.boolalg.Equivalent*](#) (page 573) for representing equality between two boolean expressions

Notes

This class is not the same as the `==` operator. The `==` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

If either object defines an `evalEq` method, it can be used in place of the default algorithm. If `lhs.evalEq(rhs)` or `rhs.evalEq(lhs)` returns anything other than None, that return value will be substituted for the Equality. If None is returned by `evalEq`, an Equality object will be created as usual.

Examples

```

>>> from diofant import Eq, simplify, exp, cos
>>> from diofant.abc import x, y
>>> Eq(y, x + x**2)
Eq(y, x**2 + x)
>>> Eq(2, 5)
false
>>> Eq(2, 5, evaluate=False)
Eq(2, 5)
>>> _.doit()
false
>>> Eq(exp(x), exp(x).rewrite(cos))
Eq(E**x, sinh(x) + cosh(x))
>>> simplify(_)
true

```

GreaterThan

class diofant.core.relational.GreaterThan

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:

```

>>> from diofant import GreaterThan, StrictGreaterThan
>>> from diofant import LessThan, StrictLessThan
>>> from diofant import And, Ge, Gt, Le, Lt, Rel, S, Integer
>>> from diofant.abc import x, y, z
>>> from diofant.core.relational import Relational

```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement $(1 < x)$, Python will first recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a 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      %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      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

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

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R70] (page 1258), there is no way for Diofant to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of `And`:

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

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in Diofant [R71] (page 1258)):

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

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

References

[R70] (page 1258), [R71] (page 1258)

Examples

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

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

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

Another option is to use the Python inequality operators (`>=`, `>`, `<=`, `<`) directly. Their main advantage over the `Ge`, `Gt`, `Le`, and `Lt` counterparts, is that one can write a more “mathematical looking” statement rather than littering the math with oddball function calls. However there are certain (minor) caveats of which to be aware (search for ‘gotcha’, below).

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

LessThan

`class diofant.core.relational.LessThan`

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

$$\text{lhs} \geq \text{rhs}$$

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

Class Name	Symbol
<code>GreaterThan</code>	(\geq)
<code>LessThan</code>	$(<=)$
<code>StrictGreaterThan</code>	$(>)$
<code>StrictLessThan</code>	$(<)$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	$\text{lhs} \geq \text{rhs}$
LessThan(lhs, rhs)	$\text{lhs} \leq \text{rhs}$
StrictGreaterThan(lhs, rhs)	$\text{lhs} > \text{rhs}$
StrictLessThan(lhs, rhs)	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from diofant import GreaterThan, StrictGreaterThan
>>> from diofant import LessThan, StrictLessThan
>>> from diofant import And, Ge, Gt, Le, Lt, Rel, S, Integer
>>> from diofant.abc import x, y, z
>>> from diofant.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the `lhs` argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement $(1 < x)$, Python will first recognize the number 1 as a native number, and then that `x` is *not* a native number. At this point, because a native Python number does not know how to compare itself with a 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%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      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x

```

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

```

>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.

```

Due to an implementation detail or decision of Python [R72] (page 1258), there is no way for Diofant to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

```

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

```

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in Diofant [R73] (page 1259)):

```

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

```

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

References

[R72] (page 1258), [R73] (page 1259)

Examples

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

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

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

Another option is to use the Python inequality operators (\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("e1: %s,    e2: %s" % (e1, e2))
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

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

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

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

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

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

Unequality

class diofant.core.relational.Unequality

An unequal relation between two objects.

Represents that two objects are not equal. If they can be shown to be definitively equal, this will reduce to False; if definitively unequal, this will reduce to True. Otherwise, the relation is maintained as an Unequality object.

See also:

[Equality](#) (page 112)

Notes

This class is not the same as the `!=` operator. The `!=` operator tests for exact structural equality between two expressions; this class compares expressions mathematically.

This class is effectively the inverse of `Equality`. As such, it uses the same algorithms, including any available `evalEq` methods.

Examples

```
>>> from diofant import Ne
>>> from diofant.abc import x, y
>>> Ne(y, x+x**2)
Ne(y, x**2 + x)
```

StrictGreaterThan

class diofant.core.relational.StrictGreaterThan

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:

$\text{lhs} \geq \text{rhs}$

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

Class Name	Symbol
<code>GreaterThan</code>	(\geq)
<code>LessThan</code>	$(<=)$
<code>StrictGreaterThan</code>	$(>)$
<code>StrictLessThan</code>	$(<)$

All classes take two arguments, `lhs` and `rhs`.

Signature Example	Math equivalent
<code>GreaterThan(lhs, rhs)</code>	$\text{lhs} \geq \text{rhs}$
<code>LessThan(lhs, rhs)</code>	$\text{lhs} \leq \text{rhs}$
<code>StrictGreaterThan(lhs, rhs)</code>	$\text{lhs} > \text{rhs}$
<code>StrictLessThan(lhs, rhs)</code>	$\text{lhs} < \text{rhs}$

In addition to the normal `.lhs` and `.rhs` of Relations, `*Than` inequality objects also have the `.lts` and `.gts` properties, which represent the “less than side” and “greater than side” of the operator. Use of `.lts` and `.gts` in an algorithm rather than `.lhs` and `.rhs` as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from diofant import GreaterThan, StrictGreaterThan
>>> from diofant import LessThan, StrictLessThan
>>> from diofant import And, Ge, Gt, Le, Lt, Rel, S, Integer
>>> from diofant.abc import x, y, z
>>> from diofant.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement `(1 < x)`, Python will first recognize the number 1 as a native number, and then that `x` is *not* a native number. At this point, because a native Python number does not know how to compare itself with a 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 %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 %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

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

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [R74] (page 1259), there is no way for Diofant to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of And:

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

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in Diofant [R75] (page 1259)):

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

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

References

[R74] (page 1259), [R75] (page 1259)

Examples

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

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

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

```
x >= 2
x > 2
x <= 2
x < 2
```

Another option is to use the Python inequality operators (\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("e1: %s,    e2: %s" % (e1, e2))
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

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

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

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

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

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

StrictLessThan

`class diofant.core.relational.StrictLessThan`

Class representations of inequalities.

The `*Than` classes represent inequal relationships, where the left-hand side is generally bigger or smaller than the right-hand side. For example, the `GreaterThan` class represents an inequal relationship where the left-hand side is at least as big as the right side, if not bigger. In mathematical notation:

lhs >= rhs

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

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

All classes take two arguments, lhs and rhs.

Signature Example	Math equivalent
GreaterThan(lhs, rhs)	lhs >= rhs
LessThan(lhs, rhs)	lhs <= rhs
StrictGreaterThan(lhs, rhs)	lhs > rhs
StrictLessThan(lhs, rhs)	lhs < rhs

In addition to the normal .lhs and .rhs of Relations, *Than inequality objects also have the .lts and .gts properties, which represent the “less than side” and “greater than side” of the operator. Use of .lts and .gts in an algorithm rather than .lhs and .rhs as an assumption of inequality direction will make more explicit the intent of a certain section of code, and will make it similarly more robust to client code changes:

```
>>> from diofant import GreaterThan, StrictGreaterThan
>>> from diofant import LessThan, StrictLessThan
>>> from diofant import And, Ge, Gt, Le, Lt, Rel, S, Integer
>>> from diofant.abc import x, y, z
>>> from diofant.core.relational import Relational
```

```
>>> e = GreaterThan(x, 1)
>>> e
x >= 1
>>> '%s >= %s is the same as %s <= %s' % (e.gts, e.lts, e.lts, e.gts)
'x >= 1 is the same as 1 <= x'
```

Notes

There are a couple of “gotchas” when using Python’s operators.

The first enters the mix when comparing against a literal number as the lhs argument. Due to the order that Python decides to parse a statement, it may not immediately find two objects comparable. For example, to evaluate the statement (1 < x), Python will first recognize the number 1 as a native number, and then that x is *not* a native number. At this point, because a native Python number does not know how to compare itself with a 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      %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      %s\n"*4 % (e1, e2, e3, e4, e5, e6, e7, e8))
1 > x      1 >= x
1 < x      1 <= x
1 > x      1 >= x
1 < x      1 <= x
```

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

```
>>> e = x < y < z
Traceback (most recent call last):
...
TypeError: symbolic boolean expression has no truth value.
```

Due to an implementation detail or decision of Python [\[R76\]](#) (page 1259), there is no way for Diofant to reliably create that as a chained inequality. To create a chained inequality, the only method currently available is to make use of `And`:

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

Note that this is different than chaining an equality directly via use of parenthesis (this is currently an open bug in Diofant [\[R77\]](#) (page 1260)):

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

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

References

[R76] (page 1259), [R77] (page 1260)

Examples

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

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

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

Another option is to use the Python inequality operators (\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("e1: %s,    e2: %s" % (e1, e2))
e1: x >= 2,    e2: x >= 2
>>> e1 == e2
True
```

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

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

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

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

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

3.1.18 multidimensional

vectorize

class `diofant.core.multidimensional.vectorize(*mdargs)`

Generalizes a function taking scalars to accept multidimensional arguments.

Examples

```
>>> from diofant import diff, sin, symbols, Function
>>> from diofant.core.multidimensional import vectorize
```

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

```
>>> @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)]]
```

3.1.19 function

Lambda

class `diofant.core.function.Lambda`

`Lambda(x, expr)` represents a lambda function similar to Python's `'lambda x: expr'`. A function of several variables is written as `Lambda((x, y, ...), expr)`.

A simple example:

```
>>> from diofant import Lambda
>>> from diofant.abc import x
>>> f = Lambda(x, x**2)
```

```
>>> f(4)
16
```

For multivariate functions, use:

```
>>> from diofant.abc import y, z, t
>>> f2 = Lambda((x, y, z, t), x + y**z + t**z)
>>> f2(1, 2, 3, 4)
73
```

A handy shortcut for lots of arguments:

```
>>> p = x, y, z
>>> f = Lambda(p, x + y*z)
>>> f(*p)
x + y*z
```

expr

The return value of the function

free_symbols

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

See also:

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

is_identity

Return True if this Lambda is an identity function.

variables

The variables used in the internal representation of the function

WildFunction

class `diofant.core.function.WildFunction`(*name*, ***assumptions*)

A WildFunction function matches any function (with its arguments).

Examples

```
>>> from diofant import WildFunction, Function, cos
>>> from diofant.abc import x, y
>>> F = WildFunction('F')
>>> f = Function('f')
>>> F.nargs
Naturals0()
>>> x.match(F)
>>> F.match(F)
{F_: F_}
>>> f(x).match(F)
{F_: f(x)}
>>> cos(x).match(F)
{F_: cos(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
```

To match functions with a given number of arguments, set `nargs` to the desired value at instantiation:

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

To match functions with a range of arguments, set `nargs` to a tuple containing the desired number of arguments, e.g. if `nargs = (1,2)` then functions with 1 or 2 arguments will be matched.

```
>>> F = WildFunction('F', nargs=(1, 2))
>>> F.nargs
{1, 2}
>>> f(x).match(F)
{F_: f(x)}
>>> f(x, y).match(F)
{F_: f(x, y)}
>>> f(x, y, 1).match(F)
```

matches(*expr*, *repl_dict*={})
Helper method for `match()`

See also:

[diofant.core.basic.Basic.matches](#) (page 51)

Derivative

class diofant.core.function.Derivative

Carries out differentiation of the given expression with respect to symbols.

`expr` must define `._eval_derivative(symbol)` method that returns the differentiation result. This function only needs to consider the non-trivial case where `expr` contains `symbol` and it should call the `diff()` method internally (not `._eval_derivative`); `Derivative` should be the only one to call `._eval_derivative`.

Simplification of high-order derivatives:

Because there can be a significant amount of simplification that can be done when multiple differentiations are performed, results will be automatically simplified in a fairly conservative fashion unless the keyword `simplify` is set to `False`.

```
>>> from diofant import sqrt, diff
>>> from diofant.abc import x
>>> e = sqrt((x + 1)**2 + x)
>>> diff(e, x, 5, simplify=False).count_ops()
136
>>> diff(e, x, 5).count_ops()
30
```

Ordering of variables:

If `evaluate` is set to `True` and the expression can not be evaluated, the list of differentiation symbols will be sorted, that is, the expression is assumed to have continuous derivatives up to the order asked. This sorting assumes that derivatives wrt `Symbols`

commute, derivatives wrt non-Symbols commute, but Symbol and non-Symbol derivatives don't commute with each other.

Derivative wrt non-Symbols:

This class also allows derivatives wrt non-Symbols that have `_diff_wrt` set to True, such as Function and Derivative. When a derivative wrt a non-Symbol is attempted, the non-Symbol is temporarily converted to a Symbol while the differentiation is performed.

Note that this may seem strange, that Derivative allows things like `f(g(x)).diff(g(x))`, or even `f(cos(x)).diff(cos(x))`. The motivation for allowing this syntax is to make it easier to work with variational calculus (i.e., the Euler-Lagrange method). The best way to understand this is that the action of derivative with respect to a non-Symbol is defined by the above description: the object is substituted for a Symbol and the derivative is taken with respect to that. This action is only allowed for objects for which this can be done unambiguously, for example Function and Derivative objects. Note that this leads to what may appear to be mathematically inconsistent results. For example:

```
>>> from diofant import cos, sin, sqrt
>>> from diofant.abc import x
>>> (2*cos(x)).diff(cos(x))
2
>>> (2*sqrt(1 - sin(x)**2)).diff(cos(x))
0
```

This appears wrong because in fact `2*cos(x)` and `2*sqrt(1 - sin(x)**2)` are identically equal. However this is the wrong way to think of this. Think of it instead as if we have something like this:

```
>>> from diofant.abc import c, s
>>> def F(u):
...     return 2*u
...
>>> def G(u):
...     return 2*sqrt(1 - u**2)
...
>>> F(cos(x))
2*cos(x)
>>> G(sin(x))
2*sqrt(-sin(x)**2 + 1)
>>> F(c).diff(c)
2
>>> F(c).diff(c)
2
>>> G(s).diff(c)
0
>>> G(sin(x)).diff(cos(x))
0
```

Here, the Symbols `c` and `s` act just like the functions `cos(x)` and `sin(x)`, respectively. Think of `2*cos(x)` as `f(c).subs(c, cos(x))` (or `f(c)` at `c = cos(x)`) and `2*sqrt(1 - sin(x)**2)` as `g(s).subs(s, sin(x))` (or `g(s)` at `s = sin(x)`), where `f(u) == 2*u` and `g(u) == 2*sqrt(1 - u**2)`. Here, we define the function first and evaluate it at the function, but we can actually unambiguously do this in reverse in 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 $(x*y*z).diff(x*y) == 0$).

```
>>> from diofant.abc import x, y, z
>>> (x*y*z).diff(x*y)
Traceback (most recent call last):
...
ValueError: Can't differentiate wrt the variable: x*y, 1
```

Note that this definition also fits in nicely with the definition of the chain rule. Note how the chain rule in Diofant is defined using unevaluated Subs objects:

```
>>> from diofant import symbols, Function
>>> f, g = symbols('f g', cls=Function)
>>> f(2*g(x)).diff(x)
2*Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                          (_xi_1,), (2*g(x),))
>>> f(g(x)).diff(x)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                          (_xi_1,), (g(x),))
```

Finally, note that, to be consistent with variational calculus, and to ensure that the definition of substituting a Function for a Symbol in an expression is well-defined, derivatives of functions are assumed to not be related to the function. In other words, we have:

```
>>> from diofant import diff
>>> diff(f(x), x).diff(f(x))
0
```

The same is true for derivatives of different orders:

```
>>> diff(f(x), x, 2).diff(diff(f(x), x, 1))
0
>>> diff(f(x), x, 1).diff(diff(f(x), x, 2))
0
```

Note, any class can allow derivatives to be taken with respect to itself. See the docstring of `Expr._diff_wrt`.

Examples

Some basic examples:

```
>>> from diofant import Derivative, Symbol, Function
>>> f = Function('f')
>>> g = Function('g')
>>> x = Symbol('x')
>>> y = Symbol('y')
```

```
>>> Derivative(x**2, x, evaluate=True)
2*x
>>> Derivative(Derivative(f(x,y), x), y)
Derivative(f(x, y), x, y)
>>> Derivative(f(x), x, 3)
Derivative(f(x), x, x, x)
>>> Derivative(f(x, y), y, x, evaluate=True)
Derivative(f(x, y), x, y)
```

Now some derivatives wrt functions:

```
>>> Derivative(f(x)**2, f(x), evaluate=True)
2*f(x)
>>> Derivative(f(g(x)), x, evaluate=True)
Derivative(g(x), x)*Subs(Derivative(f(_xi_1), _xi_1),
                        (_xi_1,), (g(x),))
```

doit(hints)**

Evaluate objects that are not evaluated by default.

See also:

[diofant.core.basic.Basic.doit](#) (page 45)

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.

expr

Return expression

free_symbols

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

See also:

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

variables

Return tuple of symbols, wrt derivative is taken.

diff

`diofant.core.function.diff(f, *symbols, **kwargs)`

Differentiate f with respect to symbols.

This is just a wrapper to unify `.diff()` and the `Derivative` class; its interface is similar to that of `integrate()`. You can use the same shortcuts for multiple variables as with `Derivative`. For example, `diff(f(x), x, x, x)` and `diff(f(x), x, 3)` both return the third derivative of `f(x)`.

You can pass `evaluate=False` to get an unevaluated `Derivative` class. Note that if there are 0 symbols (such as `diff(f(x), x, 0)`), then the result will be the function (the zeroth derivative), even if `evaluate=False`.

See also:

[Derivative](#) (page 129)

[diofant.geometry.util.idiff](#) (page 435) computes the derivative implicitly

References

[R78] (page 1260)

Examples

```
>>> from diofant import sin, cos, Function, diff
>>> from diofant.abc import x, y
>>> f = Function('f')
```

```
>>> diff(sin(x), x)
cos(x)
>>> diff(f(x), x, x, x)
Derivative(f(x), x, x, x)
>>> diff(f(x), x, 3)
Derivative(f(x), x, x, x)
>>> diff(sin(x)*cos(y), x, 2, y, 2)
sin(x)*cos(y)
```

```
>>> type(diff(sin(x), x))
cos
>>> type(diff(sin(x), x, evaluate=False))
<class '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.

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.

nargs

Return a set of the allowed number of arguments for the function.

Examples

```
>>> from diofant.core.function import Function
>>> from diofant.abc import x, y
>>> f = Function('f')
```

If the function can take any number of arguments, the set of whole numbers is returned:

```
>>> Function('f').nargs
Naturals0()
```

If the function was initialized to accept one or more arguments, a corresponding set will be returned:

```
>>> Function('f', nargs=1).nargs
{1}
>>> Function('f', nargs=(2, 1)).nargs
{1, 2}
```

The undefined function, after application, also has the nargs attribute; the actual number of arguments is always available by checking the args attribute:

```
>>> f = Function('f')
>>> f(1).nargs
Naturals0()
>>> len(f(1).args)
1
```

Function

class diofant.core.function.Function

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

Examples

First example shows how to use Function as a constructor for undefined function classes:

```
>>> from diofant import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

In the following example Function is used as a base class for `my_func` that represents a mathematical function *my_func*. Suppose that it is well known, that *my_func*(0) is 1 and *my_func* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *my_func*(x) is real exactly when x is real. Here is an implementation that honours those requirements:

```

>>> from diofant import Function, S, oo, I, sin, Symbol
>>> class my_func(Function):
...
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...
...     def _eval_is_real(self):
...         return self.args[0].is_real
...
>>> x = Symbol('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_real
False

```

In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```

>>> class my_func(Function):
...     nargs = (1, 2)
...
>>>

```

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

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:

```

>>> from diofant import Function, cos
>>> from diofant.abc import x
>>> f = Function('f')
>>> f(2) + f(x)
f(2) + f(x)

```

If you want to see which functions appear in an expression you can use the `atoms` method:

```
>>> e = (f(x) + cos(x) + 2)
>>> e.atoms(Function) == {f(x), cos(x)}
True
```

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)}
True
```

Subs

`class diofant.core.function.Subs`

Represents unevaluated substitutions of an expression.

`Subs(expr, x, x0)` receives 3 arguments: an expression, a variable or list of distinct variables and a point or list of evaluation points corresponding to those variables.

`Subs` objects are generally useful to represent unevaluated derivatives calculated at a point.

The variables may be expressions, but they are subjected to the limitations of `subs()`, so it is usually a good practice to use only symbols for variables, since in that case there can be no ambiguity.

There's no automatic expansion - use the method `.doit()` to effect all possible substitutions of the object and also of objects inside the expression.

When evaluating derivatives at a point that is not a symbol, a `Subs` object is returned. One is also able to calculate derivatives of `Subs` objects - in this case the expression is always expanded (for the unevaluated form, use `Derivative()`).

A simple example:

```
>>> from diofant import Subs, Function, sin
>>> from diofant.abc import x, y, z
>>> f = Function('f')
>>> e = Subs(f(x).diff(x), x, y)
>>> e.subs(y, 0)
Subs(Derivative(f(x), x), (x,), (0,))
>>> e.subs(f, sin).doit()
cos(y)
```

An example with several variables:

```
>>> Subs(f(x)*sin(y) + z, (x, y), (0, 1))
Subs(z + f(x)*sin(y), (x, y), (0, 1))
>>> _.doit()
z + f(0)*sin(1)
```

`doit(**hints)`

Evaluate objects that are not evaluated by default.

See also:

[diofant.core.basic.Basic.doit](#) (page 45)

evalf(*prec=None, **options*)

Evaluate the given formula to an accuracy of *prec* digits.

See also:

[diofant.core.evalf.EvalfMixin.evalf](#) (page 147)

expr

The expression on which the substitution operates

free_symbols

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

See also:

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

n(*prec=None, **options*)

point

The values for which the variables are to be substituted

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 (“):

```
>>> from diofant import cos, exp, sin
>>> from diofant.abc import x, y, z
>>> (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:

```
>>> from diofant import log, symbols
>>> log(x**2*y).expand(log=True)
log(x**2*y)
>>> log(x**2*y).expand(log=True, force=True)
2*log(x) + log(y)
>>> x, y = symbols('x,y', positive=True)
>>> log(x**2*y).expand(log=True)
2*log(x) + log(y)
```

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

Expand other functions.

```
>>> from diofant import gamma
>>> 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. See [R79] (page 1260) for more information.

force : boolean, optional

If the `force` hint is used, assumptions about variables will be ignored in making the expansion.

See also:

[expand_log](#) (page 143), [expand_mul](#) (page 143), [expand_multinomial](#) (page 145), [expand_complex](#) (page 144), [expand_trig](#) (page 144), [expand_power_base](#) (page 145), [expand_power_exp](#) (page 145), [expand_func](#) (page 144), [diofant.simplify.hyperexpand.hyperexpand](#) (page 943)

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 + y)*x + E**(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:

```
>>> from diofant import expand, expand_mul, expand_power_base
>>> x, y, z = symbols('x,y,z', positive=True)

>>> expand(log(x*(y + z)))
log(x) + log(y + z)
```

Here, we see that `log` was applied before `mul`. To get the `mul` expanded form, either of the following will work:

```
>>> expand_mul(log(x*(y + z)))
log(x*y + x*z)
>>> expand(log(x*(y + z)), log=False)
log(x*y + x*z)
```

A similar thing can happen with the `power_base` hint:

```
>>> expand((x*(y + z))**x)
(x*y + x*z)**x
```

To get the `power_base` expanded form, either of the following will work:

```
>>> expand((x*(y + z))**x, mul=False)
x**x*(y + z)**x
>>> expand_power_base((x*(y + z))**x)
x**x*(y + z)**x

>>> expand((x + y)*y/x)
y + y**2/x
```

The parts of a rational expression can be targeted:

```
>>> expand((x + y)*y/x/(x + 1), frac=True)
(x*y + y**2)/(x**2 + x)
>>> expand((x + y)*y/x/(x + 1), numer=True)
(x*y + y**2)/(x*(x + 1))
>>> expand((x + y)*y/x/(x + 1), denom=True)
y*(x + y)/(x**2 + x)
```

- The modulus meta-hint can be used to reduce the coefficients of an expression post-expansion:

```
>>> expand((3*x + 1)**2)
9*x**2 + 6*x + 1
>>> expand((3*x + 1)**2, modulus=5)
4*x**2 + x + 1
```

- Either `expand()` the function or `.expand()` the method can be used. Both are equivalent:

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


- 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.

References

[R79] (page 1260)

Examples

```
>>> from diofant import Expr, sympify
>>> class MyClass(Expr):
...     def __new__(cls, *args):
...         args = sympify(args)
...         return Expr.__new__(cls, *args)
...
...     def _eval_expand_double(self, **hints):
...         '''
...         Doubles the args of MyClass.
...
...         If there more than four args, doubling is not performed,
...         unless force=True is also used (False by default).
...         '''
```

```
...         '''
...         force = hints.pop('force', False)
...         if not force and len(self.args) > 4:
...             return self
...         return self.func(*(self.args + self.args))
...
>>> a = MyClass(1, 2, MyClass(3, 4))
>>> a
MyClass(1, 2, MyClass(3, 4))
>>> a.expand(double=True)
MyClass(1, 2, MyClass(3, 4, 3, 4), 1, 2, MyClass(3, 4, 3, 4))
>>> a.expand(double=True, deep=False)
MyClass(1, 2, MyClass(3, 4), 1, 2, MyClass(3, 4))
```

```
>>> b = MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True)
MyClass(1, 2, 3, 4, 5)
>>> b.expand(double=True, force=True)
MyClass(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

PoleError

`class diofant.core.function.PoleError`

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

```
>>> from diofant.abc import a, b, x, y
>>> from diofant import sin, count_ops
```

Although there isn't a SUB object, minus signs are interpreted as either negations or subtractions:

```
>>> (x - y).count_ops(visual=True)
SUB
>>> (-x).count_ops(visual=True)
NEG
```

Here, there are two Adds and a Pow:

```
>>> (1 + a + b**2).count_ops(visual=True)
2*ADD + POW
```

In the following, an Add, Mul, Pow and two functions:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=True)
ADD + MUL + POW + 2*SIN
```

for a total of 5:

```
>>> (sin(x)*x + sin(x)**2).count_ops(visual=False)
5
```

Note that “what you type” is not always what you get. The expression $1/x/y$ is translated by 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

```
>>> from diofant import symbols, expand_mul, exp, log
>>> 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

```
>>> from diofant import symbols, expand_log, exp, log
>>> 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

```
>>> from diofant import expand_func, gamma
>>> from diofant.abc import x
>>> 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

```
>>> from diofant import expand_trig, sin
>>> from diofant.abc import x, y
>>> 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.

See also:

[`diofant.core.expr.Expr.as_real_imag`](#) (page 67)

Examples

```
>>> from diofant import expand_complex, exp, sqrt, I
>>> from diofant.abc import z
>>> 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
```

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

```
>>> from diofant import symbols, expand_multinomial, exp
>>> 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.

See the `expand` docstring for more information.

Examples

```
>>> from diofant import expand_power_exp
>>> from diofant.abc import x, y
>>> expand_power_exp(x**(y + 2))
x**2*x**y
```

expand_power_base

`diofant.core.function.expand_power_base(expr, deep=True, force=False)`

Wrapper around `expand` that only uses the `power_base` hint.

See the `expand` docstring for more information.

A wrapper to `expand(power_base=True)` which separates a power with a base that is a `Mul` into a product of powers, without performing any other expansions, provided that assumptions about the power's base and exponent allow.

`deep=False` (default is `True`) will only apply to the top-level expression.

`force=True` (default is `False`) will cause the expansion to ignore assumptions about the base and exponent. When `False`, the expansion will only happen if the base is non-negative or the exponent is an integer.

```
>>> from diofant.abc import x, y, z
>>> from diofant import expand_power_base, sin, cos, exp
```

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

```
>>> (2*x)**y
(2*x)**y
>>> expand_power_base(_)
2**y*x**y
```

```
>>> expand_power_base((x*y)**z)
(x*y)**z
>>> expand_power_base((x*y)**z, force=True)
x**z*y**z
>>> expand_power_base(sin((x*y)**z), deep=False)
sin((x*y)**z)
>>> expand_power_base(sin((x*y)**z), force=True)
sin(x**z*y**z)
```

```
>>> expand_power_base((2*sin(x))**y + (2*cos(x))**y)
2**y*sin(x)**y + 2**y*cos(x)**y
```

```
>>> expand_power_base((2*exp(y))**x)
2**x*(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
```

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

```

>>> from diofant.core.function import nfloat
>>> from diofant.abc import x, y
>>> from diofant import cos, pi, sqrt
>>> nfloat(x**4 + x/2 + cos(pi/3) + 1 + sqrt(y))
x**4 + 0.5*x + sqrt(y) + 1.5
>>> nfloat(x**4 + sqrt(y), exponent=True)
x**4.0 + y**0.5

```

3.1.20 evalf

class diofant.core.evalf.EvalfMixin

Mixin class adding evalf capability.

evalf(*n=15, subs=None, maxn=100, chop=False, strict=False, quad=None*)

Evaluate the given formula to an accuracy of *n* 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=100)

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=False)

quad=<str> Choose algorithm for numerical quadrature. By default, tanh-sinh quadrature is used. For oscillatory integrals on an infinite interval, try `quad='osc'`.

n(*n=15, subs=None, maxn=100, chop=False, strict=False, quad=None*)

Evaluate the given formula to an accuracy of *n* 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=100)

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=False)

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

N

`diofant.core.evalf.N(x, n=15, **options)`

Calls `x.evalf(n, **options)`.

Both `.n()` and `N()` are equivalent to `.evalf()`; use the one that you like better.

See also:

`diofant.core.evalf.EvalfMixin.evalf` (page 147)

Examples

```
>>> from diofant import Sum, oo, N
>>> from diofant.abc import k
>>> Sum(1/k**k, (k, 1, oo))
Sum(k**(-k), (k, 1, oo))
>>> N(_, 4)
1.291
```

3.1.21 containers

Tuple

class `diofant.core.containers.Tuple`

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.

```
>>> from diofant import symbols
>>> from diofant.core.containers import Tuple
>>> a, b, c, d = symbols('a b c d')
>>> Tuple(a, b, c)[1:]
(b, c)
>>> Tuple(a, b, c).subs(a, d)
(d, b, c)
```

index(`value`[, `start`[, `stop`]]) → integer - return first index of value.

Raises `ValueError` if the value is not present.

tuple_count(`value`)

`T.count(value)` -> integer - return number of occurrences of value

Dict

class `diofant.core.containers.Dict`

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.


```
>>> from diofant.core.containers import Dict
```

```
>>> D = Dict({1: 'one', 2: 'two'})
>>> for key in D:
...     if key == 1:
...         print('%s %s' % (key, D[key]))
1 one
```

The args are sympified so the 1 and 2 are Integers and the values are Symbols. Queries automatically sympify args so the following work:

```
>>> 1 in D
True
>>> D.has('one') # searches keys and values
True
>>> 'one' in D # not in the keys
False
>>> D[1]
one
```

args

Returns a tuple of arguments of 'self'.

See also:

[diofant.core.basic.Basic.args](#) (page 42)

get(k , d) → $D[k]$ if k in D , else d . d defaults to `None`.

items() → list of D 's (key, value) pairs, as 2-tuples

keys() → list of D 's keys

values() → list of D 's values

3.1.22 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.

class diofant.core.compatibility.NotIterable

Use this as mixin when creating a class which is not supposed to return true when `iterable()` is called on its instances. I.e. avoid infinite loop when calling e.g. `list()` on the instance

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

```
>>> from diofant.core.compatibility import as_int
>>> from diofant import sqrt
>>> 3.0
3.0
```

```
>>> as_int(3.0) # convert to int and test for equality
3
>>> int(sqrt(10))
3
>>> as_int(sqrt(10))
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

`diofant.core.compatibility.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').

See also:

[ordered](#) (page 153), [diofant.core.expr.Expr.as_ordered_factors](#) (page 66), [diofant.core.expr.Expr.as_ordered_terms](#) (page 66)

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]
```

Examples

```
>>> from diofant import S, I, default_sort_key, sin, cos, sqrt
>>> from diofant.core.function import UndefinedFunction
>>> from diofant.abc import x
```

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(S.One)
((1, 0, 'Number'), (0, ()), ()), 1)
>>> default_sort_key(2)
((1, 0, 'Number'), (0, ()), ()), 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
```

`diofant.core.compatibility.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:

[iterable](#) (page 152)

Examples

```
>>> from diofant.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

`diofant.core.compatibility.iterable(i, exclude=(<class 'str'>, <class 'dict'>, <class 'diofant.core.compatibility.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 151)

Examples

```
>>> from diofant.utilities.iterables import iterable
>>> from diofant import Tuple
>>> things = [[1], (1,), {1}, Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'diofant.core.containers.Tuple'>
True <... 'generator'>
```

```
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

`diofant.core.compatibility.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 `default_sort_key` which (if the `sort_key` 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.

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.

Examples

```
>>> from diofant.core.compatibility import ordered
>>> from diofant import count_ops
>>> from diofant.abc import x, y
```

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
```

iterable

`diofant.core.compatibility.iterable(i, exclude=(<class 'str'>, <class 'dict'>, <class 'diofant.core.compatibility.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 151)

Examples

```
>>> from diofant.utilities.iterables import iterable
>>> from diofant import Tuple
>>> things = [[1], (1,), {1}, Tuple(1), (j for j in [1, 2]), {1:2}, '1', 1]
>>> for i in things:
...     print('%s %s' % (iterable(i), type(i)))
True <... 'list'>
True <... 'tuple'>
True <... 'set'>
True <class 'diofant.core.containers.Tuple'>
True <... 'generator'>
False <... 'dict'>
False <... 'str'>
False <... 'int'>
```

```
>>> iterable({}, exclude=None)
True
>>> iterable({}, exclude=str)
True
>>> iterable("no", exclude=str)
False
```

is_sequence

`diofant.core.compatibility.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:

[iterable](#) (page 152)

Examples

```
>>> from diofant.utilities.iterables import is_sequence
>>> from types import GeneratorType
>>> is_sequence([])
True
>>> is_sequence(set())
False
>>> is_sequence('abc')
False
>>> is_sequence('abc', include=str)
True
>>> generator = (c for c in 'abc')
>>> is_sequence(generator)
False
>>> is_sequence(generator, include=(str, GeneratorType))
True
```

as_int

`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

```
>>> from diofant.core.compatibility import as_int
>>> from diofant import sqrt
>>> 3.0
3.0
>>> as_int(3.0) # convert to int and test for equality
3
>>> int(sqrt(10))
3
>>> as_int(sqrt(10))
Traceback (most recent call last):
...
ValueError: ... is not an integer
```

3.1.23 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.

See also:

[factor_terms](#) (page 157), [diofant.polys.polytools.terms_gcd](#) (page 680)

Examples

```
>>> from diofant.core import gcd_terms
>>> from diofant.abc import x, y
```

```
>>> gcd_terms((x + 1)**2*y + (x + 1)*y**2)
y*(x + 1)*(x + y + 1)
>>> gcd_terms(x/2 + 1)
(x + 2)/2
>>> gcd_terms(x/2 + 1, clear=False)
x/2 + 1
>>> gcd_terms(x/2 + y/2, clear=False)
(x + y)/2
>>> gcd_terms(x/2 + 1/x)
(x**2 + 2)/(2*x)
>>> gcd_terms(x/2 + 1/x, fraction=False)
(x + 2/x)/2
>>> gcd_terms(x/2 + 1/x, fraction=False, clear=False)
x/2 + 1/x
```



```
>>> gcd_terms(x/2/y + 1/x/y)
(x**2 + 2)/(2*x*y)
>>> gcd_terms(x/2/y + 1/x/y, fraction=False, clear=False)
(x + 2/x)/(2*y)
```

The `clear` flag was ignored in this case because the returned expression was a rational expression, not a simple sum.

factor_terms

`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.

See also:

[gcd_terms](#) (page 156), [diofant.polys.polytools.terms_gcd](#) (page 680)

Examples

```
>>> from diofant import factor_terms, Symbol
>>> from diofant.abc import x, y
>>> factor_terms(x + x*(2 + 4*y)**3)
x*(8*(2*y + 1)**3 + 1)
>>> A = Symbol('A', commutative=False)
>>> factor_terms(x*A + x*A + x*y*A)
x*(y*A + 2*A)
```

When `clear` is False, a rational will only be factored out of an Add expression if all terms of the Add have coefficients that are fractions:

```
>>> factor_terms(x/2 + 1, clear=False)
x/2 + 1
>>> factor_terms(x/2 + 1, clear=True)
(x + 2)/2
```

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

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

If a -1 is all that can be factored out, to *not* factor it out, the flag sign must be False:

```
>>> factor_terms(-x - y)
-(x + y)
>>> factor_terms(-x - y, sign=False)
-x - y
>>> factor_terms(-2*x - 2*y, sign=False)
-2*(x + y)
```

3.2 Combinatorics

3.2.1 Partitions

class `diofant.combinatorics.partitions.Partition`

This class represents an abstract partition.

A partition is a set of disjoint sets whose union equals a given set.

See also:

[diofant.utilities.iterables.partitions](#) (page 1155), [diofant.utilities.iterables.multiset_partitions](#) (page 1153)

RGS

Returns the “restricted growth string” of the partition.

The RGS is returned as a list of indices, L, where L[i] indicates the block in which element i appears. For example, in a partition of 3 elements (a, b, c) into 2 blocks ([c], [a, b]) the RGS is [1, 1, 0]: “a” is in block 1, “b” is in block 1 and “c” is in block 0.

Examples

```
>>> from diofant.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.members
(1, 2, 3, 4, 5)
>>> a.RGS
(0, 0, 1, 2, 2)
>>> a + 1
{{3}, {4}, {5}, {1, 2}}
>>> a.RGS
(0, 0, 1, 2, 3)
```

classmethod `from_rgs(rgs, elements)`

Creates a set partition from a restricted growth string.

The indices given in rgs are assumed to be the index of the element as given in *elements as provided* (the elements are not sorted by this routine). Block numbering starts from 0. If any block was not referenced in rgs an error will be raised.

Examples

```
>>> from diofant.combinatorics.partitions import Partition
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('abcde'))
{{c}, {a, d}, {b, e}}
>>> Partition.from_rgs([0, 1, 2, 0, 1], list('cbead'))
{{e}, {a, c}, {b, d}}
>>> a = Partition([1, 4], [2], [3, 5])
>>> Partition.from_rgs(a.RGS, a.members)
{{2}, {1, 4}, {3, 5}}
```

partition

Return partition as a sorted list of lists.

Examples

```
>>> from diofant.combinatorics.partitions import Partition
>>> Partition([1], [2, 3]).partition
[[1], [2, 3]]
```

rank

Gets the rank of a partition.

Examples

```
>>> from diofant.combinatorics.partitions import Partition
>>> a = Partition([1, 2], [3], [4, 5])
>>> a.rank
13
```

sort_key(*order=None*)

Return a canonical key that can be used for sorting.

Ordering is based on the size and sorted elements of the partition and ties are broken with the rank.

Examples

```
>>> from diofant.utilities.iterables import default_sort_key
>>> from diofant.combinatorics.partitions import Partition
>>> from diofant.abc import x
>>> a = Partition([1, 2])
>>> b = Partition([3, 4])
>>> c = Partition([1, x])
>>> d = Partition(list(range(4)))
>>> l = [d, b, a + 1, a, c]
>>> l.sort(key=default_sort_key); l
[[{1, 2}], {{1}, {2}}, {{1, x}}, {{3, 4}}, {{0, 1, 2, 3}}]
```

class diofant.combinatorics.partitions.IntegerPartition

This class represents an integer partition.

In number theory and combinatorics, a partition of a positive integer, n , also called an integer partition, is a way of writing n as a list of positive integers that sum to n . Two partitions that differ only in the order of summands are considered to be the same partition; if order matters then the partitions are referred to as compositions. For example, 4 has five partitions: [4], [3, 1], [2, 2], [2, 1, 1], and [1, 1, 1, 1]; the compositions [1, 2, 1] and [1, 1, 2] are the same as partition [2, 1, 1].

See also:

[*diofant.utilities.iterables.partitions*](#) (page 1155), [*diofant.utilities.iterables.multiset_partitions*](#) (page 1153)

References

[R8] (page 1260)

as_dict()

Return the partition as a dictionary whose keys are the partition integers and the values are the multiplicity of that integer.

Examples

```
>>> from diofant.combinatorics.partitions import IntegerPartition
>>> IntegerPartition([1]*3 + [2] + [3]*4).as_dict()
{1: 3, 2: 1, 3: 4}
```

as_ferrers(char='#')

Prints the ferrer diagram of a partition.

Examples

```
>>> from diofant.combinatorics.partitions import IntegerPartition
>>> print(IntegerPartition([1, 1, 5]).as_ferrers())
#####
#
#
```

conjugate

Computes the conjugate partition of itself.

Examples

```
>>> from diofant.combinatorics.partitions import IntegerPartition
>>> a = IntegerPartition([6, 3, 3, 2, 1])
>>> a.conjugate
[5, 4, 3, 1, 1, 1]
```

next_lex()

Return the next partition of the integer, n , in lexical order, wrapping around to $[n]$ if the partition is $[1, \dots, 1]$.

Examples

```
>>> from diofant.combinatorics.partitions import IntegerPartition
>>> p = IntegerPartition([3, 1])
>>> print(p.next_lex())
[4]
>>> p.partition < p.next_lex().partition
True
```

prev_lex()

Return the previous partition of the integer, n , in lexical order, wrapping around to $[1, \dots, 1]$ if the partition is $[n]$.

Examples

```
>>> from diofant.combinatorics.partitions import IntegerPartition
>>> 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

```
>>> from diofant.combinatorics.partitions import random_integer_partition
```

For the following, a seed is given so a known value can be shown; in practice, the seed would not be given.

```
>>> random_integer_partition(100, seed=[1, 1, 12, 1, 2, 1, 85, 1])
[85, 12, 2, 1]
>>> random_integer_partition(10, seed=[1, 2, 3, 1, 5, 1])
[5, 3, 1, 1]
>>> random_integer_partition(1)
[1]
```

`diofant.combinatorics.partitions.RGS_generalized(m)`
Computes the $m + 1$ generalized unrestricted growth strings and returns them as rows in matrix.

Examples

```
>>> from diofant.combinatorics.partitions import RGS_generalized
>>> RGS_generalized(6)
Matrix([
[ 1,  1,  1,  1,  1,  1,  1],
[ 1,  2,  3,  4,  5,  6,  0],
[ 2,  5, 10, 17, 26,  0,  0],
```

```
[ 5, 15, 37, 77, 0, 0, 0],  
[ 15, 52, 151, 0, 0, 0, 0],  
[ 52, 203, 0, 0, 0, 0, 0],  
[203, 0, 0, 0, 0, 0, 0]])
```

`diofant.combinatorics.partitions.RGS_enum(m)`

`RGS_enum` computes the total number of restricted growth strings possible for a superset of size `m`.

Examples

```
>>> from diofant.combinatorics.partitions import RGS_enum  
>>> from diofant.combinatorics.partitions import Partition  
>>> RGS_enum(4)  
15  
>>> RGS_enum(5)  
52  
>>> RGS_enum(6)  
203
```

We can check that the enumeration is correct by actually generating the partitions. Here, the 15 partitions of 4 items are generated:

```
>>> a = Partition(list(range(4)))  
>>> s = set()  
>>> for i in range(20):  
...     s.add(a)  
...     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

```
>>> from diofant.combinatorics.partitions import RGS_unrank  
>>> 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

```
>>> from diofant.combinatorics.partitions import RGS_rank, RGS_unrank  
>>> RGS_rank([0, 1, 2, 1, 3])  
42  
>>> RGS_rank(RGS_unrank(4, 7))  
4
```

3.2.2 Permutations

class diofant.combinatorics.permutations.Permutation

A permutation, alternatively known as an ‘arrangement number’ or ‘ordering’ is an arrangement of the elements of an ordered list into a one-to-one mapping with itself. The permutation of a given arrangement is given by indicating the positions of the elements after re-arrangement [R13] (page 1260). 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.

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = False
```

See also:

Cycle (page 184)

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:

$$[0 \ 1 \ 2 \ \dots \ n-1] [p(0) \ p(1) \ p(2) \ \dots \ p(n-1)]$$

Since the first line is always $\text{range}(n)$, where n is the size of p , it is sufficient to represent the permutation by the second line, referred to as the “array form” of the permutation. This is entered in brackets as the argument to the Permutation class:

```
>>> p = Permutation([0, 2, 1]); p
Permutation([0, 2, 1])
```

Given i in $\text{range}(p.\text{size})$, the permutation maps i to i^p

```
>>> [i^p for i in range(p.size)]
[0, 2, 1]
```

The composite of two permutations $p*q$ means first apply p , then q , so $i^{(p*q)} = (i^p)^q$ which is $i^p \wedge q$ 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, $(p*q)(i) = q(p(i))$ [R17] (page 1260).

```
>>> 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])
```

References

[R12] (page 1260), [R13] (page 1260), [R14] (page 1260), [R15] (page 1260), [R16] (page 1260), [R17] (page 1260), [R18] (page 1260)

array_form

Return a copy of the attribute `_array_form` Examples =====

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
```

```
>>> p = Permutation([[2, 0], [3, 1]])
>>> p.array_form
[2, 3, 0, 1]
>>> Permutation([[2, 0, 3, 1]]).array_form
[3, 2, 0, 1]
>>> Permutation([2, 0, 3, 1]).array_form
[2, 0, 3, 1]
>>> Permutation([[1, 2], [4, 5]]).array_form
[0, 2, 1, 3, 5, 4]
```

ascents()

Returns the positions of ascents in a permutation, ie, the location where $p[i] < p[i+1]$

See also:

descents (page 170), *inversions* (page 174), *min* (page 178), *max* (page 178)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.ascents()
[1, 2]
```

atoms()

Returns all the elements of a permutation

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation([0, 1, 2, 3, 4, 5]).atoms()
{0, 1, 2, 3, 4, 5}
>>> Permutation([[0, 1], [2, 3], [4, 5]]).atoms()
{0, 1, 2, 3, 4, 5}
```

cardinality

Returns the number of all possible permutations.

See also:

length (page 177), *order* (page 179), *rank* (page 180), *size* (page 183)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.cardinality
24
```

commutator(x)

Return the commutator of self and x: $\sim x \sim \text{self} * x * \text{self}$

If f and g are part of a group, G, then the commutator of f and g is the group identity iff f and g commute, i.e. $fg == gf$.

References

<http://en.wikipedia.org/wiki/Commutator>

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 2, 3, 1])
>>> x = Permutation([2, 0, 3, 1])
>>> c = p.commutator(x); c
Permutation([2, 1, 3, 0])
>>> c == ~x*~p*x*p
True
```

```
>>> I = Permutation(3)
>>> p = [I + i for i in range(6)]
>>> for i in range(len(p)):
...     for j in range(len(p)):
...         c = p[i].commutator(p[j])
...         if p[i]*p[j] == p[j]*p[i]:
...             assert c == I
...         else:
...             assert c != I
...
...
...
...
...
```

`commutes_with(other)`

Checks if the elements are commuting.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> a = Permutation([1, 4, 3, 0, 2, 5])
>>> b = Permutation([0, 1, 2, 3, 4, 5])
>>> a.commutes_with(b)
True
>>> b = Permutation([2, 3, 5, 4, 1, 0])
>>> a.commutes_with(b)
False
```

`cycle_structure`

Return the cycle structure of the permutation as a dictionary indicating the multiplicity of each cycle length.

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> Permutation(3).cycle_structure
{1: 4}
>>> Permutation(0, 4, 3)(1, 2)(5, 6).cycle_structure
{2: 2, 3: 1}
```

cycles

Returns the number of cycles contained in the permutation (including singletons).

See also:

[diofant.functions.combinatorial.numbers.stirling](#) (page 352)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation([0, 1, 2]).cycles
3
>>> Permutation([0, 1, 2]).full_cyclic_form
[[0], [1], [2]]
>>> Permutation(0, 1)(2, 3).cycles
2
```

cyclic_form

This is used to convert to the cyclic notation from the canonical notation. Singletons are omitted.

See also:

[array_form](#) (page 167), [full_cyclic_form](#) (page 171)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([0, 3, 1, 2])
>>> p.cyclic_form
[[1, 3, 2]]
>>> Permutation([1, 0, 2, 4, 3, 5]).cyclic_form
[[0, 1], [3, 4]]
```

descents()

Returns the positions of descents in a permutation, ie, the location where $p[i] > p[i+1]$

See also:

[ascents](#) (page 168), [inversions](#) (page 174), [min](#) (page 178), [max](#) (page 178)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([4, 0, 1, 3, 2])
>>> p.descents()
[0, 3]
```

classmethod from_inversion_vector(*inversion*)

Calculates the permutation from the inversion vector.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> Permutation.from_inversion_vector([3, 2, 1, 0, 0])
Permutation([3, 2, 1, 0, 4, 5])
```

classmethod `from_sequence(i, key=None)`

Return the permutation needed to obtain `i` from the sorted elements of `i`. If custom sorting is desired, a key can be given.

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
```

```
>>> Permutation.from_sequence('SymPy')
Permutation(4)(0, 1, 3)
>>> _(sorted("SymPy"))
['S', 'y', 'm', 'P', 'y']
>>> Permutation.from_sequence('SymPy', key=Lambda x: x.lower())
Permutation(4)(0, 2)(1, 3)
```

full_cyclic_form

Return permutation in cyclic form including singletons.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation([0, 2, 1]).full_cyclic_form
[[0], [1, 2]]
```

get_adjacency_distance(other)

Computes the adjacency distance between two permutations.

This metric counts the number of times a pair i,j of jobs is adjacent in both p and p' . If n_{adj} is this quantity then the adjacency distance is $n - n_{adj} - 1$ [1]

[1] Reeves, Colin R. Landscapes, Operators and Heuristic search, Annals of Operational Research, 86, pp 473-490. (1999)

See also:

[get_precedence_matrix](#) (page 173), [get_precedence_distance](#) (page 172), [get_adjacency_matrix](#) (page 172)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> p.get_adjacency_distance(q)
3
```

```
>>> r = Permutation([0, 2, 1, 4, 3])
>>> p.get_adjacency_distance(r)
4
```

get_adjacency_matrix()

Computes the adjacency matrix of a permutation.

If job i is adjacent to job j in a permutation p then we set $m[i, j] = 1$ where m is the adjacency matrix of p .

See also:

[get_precedence_matrix](#) (page 173), [get_precedence_distance](#) (page 172), [get_adjacency_distance](#) (page 171)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation.josephus(3, 6, 1)
>>> p.get_adjacency_matrix()
Matrix([
[0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1],
[0, 1, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0],
[0, 0, 0, 1, 0, 0]])
>>> q = Permutation([0, 1, 2, 3])
>>> q.get_adjacency_matrix()
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[0, 0, 0, 0]])
```

get_positional_distance(*other*)

Computes the positional distance between two permutations.

See also:

[get_precedence_distance](#) (page 172), [get_adjacency_distance](#) (page 171)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 3, 1, 2, 4])
>>> q = Permutation.josephus(4, 5, 2)
>>> r = Permutation([3, 1, 4, 0, 2])
>>> p.get_positional_distance(q)
12
>>> p.get_positional_distance(r)
12
```

get_precedence_distance(*other*)

Computes the precedence distance between two permutations.

Suppose p and p' represent n jobs. The precedence metric counts the number of times a job j is preceded by job i in both p and p' . This metric is commutative.

See also:

[get_precedence_matrix](#) (page 173), [get_adjacency_matrix](#) (page 172), [get_adjacency_distance](#) (page 171)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([2, 0, 4, 3, 1])
>>> q = Permutation([3, 1, 2, 4, 0])
>>> p.get_precedence_distance(q)
7
>>> q.get_precedence_distance(p)
7
```

get_precedence_matrix()

Gets the precedence matrix. This is used for computing the distance between two permutations.

See also:

[get_precedence_distance](#) (page 172), [get_adjacency_matrix](#) (page 172), [get_adjacency_distance](#) (page 171)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> 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, 0, 1, 0],
[1, 0, 0, 0, 0, 0],
[1, 1, 0, 1, 1, 0]])
```

index()

Returns the index of a permutation.

The index of a permutation is the sum of all subscripts j such that $p[j]$ is greater than $p[j+1]$.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([3, 0, 2, 1, 4])
>>> p.index()
2
```

inversion_vector()

Return the inversion vector of the permutation.

The inversion vector consists of elements whose value indicates the number of elements in the permutation that are lesser than it and lie on its right hand side.

The inversion vector is the same as the Lehmer encoding of a permutation.

See also:

from_inversion_vector (page 170)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([4, 8, 0, 7, 1, 5, 3, 6, 2])
>>> p.inversion_vector()
[4, 7, 0, 5, 0, 2, 1, 1]
>>> p = Permutation([3, 2, 1, 0])
>>> p.inversion_vector()
[3, 2, 1]
```

The inversion vector increases lexicographically with the rank of the permutation, the *i*-th element cycling through 0..i.

```
>>> p = Permutation(2)
>>> Permutation.print_cyclic = False
>>> while p:
...     print('%s %s %s' % (repr(p), p.inversion_vector(), p.rank()))
...     p = p.next_lex()
...
Permutation([0, 1, 2]) [0, 0] 0
Permutation([0, 2, 1]) [0, 1] 1
Permutation([1, 0, 2]) [1, 0] 2
Permutation([1, 2, 0]) [1, 1] 3
Permutation([2, 0, 1]) [2, 0] 4
Permutation([2, 1, 0]) [2, 1] 5
```

inversions()

Computes the number of inversions of a permutation.

An inversion is where $i > j$ but $p[i] < p[j]$.

For small length of *p*, it iterates over all *i* and *j* values and calculates the number of inversions. For large length of *p*, it uses a variation of merge sort to calculate the number of inversions.

See also:

descents (page 170), *ascents* (page 168), *min* (page 178), *max* (page 178)

References

[1] <http://www.cp.eng.chula.ac.th/~piak/teaching/algo/algo2008/count-inv.htm>

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3, 4, 5])
>>> p.inversions()
0
>>> Permutation([3, 2, 1, 0]).inversions()
6
```

is_Empty

Checks to see if the permutation is a set with zero elements

See also:

[is_Singleton](#) (page 175)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation([]).is_Empty
True
>>> Permutation([0]).is_Empty
False
```

is_Identity

Returns True if the Permutation is an identity permutation.

See also:

[order](#) (page 179)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([])
>>> p.is_Identity
True
>>> p = Permutation([[0], [1], [2]])
>>> p.is_Identity
True
>>> p = Permutation([0, 1, 2])
>>> p.is_Identity
True
>>> p = Permutation([0, 2, 1])
>>> p.is_Identity
False
```

is_Singleton

Checks to see if the permutation contains only one number and is thus the only possible permutation of this set of numbers

See also:

[is_Empty](#) (page 175)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation([0]).is_Singleton
True
>>> Permutation([0, 1]).is_Singleton
False
```

`is_even`

Checks if a permutation is even.

See also:

[`is_odd`](#) (page 176)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_even
True
>>> p = Permutation([3, 2, 1, 0])
>>> p.is_even
True
```

`is_odd`

Checks if a permutation is odd.

See also:

[`is_even`](#) (page 176)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.is_odd
False
>>> p = Permutation([3, 2, 0, 1])
>>> p.is_odd
True
```

`classmethod josephus(m, n, s=1)`

Return as a permutation the shuffling of `range(n)` using the Josephus scheme in which every `m`-th item is selected until all have been chosen. The returned permutation has elements listed by the order in which they were selected.

The parameter `s` stops the selection process when there are `s` items remaining and these are selected by continuing the selection, counting by 1 rather than by `m`.

Consider selecting every 3rd item from 6 until only 2 remain:

```
choices   chosen
=====   =====
012345
01 345   2
```

```

01 34    25
01  4   253
0   4   2531
0      25314
      253140

```

References

- 1.http://en.wikipedia.org/wiki/Flavius_Josephus
- 2.http://en.wikipedia.org/wiki/Josephus_problem
- 3.<http://www.wou.edu/~burtonl/josephus.html>

Examples

```

>>> from diofant.combinatorics import Permutation
>>> Permutation.josephus(3, 6, 2).array_form
[2, 5, 3, 1, 4, 0]

```

length()

Returns the number of integers moved by a permutation.

See also:

min (page 178), *max* (page 178), *support* (page 183), *cardinality* (page 168), *order* (page 179), *rank* (page 180), *size* (page 183)

Examples

```

>>> from diofant.combinatorics import Permutation
>>> Permutation([0, 3, 2, 1]).length()
2
>>> Permutation([[0, 1], [2, 3]]).length()
4

```

list(size=None)

Return the permutation as an explicit list, possibly trimming unmoved elements if size is less than the maximum element in the permutation; if this is desired, setting size=-1 will guarantee such trimming.

Examples

```

>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]

```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Permutation(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
>>> Permutation(3).list(-1)
[]
```

max()

The maximum element moved by the permutation.

See also:

min (page 178), *descents* (page 170), *ascents* (page 168), *inversions* (page 174)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([1, 0, 2, 3, 4])
>>> p.max()
1
```

min()

The minimum element moved by the permutation.

See also:

max (page 178), *descents* (page 170), *ascents* (page 168), *inversions* (page 174)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 4, 3, 2])
>>> p.min()
2
```

mul_inv(*other*)

*other**~*self*, *self* and *other* have `_array_form`

next_lex()

Returns the next permutation in lexicographical order. If *self* is the last permutation in lexicographical order it returns `None`. See [4] section 2.4.

See also:

rank (page 180), *unrank_lex* (page 184)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([2, 3, 1, 0])
>>> p = Permutation([2, 3, 1, 0]); p.rank()
17
>>> p = p.next_lex(); p.rank()
18
```

next_nonlex()

Returns the next permutation in nonlex order [3]. If self is the last permutation in this order it returns None.

See also:

[rank_nonlex](#) (page 181), [unrank_nonlex](#) (page 184)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([2, 0, 3, 1]); p.rank_nonlex()
5
>>> p = p.next_nonlex(); p
Permutation([3, 0, 1, 2])
>>> p.rank_nonlex()
6
```

next_trotterjohnson()

Returns the next permutation in Trotter-Johnson order. If self is the last permutation it returns None. See [4] section 2.4. If it is desired to generate all such permutations, they can be generated in order more quickly with the `generate_bell` function.

See also:

[rank_trotterjohnson](#) (page 181), [unrank_trotterjohnson](#) (page 184), [diofant.utilities.iterables.generate_bell](#) (page 1146)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 0, 2, 1])
>>> p.rank_trotterjohnson()
4
>>> p = p.next_trotterjohnson(); p
Permutation([0, 3, 2, 1])
>>> p.rank_trotterjohnson()
5
```

order()

Computes the order of a permutation.

When the permutation is raised to the power of its order it equals the identity permutation.

See also:

[is_Identity](#) (page 175), [cardinality](#) (page 168), [length](#) (page 177), [rank](#) (page 180), [size](#) (page 183)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Permutation([3, 1, 5, 2, 4, 0])
>>> p.order()
4
>>> (p**(p.order()))
Permutation([], size=6)
```

parity()

Computes the parity of a permutation.

The parity of a permutation reflects the parity of the number of inversions in the permutation, i.e., the number of pairs of x and y such that $x > y$ but $p[x] < p[y]$.

See also:

[_af_parity](#) (page 186)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.parity()
0
>>> p = Permutation([3, 2, 0, 1])
>>> p.parity()
1
```

classmethod random(n)

Generates a random permutation of length n .

Uses the underlying Python pseudo-random number generator.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.random(2) in (Permutation([1, 0]), Permutation([0, 1]))
True
```

rank()

Returns the lexicographic rank of the permutation.

See also:

[next_lex](#) (page 178), [unrank_lex](#) (page 184), [cardinality](#) (page 168), [length](#) (page 177), [order](#) (page 179), [size](#) (page 183)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank()
```



```
0
>>> p = Permutation([3, 2, 1, 0])
>>> p.rank()
23
```

rank_nonlex(*inv_perm=None*)

This is a linear time ranking algorithm that does not enforce lexicographic order [3].

See also:

[next_nonlex](#) (page 178), [unrank_nonlex](#) (page 184)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_nonlex()
23
```

rank_trotterjohnson()

Returns the Trotter Johnson rank, which we get from the minimal change algorithm. See [4] section 2.4.

See also:

[unrank_trotterjohnson](#) (page 184), [next_trotterjohnson](#) (page 179)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2, 3])
>>> p.rank_trotterjohnson()
0
>>> p = Permutation([0, 2, 1, 3])
>>> p.rank_trotterjohnson()
7
```

static rmul(*args)

Return product of Permutations [a, b, c, ...] as the Permutation whose *i*th value is $a(b(c(i)))$.

a, b, c, ... can be Permutation objects or tuples.

Notes

All items in the sequence will be parsed by Permutation as necessary as long as the first item is a Permutation:

```
>>> Permutation.rmul(a, [0, 2, 1]) == Permutation.rmul(a, b)
True
```

The reverse order of arguments will raise a `TypeError`.

Examples

```
>>> from diofant.combinatorics.permutations import _af_rmul, Permutation
>>> Permutation.print_cyclic = False
```

```
>>> a, b = [1, 0, 2], [0, 2, 1]
>>> a = Permutation(a); b = Permutation(b)
>>> list(Permutation.rmul(a, b))
[1, 2, 0]
>>> [a(b(i)) for i in range(3)]
[1, 2, 0]
```

This handles the operands in reverse order compared to the `*` operator:

```
>>> a = Permutation(a); b = Permutation(b)
>>> list(a*b)
[2, 0, 1]
>>> [b(a(i)) for i in range(3)]
[2, 0, 1]
```

static `rmul_with_af(*args)`

same as `rmul`, but the elements of `args` are `Permutation` objects which have `_array_form`

runs()

Returns the runs of a permutation.

An ascending sequence in a permutation is called a run [5].

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([2, 5, 7, 3, 6, 0, 1, 4, 8])
>>> p.runs()
[[2, 5, 7], [3, 6], [0, 1, 4, 8]]
>>> q = Permutation([1,3,2,0])
>>> q.runs()
[[1, 3], [2], [0]]
```

signature()

Gives the signature of the permutation needed to place the elements of the permutation in canonical order.

The signature is calculated as $(-1)^{\langle \text{number of inversions} \rangle}$

See also:

[*inversions*](#) (page 174)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([0, 1, 2])
>>> p.inversions()
0
```

```
>>> p.signature()
1
>>> q = Permutation([0,2,1])
>>> q.inversions()
1
>>> q.signature()
-1
```

size

Returns the number of elements in the permutation.

See also:

cardinality (page 168), *length* (page 177), *order* (page 179), *rank* (page 180)

Examples

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

support()

Return the elements in permutation, P, for which $P[i] \neq i$.

Examples

```
>>> from diofant.combinatorics import Permutation
>>> p = Permutation([[3, 2], [0, 1], [4]])
>>> p.array_form
[1, 0, 3, 2, 4]
>>> p.support()
[0, 1, 2, 3]
```

transpositions()

Return the permutation decomposed into a list of transpositions.

It is always possible to express a permutation as the product of transpositions, see [1]

References

1. http://en.wikipedia.org/wiki/Transposition_%28mathematics%29#Properties

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> p = Permutation([[1, 2, 3], [0, 4, 5, 6, 7]])
>>> t = p.transpositions()
>>> t
[(0, 7), (0, 6), (0, 5), (0, 4), (1, 3), (1, 2)]
>>> print(''.join(str(c) for c in t))
(0, 7)(0, 6)(0, 5)(0, 4)(1, 3)(1, 2)
```

```
>>> Permutation.rmul(*[Permutation([ti], size=p.size) for ti in t]) == p
True
```

classmethod unrank_lex(*size*, *rank*)
Lexicographic permutation unranking.

See also:

[rank](#) (page 180), [next_lex](#) (page 178)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> a = Permutation.unrank_lex(5, 10)
>>> a.rank()
10
>>> a
Permutation([0, 2, 4, 1, 3])
```

classmethod unrank_nonlex(*n*, *r*)
This is a linear time unranking algorithm that does not respect lexicographic order [3].

See also:

[next_nonlex](#) (page 178), [rank_nonlex](#) (page 181)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> Permutation.unrank_nonlex(4, 5)
Permutation([2, 0, 3, 1])
>>> Permutation.unrank_nonlex(4, -1)
Permutation([0, 1, 2, 3])
```

classmethod unrank_trotterjohnson(*size*, *rank*)
Trotter Johnson permutation unranking. See [4] section 2.4.

See also:

[rank_trotterjohnson](#) (page 181), [next_trotterjohnson](#) (page 179)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.unrank_trotterjohnson(5, 10)
Permutation([0, 3, 1, 2, 4])
```

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:

```
>>> from diofant.combinatorics.permutations import Perm, Cycle
```

A Cycle will automatically parse a cycle given as a tuple on the rhs:

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

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

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

The array form of a Cycle can be obtained by calling the list method (or passing it to the list function) and all elements from 0 will be shown:

```
>>> a = Cycle(1, 2)
>>> a.list()
[0, 2, 1]
>>> list(a)
[0, 2, 1]
```

If a larger (or smaller) range is desired use the list method and provide the desired size - but the Cycle cannot be truncated to a size smaller than the largest element that is out of place:

```
>>> b = Cycle(2, 4)(1, 2)(3, 1, 4)(1, 3)
>>> b.list()
[0, 2, 1, 3, 4]
>>> b.list(b.size + 1)
[0, 2, 1, 3, 4, 5]
>>> b.list(-1)
[0, 2, 1]
```

Singletons are not shown when printing with one exception: the largest element is always shown - as a singleton if necessary:

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

The array form can be used to instantiate a Permutation so other properties of the permutation can be investigated:

```
>>> Perm(Cycle(1, 2)(3, 4).list()).transpositions()
[(1, 2), (3, 4)]
```

See also:

[Permutation](#) (page 163)

Notes

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

```
>>> list(Cycle(1, 2).items())
[(1, 2), (2, 1)]
```

`list(size=None)`

Return the cycles as an explicit list starting from 0 up to the greater of the largest value in the cycles and size.

Truncation of trailing unmoved items will occur when size is less than the maximum element in the cycle; if this is desired, setting `size=-1` will guarantee such trimming.

Examples

```
>>> from diofant.combinatorics.permutations import Cycle
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = False
>>> p = Cycle(2, 3)(4, 5)
>>> p.list()
[0, 1, 3, 2, 5, 4]
>>> p.list(10)
[0, 1, 3, 2, 5, 4, 6, 7, 8, 9]
```

Passing a length too small will trim trailing, unchanged elements in the permutation:

```
>>> Cycle(2, 4)(1, 2, 4).list(-1)
[0, 2, 1]
```

`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 $p[x] < p[y]$.

See also:

[Permutation](#) (page 163)

Examples

```
>>> from diofant.combinatorics.permutations import _af_parity
>>> _af_parity([0, 1, 2, 3])
0
>>> _af_parity([3, 2, 0, 1])
1
```

Generators

`generators.symmetric(n)`

Generates the symmetric group of order n , S_n .

Examples

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

`generators.cyclic(n)`

Generates the cyclic group of order n , C_n .

See also:

[*dihedral*](#) (page 187)

Examples

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

`generators.alternating(n)`

Generates the alternating group of order n , A_n .

Examples

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

`generators.dihedral(n)`

Generates the dihedral group of order $2n$, D_n .

The result is given as a subgroup of S_n , except for the special cases $n=1$ (the group S_2) and $n=2$ (the Klein 4-group) where that's not possible and embeddings in S_2 and S_4 respectively are given.

See also:

[*cyclic*](#) (page 187)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.generators import dihedral
>>> list(dihedral(3))
```

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

3.2.3 Permutation Groups

class diofant.combinatorics.perm_groups.PermutationGroup

The class defining a Permutation group.

PermutationGroup([p1, p2, ..., pn]) returns the permutation group generated by the list of permutations. This group can be supplied to Polyhedron if one desires to decorate the elements to which the indices of the permutation refer.

See also:

diofant.combinatorics.polyhedron.Polyhedron (page 216), *diofant.combinatorics.permutations.Permutation* (page 163)

References

- [1] Holt, D., Eick, B., O'Brien, E. "Handbook of Computational Group Theory"
- [2] Seress, A. "Permutation Group Algorithms"
- [3] http://en.wikipedia.org/wiki/Schreier_vector
- [4] http://en.wikipedia.org/wiki/Nielsen_transformation #Product_replacement_algorithm
- [5] Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E.A. O'Brien. "Generating Random Elements of a Finite Group"
- [6] http://en.wikipedia.org/wiki/Block_permutation_group_theory
- [7] http://www.algorithmist.com/index.php/Union_Find
- [8] http://en.wikipedia.org/wiki/Multiply_transitive_group#Multiply_transitive_groups
- [9] http://en.wikipedia.org/wiki/Center_group_theory
- [10] http://en.wikipedia.org/wiki/Centralizer_and_normalizer
- [11] http://groupprops.subwiki.org/wiki/Derived_subgroup
- [12] http://en.wikipedia.org/wiki/Nilpotent_group
- [13] <http://www.math.colostate.edu/~hulpke/CGT/cgtnotes.pdf>

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.permutations import Cycle
>>> from diofant.combinatorics.polyhedron import Polyhedron
>>> from diofant.combinatorics.perm_groups import PermutationGroup
```

The permutations corresponding to motion of the front, right and bottom face of a 2x2 Rubik's cube are defined:


```
>>> F = Permutation(2, 19, 21, 8)(3, 17, 20, 10)(4, 6, 7, 5)
>>> R = Permutation(1, 5, 21, 14)(3, 7, 23, 12)(8, 10, 11, 9)
>>> D = Permutation(6, 18, 14, 10)(7, 19, 15, 11)(20, 22, 23, 21)
```

These are passed as permutations to `PermutationGroup`:

```
>>> G = PermutationGroup(F, R, D)
>>> G.order()
3674160
```

The group can be supplied to a `Polyhedron` in order to track the objects being moved. An example involving the 2x2 Rubik's cube is given there, but here is a simple demonstration:

```
>>> a = Permutation(2, 1)
>>> b = Permutation(1, 0)
>>> G = PermutationGroup(a, b)
>>> P = Polyhedron(list('ABC'), pgroup=G)
>>> P.corners
(A, B, C)
>>> P.rotate(0) # apply permutation 0
>>> P.corners
(A, C, B)
>>> P.reset()
>>> P.corners
(A, B, C)
```

Or one can make a permutation as a product of selected permutations and apply them to an iterable directly:

```
>>> P10 = G.make_perm([0, 1])
>>> P10('ABC')
['C', 'A', 'B']
```

`__contains__(i)`

Return True if *i* is contained in `PermutationGroup`.

Examples

```
>>> from diofant.combinatorics import Permutation, PermutationGroup
>>> 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

```
>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> 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

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import CyclicGroup
>>> 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__` (**args, **kwargs*)

The default constructor. Accepts Cycle and Permutation forms. Removes duplicates unless `dups` keyword is False.

`_random_pr_init` (*r, n, _random_prec_n=None*)

Initialize random generators for the product replacement algorithm.

The implementation uses a modification of the original product replacement algorithm due to Leedham-Green, as described in [1], pp. 69-71; also, see [2], pp. 27-29 for a detailed theoretical analysis of the original product replacement algorithm, and [4].

The product replacement algorithm is used for producing random, uniformly distributed elements of a group G with a set of generators S . For the initialization `_random_pr_init`, a list R of $\max\{r, |S|\}$ group generators is created as the attribute `G._random_gens`, repeating elements of S if necessary, and the identity element of G is appended to R - we shall refer to this last element as the accumulator. Then the function `random_pr()` is called n times, randomizing the list R while preserving the generation of G by R . The function `random_pr()` itself takes two random elements g, h among all elements of R but the accumulator and replaces g with a randomly chosen element from $\{gh, g(\sim h), hg, (\sim h)g\}$. Then the accumulator is multiplied by whatever g was replaced by. The new value of the accumulator is then returned by `random_pr()`.

The elements returned will eventually (for n large enough) become uniformly distributed across G ([5]). For practical purposes however, the values $n = 50, r = 11$ are suggested in [1].

See also:

[random_pr](#) (page 211)

Notes

THIS FUNCTION HAS SIDE EFFECTS: it changes the attribute `self._random_gens`

`_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])

See also:

[*minimal_block*](#) (page 207), [*_union_find_rep*](#) (page 191)

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.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[7] http://www.algorithmist.com/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]).

See also:

[*minimal_block*](#) (page 207), [*_union_find_merge*](#) (page 191)

Notes

THIS FUNCTION HAS SIDE EFFECTS: the list of class representatives, `parents`, is altered due to path compression.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

[7] http://www.algorithmist.com/index.php/Union_Find

base

Return a base from the Schreier-Sims algorithm.

For a permutation group G , a base is a sequence of points $B = (b_1, b_2, \dots, b_k)$ such that no element of G apart from the identity fixes all the points in B . The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

An alternative way to think of `B` is that it gives the indices of the stabilizer cosets that contain more than the identity permutation.

See also:

[strong_gens](#) (page 214), [basic_transversals](#) (page 194), [basic_orbits](#) (page 193), [basic_stabilizers](#) (page 193)

Examples

```
>>> from diofant.combinatorics import Permutation, PermutationGroup
>>> G = PermutationGroup([Permutation(0, 1, 3)(2, 4)])
>>> G.base
[0, 2]
```

baseswap(*base*, *strong_gens*, *pos*, *randomized=False*, *transversals=None*, *basic_orbits=None*, *strong_gens_distr=None*)

Swap two consecutive base points in *base* and *strong* generating set.

If a base for a group G is given by (b_1, b_2, \dots, b_k) , this function returns a base $(b_1, b_2, \dots, b_{i+1}, b_i, \dots, b_k)$, where i is given by *pos*, and a strong generating set relative to that base. The original base and strong generating set are not modified.

The randomized version (default) is of Las Vegas type.

Parameters *base*, *strong_gens*

The base and strong generating set.

pos

The position at which swapping is performed.

randomized

A switch between randomized and deterministic version.

transversals

The transversals for the basic orbits, if known.

basic_orbits

The basic orbits, if known.

strong_gens_distr

The strong generators distributed by basic stabilizers, if known.

Returns (*base*, *strong_gens*)

base is the new base, and *strong_gens* is a generating set relative to it.

See also:

[schreier_sims](#) (page 211)

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.

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> S.base
[0, 1, 2]
>>> base, gens = S.baseswap(S.base, S.strong_gens, 1, randomized=False)
>>> base, gens
([0, 2, 1],
 [Permutation(0, 1, 2, 3), Permutation(3)(0, 1), Permutation(1, 3, 2),
  Permutation(2, 3), Permutation(1, 3)])
```

check that base, gens is a BSGS

```
>>> S1 = PermutationGroup(gens)
>>> _verify_bsgs(S1, base, gens)
True
```

basic_orbits

Return the basic orbits relative to a base and strong generating set.

If (b_1, b_2, \dots, b_k) is a base for a group G , and $G^{\{i\}} = G_{\{b_1, b_2, \dots, b_{i-1}\}}$ is the i -th basic stabilizer (so that $G^{\{1\}} = G$), the i -th basic orbit relative to this base is the orbit of b_i under $G^{\{i\}}$. See [1], pp. 87-89 for more information.

See also:

[base](#) (page 191), [strong_gens](#) (page 214), [basic_transversals](#) (page 194), [basic_stabilizers](#) (page 193)

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(4)
>>> S.basic_orbits
[[0, 1, 2, 3], [1, 2, 3], [2, 3]]
```

basic_stabilizers

Return a chain of stabilizers relative to a base and strong generating set.

The i -th basic stabilizer $G^{\{i\}}$ relative to a base (b_1, b_2, \dots, b_k) is $G_{\{b_1, b_2, \dots, b_{i-1}\}}$. For more information, see [1], pp. 87-89.

See also:

[base](#) (page 191), [strong_gens](#) (page 214), [basic_orbits](#) (page 193), [basic_transversals](#) (page 194)

Examples

```
>>> from diofant.combinatorics.named_groups import AlternatingGroup
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> A.base
[0, 1]
>>> for g in A.basic_stabilizers:
...     print(g)
...
PermutationGroup([
    Permutation(3)(0, 1, 2),
    Permutation(1, 2, 3)])
PermutationGroup([
    Permutation(1, 2, 3)])
```

basic_transversals

Return basic transversals relative to a base and strong generating set.

The basic transversals are transversals of the basic orbits. They are provided as a list of dictionaries, each dictionary having keys - the elements of one of the basic orbits, and values - the corresponding transversal elements. See [1], pp. 87-89 for more information.

See also:

[strong_gens](#) (page 214), [base](#) (page 191), [basic_orbits](#) (page 193), [basic_stabilizers](#) (page 193)

Examples

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

center()

Return the center of a permutation group.

The center for a group G is defined as $Z(G) = \{z \in G \mid \forall g \in G, zg = gz\}$, the set of elements of G that commute with all elements of G . It is equal to the centralizer of G inside G , and is naturally a subgroup of G ([9]).

See also:

[centralizer](#) (page 195)

Notes

This is a naive implementation that is a straightforward application of `.centralizer()`

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(4)
>>> G = D.center()
>>> G.order()
2
```

centralizer(*other*)

Return the centralizer of a group/set/element.

The centralizer of a set of permutations S inside a group G is the set of elements of G that commute with all elements of S :

$$C_G(S) = \{ g \in G \mid gs = sg \text{ for all } s \in S \}$$
 ([10])

Usually, S is a subset of G , but if G is a proper subgroup of the full symmetric group, we allow for S to have elements outside G .

It is naturally a subgroup of G ; the centralizer of a permutation group is equal to the centralizer of any set of generators for that group, since any element commuting with the generators commutes with any product of the generators.

Parameters other

a permutation group/list of permutations/single permutation

See also:

[subgroup_search](#) (page 215)

Notes

The implementation is an application of `.subgroup_search()` with tests using a specific base for the group G .

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup)
>>> S = SymmetricGroup(6)
>>> C = CyclicGroup(6)
>>> H = S.centralizer(C)
>>> H.is_subgroup(C)
True
```

commutator(G, H)

Return the commutator of two subgroups.

For a permutation group K and subgroups G, H , the commutator of G and H is defined as the group generated by all the commutators $[g, h] = hgh^{-1}g^{-1}$ for g in G and h in H . It is naturally a subgroup of K ([1], p.27).

See also:

[derived_subgroup](#) (page 199)

Notes

The commutator of two subgroups H, G is equal to the normal closure of the commutators of all the generators, i.e. $hgh^{-1}g^{-1}$ for h a generator of H and g a generator of G ([1], p.28)

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> G = S.commutator(S, A)
>>> G.is_subgroup(A)
True
```

contains(g , *strict=True*)

Test if permutation g belong to self, G .

If g is an element of G it can be written as a product of factors drawn from the cosets of G 's stabilizers. To see if g is one of the actual generators defining the group use $G.has(g)$.

If *strict* is not `True`, g will be resized, if necessary, to match the size of permutations in self.

See also:

coset_factor (page 197), *diofant.core.basic.Basic.has* (page 47)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
```

```
>>> a = Permutation(1, 2)
>>> b = Permutation(2, 3, 1)
>>> G = PermutationGroup(a, b, degree=5)
>>> G.contains(G[0]) # trivial check
True
>>> elem = Permutation([[2, 3]], size=5)
>>> G.contains(elem)
True
>>> G.contains(Permutation(4)(0, 1, 2, 3))
False
```

If *strict* is `False`, a permutation will be resized, if necessary:

```
>>> H = PermutationGroup(Permutation(5))
>>> H.contains(Permutation(3))
False
>>> H.contains(Permutation(3), strict=False)
True
```


To test if a given permutation is present in the group:

```
>>> elem in G.generators
False
>>> G.has(elem)
False
```

coset_factor(*g*, *factor_index=False*)

Return G's (self's) coset factorization of *g*

If *g* is an element of *G* then it can be written as the product of permutations drawn from the Schreier-Sims coset decomposition,

The permutations returned in *f* are those for which the product gives *g*: $g = f[n]*\dots*f[1]*f[0]$ where $n = \text{len}(B)$ and $B = G.\text{base}$. *f*[*i*] is one of the permutations in *self._basic_orbits*[*i*].

If *factor_index==True*, returns a tuple [*b*[0],...,*b*[*n*]], where *b*[*i*] belongs to *self._basic_orbits*[*i*]

Examples

```
>>> from diofant.combinatorics import Permutation, PermutationGroup
>>> Permutation.print_cyclic = True
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
```

Define *g*:

```
>>> g = Permutation(7)(1, 2, 4)(3, 6, 5)
```

Confirm that it is an element of *G*:

```
>>> G.contains(g)
True
```

Thus, it can be written as a product of factors (up to 3) drawn from *u*. See below that a factor from *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`.

See also:

[`coset_factor`](#) (page 197)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(0, 1, 3, 7, 6, 4)(2, 5)
>>> b = Permutation(0, 1, 3, 2)(4, 5, 7, 6)
>>> G = PermutationGroup([a, b])
>>> c = Permutation(7)(2, 4)(3, 5)
>>> G.coset_rank(c)
16
>>> G.coset_unrank(16)
Permutation(7)(2, 4)(3, 5)
```

`coset_unrank(rank, af=False)`

unrank using Schreier-Sims representation

`coset_unrank` is the inverse operation of `coset_rank` if $0 \leq \text{rank} < \text{order}$; otherwise it returns `None`.

`degree`

Returns the size of the permutations in the group.

The number of permutations comprising the group is given by `len(group)`; the number of permutations that can be generated by the group is given by `group.order()`.

See also:

[`order`](#) (page 210)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[Permutation(2), Permutation(2)(0, 1)]
```

derived_series()

Return the derived series for the group.

The derived series for a group G is defined as $G = G_0 > G_1 > G_2 > \dots$ where $G_i = [G_{i-1}, G_{i-1}]$, i.e. G_i is the derived subgroup of G_{i-1} , for $i \in \mathbb{N}$. When we have $G_k = G_{k-1}$ for some $k \in \mathbb{N}$, the series terminates.

Returns A list of permutation groups containing the members of the derived series in the order $G = G_0, G_1, G_2, \dots$.

See also:

[derived_subgroup](#) (page 199)

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup, DihedralGroup)
>>> A = AlternatingGroup(5)
>>> len(A.derived_series())
1
>>> S = SymmetricGroup(4)
>>> len(S.derived_series())
4
>>> S.derived_series()[1].is_subgroup(AlternatingGroup(4))
True
>>> S.derived_series()[2].is_subgroup(DihedralGroup(2))
True
```

derived_subgroup()

Compute the derived subgroup.

The derived subgroup, or commutator subgroup is the subgroup generated by all commutators $[g, h] = hgh^{-1}g^{-1}$ for $g, h \in G$; it is equal to the normal closure of the set of commutators of the generators ([1], p.28, [11]).

See also:

[derived_series](#) (page 198)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 0, 2, 4, 3])
>>> b = Permutation([0, 1, 3, 2, 4])
>>> G = PermutationGroup([a, b])
>>> C = G.derived_subgroup()
>>> list(C.generate(af=True))
[[0, 1, 2, 3, 4], [0, 1, 3, 4, 2], [0, 1, 4, 2, 3]]
```

elements

Returns all the elements of the permutatio group in a list

Examples

```
>>> from diofant.combinatorics import Permutation
```

generate(*method='coset', af=False*)

Return iterator to generate the elements of the group

Iteration is done with one of these methods:

```
method='coset' using the Schreier-Sims coset representation
method='dimino' using the Dimino method
```

If *af* = True it yields the array form of the permutations

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics import PermutationGroup
>>> from diofant.combinatorics.polyhedron import tetrahedron
```

The permutation group given in the tetrahedron object is also true groups:

```
>>> G = tetrahedron.pgroup
>>> G.is_group
True
```

Also the group generated by the permutations in the tetrahedron pgroup - even the first two - is a proper group:

```
>>> H = PermutationGroup(G[0], G[1])
>>> J = PermutationGroup(list(H.generate())); J
PermutationGroup([
  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 Com-

puter Algebra System: AXIOM, N.J. Doye, M.Sc. Thesis

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_dimino(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 2, 3, 1],
 [0, 1, 3, 2], [0, 3, 2, 1], [0, 3, 1, 2]]
```

`generate_schreier_sims(af=False)`

Yield group elements using the Schreier-Sims representation in coset_rank order

If af = True it yields the array form of the permutations

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([0, 2, 3, 1])
>>> g = PermutationGroup([a, b])
>>> list(g.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [0, 2, 1, 3], [0, 3, 2, 1],
 [0, 1, 3, 2], [0, 2, 3, 1], [0, 3, 1, 2]]
```

generators

Returns the generators of the group.

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.generators
[Permutation(1, 2), Permutation(2)(0, 1)]
```

`is_abelian`

Test if the group is Abelian.

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.is_abelian
False
>>> a = Permutation([0, 2, 1])
>>> G = PermutationGroup([a])
>>> G.is_abelian
True
```

`is_alt_sym(eps=0.05, _random_prec=None)`

Monte Carlo test for the symmetric/alternating group for degrees ≥ 8 .

More specifically, it is one-sided Monte Carlo with the answer True (i.e., G is symmetric/alternating) guaranteed to be correct, and the answer False being incorrect with probability eps.

See also:

[diofant.combinatorics.util._check_cycles_alt_sym](#) (page 238)

Notes

The algorithm itself uses some nontrivial results from group theory and number theory: 1) If a transitive group G of degree n contains an element with a cycle of length $n/2 < p < n-2$ for p a prime, G is the symmetric or alternating group ([1], pp. 81-82) 2) The proportion of elements in the symmetric/alternating group having the property described in 1) is approximately $\log(2)/\log(n)$ ([1], p.82; [2], pp. 226-227). The helper function `_check_cycles_alt_sym` is used to go over the cycles in a permutation and look for ones satisfying 1).

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_alt_sym()
False
```

`is_nilpotent`

Test if the group is nilpotent.

A group G is nilpotent if it has a central series of finite length. Alternatively, G is nilpotent if its lower central series terminates with the trivial group. Every nilpotent group is also solvable ([1], p.29, [12]).

See also:

[lower_central_series](#) (page 205), [is_solvable](#) (page 204)

Examples

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

`is_normal(gr)`

Test if $G = \text{self}$ 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

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G1 = PermutationGroup([a, Permutation([2, 0, 1])])
>>> G1.is_normal(G)
True
```

`is_primitive(randomized=True)`

Test if a group is primitive.

A permutation group G acting on a set S is called primitive if S contains no nontrivial block under the action of G (a block is nontrivial if its cardinality is more than 1).

See also:

[*minimal_block*](#) (page 207), [*random_stab*](#) (page 211)

Notes

The algorithm is described in [1], p.83, and uses the function `minimal_block` to search for blocks of the form $\{\theta, k\}$ for k ranging over representatives for the orbits of G_θ , the stabilizer of θ . This algorithm has complexity $O(n^2)$ where n is the degree of the group, and will perform badly if G_θ is small.

There are two implementations offered: one finds G_θ deterministically using the function `stabilizer`, and the other (default) produces random elements of G_θ using `random_stab`, hoping that they generate a subgroup of G_θ with not too many more orbits than G_θ (this is suggested in [1], p.83). Behavior is changed by the `randomized` flag.

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> D = DihedralGroup(10)
>>> D.is_primitive()
False
```

`is_solvable`

Test if the group is solvable.

G is solvable if its derived series terminates with the trivial group ([1], p.29).

See also:

[*is_nilpotent*](#) (page 202), [*derived_series*](#) (page 198)

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(3)
>>> S.is_solvable
True
```

`is_subgroup(G, strict=True)`

Return True if all elements of self belong to G.

If `strict` is False then if self's degree is smaller than G's, the elements will be resized to have the same degree.

Examples

```
>>> from diofant.combinatorics import Permutation, PermutationGroup
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
...         CyclicGroup)
```

Testing `is_strict` by default: the degree of each group must be the same:

```
>>> p = Permutation(0, 1, 2, 3, 4, 5)
>>> G1 = PermutationGroup([Permutation(0, 1, 2), Permutation(0, 1)])
>>> G2 = PermutationGroup([Permutation(0, 2), Permutation(0, 1, 2)])
>>> G3 = PermutationGroup([p, p**2])
>>> assert G1.order() == G2.order() == G3.order() == 6
>>> G1.is_subgroup(G2)
True
>>> G1.is_subgroup(G3)
False
>>> G3.is_subgroup(PermutationGroup(G3[1]))
False
>>> G3.is_subgroup(PermutationGroup(G3[0]))
True
```

To ignore the size, set `strict` to False:


```

>>> S3 = SymmetricGroup(3)
>>> S5 = SymmetricGroup(5)
>>> S3.is_subgroup(S5, strict=False)
True
>>> C7 = CyclicGroup(7)
>>> G = S5*C7
>>> S5.is_subgroup(G, False)
True
>>> C7.is_subgroup(G, 0)
False

```

is_transitive(*strict=True*)

Test if the group is transitive.

A group is transitive if it has a single orbit.

If *strict* is *False* the group is transitive if it has a single orbit of length different from 1.

Examples

```

>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1, 3])
>>> b = Permutation([2, 0, 1, 3])
>>> G1 = PermutationGroup([a, b])
>>> G1.is_transitive()
False
>>> G1.is_transitive(strict=False)
True
>>> c = Permutation([2, 3, 0, 1])
>>> G2 = PermutationGroup([a, c])
>>> G2.is_transitive()
True
>>> d = Permutation([1, 0, 2, 3])
>>> e = Permutation([0, 1, 3, 2])
>>> G3 = PermutationGroup([d, e])
>>> G3.is_transitive() or G3.is_transitive(strict=False)
False

```

is_trivial

Test if the group is the trivial group.

This is true if the group contains only the identity permutation.

Examples

```

>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 1, 2])])
>>> G.is_trivial
True

```

lower_central_series()

Return the lower central series for the group.

The lower central series for a group G is the series $G = G_0 > G_1 > G_2 > \dots$ where $G_k = [G, G_{k-1}]$, i.e. every term after the first is equal to the commutator of G and the previous term in G ([1], p.29).

Returns A list of permutation groups in the order

$G = G_0, G_1, G_2, \dots$

See also:

[commutator](#) (page 195), [derived_series](#) (page 198)

Examples

```
>>> from diofant.combinatorics.named_groups import (AlternatingGroup,
... DihedralGroup)
>>> A = AlternatingGroup(4)
>>> len(A.lower_central_series())
2
>>> A.lower_central_series()[1].is_subgroup(DihedralGroup(2))
True
```

make_perm(n , *seed=None*)

Multiply n randomly selected permutations from `pgroup` together, starting with the identity permutation. If n is a list of integers, those integers will be used to select the permutations and they will be applied in L to R order: `make_perm((A, B, C))` will give `CBA(I)` where `I` is the identity permutation.

`seed` is used to set the seed for the random selection of permutations from `pgroup`. If this is a list of integers, the corresponding permutations from `pgroup` will be selected in the order give. This is mainly used for testing purposes.

See also:

[random](#) (page 211)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a, b = [Permutation([1, 0, 3, 2]), Permutation([1, 3, 0, 2])]
>>> G = PermutationGroup([a, b])
>>> G.make_perm(1, [0])
Permutation(0, 1)(2, 3)
>>> G.make_perm(3, [0, 1, 0])
Permutation(0, 2, 3, 1)
>>> G.make_perm([0, 1, 0])
Permutation(0, 2, 3, 1)
```

max_div

Maximum proper divisor of the degree of a permutation group.

See also:

[minimal_block](#) (page 207), [_union_find_merge](#) (page 191)

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

```
>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> G = PermutationGroup([Permutation([0, 2, 1, 3])])
>>> G.max_div
2
```

`minimal_block(points)`

For a transitive group, finds the block system generated by points.

If a group G acts on a set S , a nonempty subset B of S is called a block under the action of G if for all g in G we have $gB = B$ (g fixes B) or gB and B have no common points (g moves B entirely). ([1], p.23; [6]).

The distinct translates gB of a block B for g in G partition the set S and this set of translates is known as a block system. Moreover, we obviously have that all blocks in the partition have the same size, hence the block size divides $|S|$ ([1], p.23). A G -congruence is an equivalence relation \sim on the set S such that $a \sim b$ implies $g(a) \sim g(b)$ for all g in G . For a transitive group, the equivalence classes of a G -congruence and the blocks of a block system are the same thing ([1], p.23).

The algorithm below checks the group for transitivity, and then finds the G -congruence generated by the pairs $(p_0, p_1), (p_0, p_2), \dots, (p_0, p_{k-1})$ which is the same as finding the maximal block system (i.e., the one with minimum block size) such that p_0, \dots, p_{k-1} are in the same block ([1], p.83).

It is an implementation of Atkinson's algorithm, as suggested in [1], and manipulates an equivalence relation on the set S using a union-find data structure. The running time is just above $O(|points| |S|)$. ([1], pp. 83-87; [7]).

See also:

[_union_find_rep](#) (page 191), [_union_find_merge](#) (page 191), [is_transitive](#) (page 205), [is_primitive](#) (page 203)

Examples

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

`normal_closure(other, k=10)`

Return the normal closure of a subgroup/set of permutations.

If S is a subset of a group G , the normal closure of A in G is defined as the intersection of all normal subgroups of G that contain A ([1], p.14). Alternatively, it is the group generated by the conjugates $x^{-1}yx$ for x a generator of G and y a generator of the subgroup $\langle S \rangle$ 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

See also:

[commutator](#) (page 195), [derived_subgroup](#) (page 199), [random_pr](#) (page 211)

Notes

The algorithm is described in [1], pp. 73-74; it makes use of the generation of random elements for permutation groups by the product replacement algorithm.

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... CyclicGroup, AlternatingGroup)
>>> S = SymmetricGroup(5)
>>> C = CyclicGroup(5)
>>> G = S.normal_closure(C)
>>> G.order()
60
>>> G.is_subgroup(AlternatingGroup(5))
True
```

orbit(alpha, action='tuples')

Compute the orbit of α $\{g(\alpha) \mid g \in G\}$ 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

See also:

[orbit_transversal](#) (page 209)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([1, 2, 0, 4, 5, 6, 3])
>>> G = PermutationGroup([a])
>>> G.orbit(0)
{0, 1, 2}
>>> G.orbit([0, 4], 'union')
{0, 1, 2, 3, 4, 5, 6}
```

orbit_rep(*alpha*, *beta*, *schreier_vector*=None)

Return a group element which sends alpha to beta.

If beta is not in the orbit of alpha, the function returns False. This implementation makes use of the schreier vector. For a proof of correctness, see [1], p.80

See also:

[schreier_vector](#) (page 213)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(5)
>>> G.orbit_rep(0, 4)
Permutation(0, 4, 1, 2, 3)
```

orbit_transversal(*alpha*, *pairs*=False)

Computes a transversal for the orbit of alpha as a set.

For a permutation group G , a transversal for the orbit $\text{Orb} = \{g(\alpha) \mid g \in G\}$ is a set $\{g_\beta \mid g_\beta(\alpha) = \beta\}$ for $\beta \in \text{Orb}$. Note that there may be more than one possible transversal. If *pairs* is set to True, it returns the list of pairs (β, g_β) . For a proof of correctness, see [1], p.79

See also:

[orbit](#) (page 208)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.orbit_transversal(0)
[Permutation(5),
 Permutation(0, 1, 2, 3, 4, 5),
 Permutation(0, 5)(1, 4)(2, 3),
 Permutation(0, 2, 4)(1, 3, 5),
```

```
Permutation(5)(0, 4)(1, 3),
Permutation(0, 3)(1, 4)(2, 5)]
```

orbits(*rep=False*)

Return the orbits of self, ordered according to lowest element in each orbit.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation(1, 5)(2, 3)(4, 0, 6)
>>> b = Permutation(1, 5)(3, 4)(2, 6, 0)
>>> G = PermutationGroup([a, b])
>>> G.orbits()
[0, 2, 3, 4, 6], {1, 5}]
```

order()

Return the order of the group: the number of permutations that can be generated from elements of the group.

The number of permutations comprising the group is given by `len(group)`; the length of each permutation in the group is given by `group.size`.

See also:

[degree](#) (page 198)

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
```

```
>>> a = Permutation([1, 0, 2])
>>> G = PermutationGroup([a])
>>> G.degree
3
>>> len(G)
1
>>> G.order()
2
>>> list(G.generate())
[Permutation(2), Permutation(2)(0, 1)]
```

```
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.order()
6
```

pointwise_stabilizer(*points, incremental=True*)

Return the pointwise stabilizer for a set of points.

For a permutation group G and a set of points $\{p_1, p_2, \dots, p_k\}$, 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 .

See also:

[stabilizer](#) (page 214), [schreier_sims_incremental](#) (page 212)

Notes

When `incremental == True`, rather than the obvious implementation using successive calls to `.stabilizer()`, this uses the incremental Schreier-Sims algorithm to obtain a base with starting segment - the given points.

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(7)
>>> Stab = S.pointwise_stabilizer([2, 3, 5])
>>> Stab.is_subgroup(S.stabilizer(2).stabilizer(3).stabilizer(5))
True
```

random(*af=False*)

Return a random group element

random_pr(*gen_count=11, iterations=50, _random_prec=None*)

Return a random group element using product replacement.

For the details of the product replacement algorithm, see `_random_pr_init`. In `random_pr` the actual ‘product replacement’ is performed. Notice that if the attribute `_random_gens` is empty, it needs to be initialized by `_random_pr_init`.

See also:

[_random_pr_init](#) (page 190)

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 211), [orbit_rep](#) (page 209)

schreier_sims()

Schreier-Sims algorithm.

It computes the generators of the chain of stabilizers $G > G_{\{b_1\}} > \dots > G_{\{b_1, \dots, b_r\}} > 1$ in which $G_{\{b_1, \dots, b_i\}}$ stabilizes b_1, \dots, b_i , and the corresponding s cosets. An element of the group can be written as the product $h_1 * \dots * h_s$.

We use the incremental Schreier-Sims algorithm.

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> a = Permutation([0, 2, 1])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_sims()
>>> G.basic_transversals
[{0: Permutation(2)(0, 1), 1: Permutation(2), 2: Permutation(1, 2)},
 {0: Permutation(2), 2: Permutation(0, 2)}]
```

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

Extend a sequence of points and generating set to a base and strong generating set.

Parameters *base*

The sequence of points to be extended to a base. Optional parameter with default value [].

gens

The generating set to be extended to a strong generating set relative to the base obtained. Optional parameter with default value `self.generators`.

Returns (*base, strong_gens*)

base is the base obtained, and *strong_gens* is the strong generating set relative to it. The original parameters *base*, *gens* remain unchanged.

See also:

[schreier_sims](#) (page 211), [schreier_sims_random](#) (page 212)

Notes

This version of the Schreier-Sims algorithm runs in polynomial time. There are certain assumptions in the implementation - if the trivial group is provided, *base* and *gens* are returned immediately, as any sequence of points is a base for the trivial group. If the identity is present in the generators *gens*, it is removed as it is a redundant generator. The implementation is described in [1], pp. 90-93.

Examples

```
>>> from diofant.combinatorics.named_groups import AlternatingGroup
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> 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]
```


`schreier_sims_random`(*base=None*, *gens=None*, *consec_succ=10*, *_random_prec=None*)

Randomized Schreier-Sims algorithm.

The randomized Schreier-Sims algorithm takes the sequence *base* and the generating set *gens*, and extends *base* to a base, and *gens* to a strong generating set relative to that base with probability of a wrong answer at most $2^{-\text{consec_succ}}$, provided the random generators are sufficiently random.

Parameters

base
The sequence to be extended to a base.

gens

The generating set to be extended to a strong generating set.

consec_succ

The parameter defining the probability of a wrong answer.

_random_prec

An internal parameter used for testing purposes.

Returns (base, strong_gens)

base is the base and *strong_gens* is the strong generating set relative to it.

See also:

[schreier_sims](#) (page 211)

Notes

The algorithm is described in detail in [1], pp. 97-98. It extends the orbits *orbs* and the permutation groups *stabs* to basic orbits and basic stabilizers for the base and strong generating set produced in the end. The idea of the extension process is to “sift” random group elements through the stabilizer chain and amend the stabilizers/orbits along the way when a sift is not successful. The helper function `_strip` is used to attempt to decompose a random group element according to the current state of the stabilizer chain and report whether the element was fully decomposed (successful sift) or not (unsuccessful sift). In the latter case, the level at which the sift failed is reported and used to amend *stabs*, *base*, *gens* and *orbs* accordingly. The halting condition is for *consec_succ* consecutive successful sifts to pass. This makes sure that the current *base* and *gens* form a BSGS with probability at least $1 - 1/\text{consec_succ}$.

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> S = SymmetricGroup(5)
>>> base, strong_gens = S.schreier_sims_random(consec_succ=5)
>>> _verify_bsgs(S, base, strong_gens)
True
```

schreier_vector(*alpha*)

Computes the schreier vector for *alpha*.

The Schreier vector efficiently stores information about the orbit of *alpha*. It can later be used to quickly obtain elements of the group that send *alpha* to a particular element in the orbit. Notice that the Schreier vector depends on the order in which the group generators are listed. For a definition, see [3]. Since list indices start from zero, we adopt the convention to use “None” instead of 0 to signify that an element doesn’t belong to the orbit. For the algorithm and its correctness, see [2], pp.78-80.

See also:

orbit (page 208)

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.permutations import Permutation
>>> a = Permutation([2, 4, 6, 3, 1, 5, 0])
>>> b = Permutation([0, 1, 3, 5, 4, 6, 2])
>>> G = PermutationGroup([a, b])
>>> G.schreier_vector(0)
[-1, None, 0, 1, None, 1, 0]
```

stabilizer(*alpha*)

Return the stabilizer subgroup of *alpha*.

The stabilizer of α is the group $G_{\alpha} = \{g \in G \mid g(\alpha) = \alpha\}$. For a proof of correctness, see [1], p.79.

See also:

orbit (page 208)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(6)
>>> G.stabilizer(5)
PermutationGroup([
  Permutation(5)(0, 4)(1, 3),
  Permutation(5)])
```

strong_gens

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

A generating set $S = \{g_1, g_2, \dots, g_t\}$ for a permutation group G is a strong generating set relative to the sequence of points (referred to as a “base”) (b_1, b_2, \dots, b_k) if, for $1 \leq i \leq k$ we have that the intersection of the pointwise stabilizer $G^{\{(i+1)\}} := G_{\{b_1, b_2, \dots, b_i\}}$ with S generates the pointwise stabilizer $G^{\{(i+1)\}}$. The concepts of a base and strong generating set and their applications are discussed in depth in [1], pp. 87-89 and [2], pp. 55-57.

See also:

base (page 191), *basic_transversals* (page 194), *basic_orbits* (page 193), *basic_stabilizers* (page 193)

Examples

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

subgroup_search(*prop*, *base=None*, *strong_gens=None*, *tests=None*, *init_subgroup=None*)

Find the subgroup of all elements satisfying the property *prop*.

This is done by a depth-first search with respect to base images that uses several tests to prune the search tree.

Parameters *prop*

The property to be used. Has to be callable on group elements and always return True or False. It is assumed that all group elements satisfying *prop* indeed form a subgroup.

base

A base for the supergroup.

strong_gens

A strong generating set for the supergroup.

tests

A list of callables of length equal to the length of *base*. These are used to rule out group elements by partial base images, so that *tests[l](g)* returns False if the element *g* is known not to satisfy *prop* base on where *g* sends the first $l + 1$ base points.

init_subgroup

if a subgroup of the sought group is known in advance, it can be passed to the function as this parameter.

Returns *res*

The subgroup of all elements satisfying *prop*. The generating set for this group is guaranteed to be a strong generating set relative to the base *base*.

Notes

This function is extremely lengthy and complicated and will require some careful attention. The implementation is described in [1], pp. 114-117, and the comments for the code here follow the lines of the pseudocode in the book for clarity.

The complexity is exponential in general, since the search process by itself visits all members of the supergroup. However, there are a lot of tests which are used to

prune the search tree, and users can define their own tests via the `tests` parameter, so in practice, and for some computations, it's not terrible.

A crucial part in the procedure is the frequent base change performed (this is line 11 in the pseudocode) in order to obtain a new basic stabilizer. The book mentions that this can be done by using `.baseswap(...)`, however the current implementation uses a more straightforward way to find the next basic stabilizer - calling the function `.stabilizer(...)` on the previous basic stabilizer.

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> S = SymmetricGroup(7)
>>> prop_even = lambda x: x.is_even
>>> base, strong_gens = S.schreier_sims_incremental()
>>> G = S.subgroup_search(prop_even, base=base, strong_gens=strong_gens)
>>> G.is_subgroup(AlternatingGroup(7))
True
>>> _verify_bsgs(G, base, G.generators)
True
```

transitivity_degree

Compute the degree of transitivity of the group.

A permutation group G acting on $\Omega = \{0, 1, \dots, n-1\}$ is k -fold transitive, if, for any k points $(a_1, a_2, \dots, a_k) \in \Omega$ and any k points $(b_1, b_2, \dots, b_k) \in \Omega$ there exists $g \in G$ such that $g(a_1) = b_1, g(a_2) = b_2, \dots, g(a_k) = b_k$. The degree of transitivity of G is the maximum k such that G is k -fold transitive. ([8])

See also:

[is_transitive](#) (page 205), [orbit](#) (page 208)

Examples

```
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.permutations import Permutation
>>> a = Permutation([1, 2, 0])
>>> b = Permutation([1, 0, 2])
>>> G = PermutationGroup([a, b])
>>> G.transitivity_degree
3
```

3.2.4 Polyhedron

`class diofant.combinatorics.polyhedron.Polyhedron`

Represents the polyhedral symmetry group (PSG).

The PSG is one of the symmetry groups of the Platonic solids. There are three polyhedral groups: the tetrahedral group of order 12, the octahedral group of order 24, and the

icosahedral group of order 60.

All doctests have been given in the docstring of the constructor of the object.

References

<http://mathworld.wolfram.com/PolyhedralGroup.html>

array_form

Return the indices of the corners.

The indices are given relative to the original position of corners.

See also:

corners (page 217), *cyclic_form* (page 217)

Examples

```
>>> from diofant.combinatorics import Permutation, Cycle
>>> from diofant.combinatorics.polyhedron import tetrahedron
>>> tetrahedron.array_form
[0, 1, 2, 3]
```

```
>>> tetrahedron.rotate(0)
>>> tetrahedron.array_form
[0, 2, 3, 1]
>>> tetrahedron.pgroup[0].array_form
[0, 2, 3, 1]
>>> tetrahedron.reset()
```

corners

Get the corners of the Polyhedron.

The method vertices is an alias for corners.

See also:

array_form (page 217), *cyclic_form* (page 217)

Examples

```
>>> from diofant.combinatorics import Polyhedron
>>> from diofant.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

cyclic_form

Return the indices of the corners in cyclic notation.

The indices are given relative to the original position of corners.

See also:

corners (page 217), *array_form* (page 217)

edges

Given the faces of the polyhedra we can get the edges.

Examples

```
>>> from diofant.combinatorics import Polyhedron
>>> from diofant.abc import a, b, c
>>> corners = (a, b, c)
>>> faces = [(0, 1, 2)]
>>> Polyhedron(corners, faces).edges
{(0, 1), (0, 2), (1, 2)}
```

faces

Get the faces of the Polyhedron.

pgroup

Get the permutations of the Polyhedron.

reset()

Return corners to their original positions.

Examples

```
>>> from diofant.combinatorics.polyhedron import tetrahedron as T
>>> T.corners
(0, 1, 2, 3)
>>> T.rotate(0)
>>> T.corners
(0, 2, 3, 1)
>>> T.reset()
>>> T.corners
(0, 1, 2, 3)
```

rotate(*perm*)

Apply a permutation to the polyhedron *in place*. The permutation may be given as a Permutation instance or an integer indicating which permutation from pgroup of the Polyhedron should be applied.

This is an operation that is analogous to rotation about an axis by a fixed increment.

Notes

When a Permutation is applied, no check is done to see if that is a valid permutation for the Polyhedron. For example, a cube could be given a permutation which effectively swaps only 2 vertices. A valid permutation (that rotates the object in a physical way) will be obtained if one only uses permutations from the pgroup of the Polyhedron. On the other hand, allowing arbitrary rotations (applications of permutations) gives a way to follow named elements rather than indices since Polyhedron allows vertices to be named while Permutation works only with indices.

Examples

```
>>> from diofant.combinatorics import Polyhedron, Permutation
>>> from diofant.combinatorics.polyhedron import cube
>>> cube.corners
(0, 1, 2, 3, 4, 5, 6, 7)
>>> cube.rotate(0)
>>> cube.corners
(1, 2, 3, 0, 5, 6, 7, 4)
```

A non-physical “rotation” that is not prohibited by this method:

```
>>> cube.reset()
>>> cube.rotate(Permutation([[1, 2]], size=8))
>>> cube.corners
(0, 2, 1, 3, 4, 5, 6, 7)
```

Polyhedron can be used to follow elements of set that are identified by letters instead of integers:

```
>>> shadow = h5 = Polyhedron(list('abcde'))
>>> p = Permutation([3, 0, 1, 2, 4])
>>> h5.rotate(p)
>>> h5.corners
(d, a, b, c, e)
>>> _ == shadow.corners
True
>>> copy = h5.copy()
>>> h5.rotate(p)
>>> h5.corners == copy.corners
False
```

size

Get the number of corners of the Polyhedron.

vertices

Get the corners of the Polyhedron.

The method vertices is an alias for corners.

See also:

[array_form](#) (page 217), [cyclic_form](#) (page 217)

Examples

```
>>> from diofant.combinatorics import Polyhedron
>>> from diofant.abc import a, b, c, d
>>> p = Polyhedron(list('abcd'))
>>> p.corners == p.vertices == (a, b, c, d)
True
```

3.2.5 Prufer Sequences

class diofant.combinatorics.prufer.Prufer

The Prufer correspondence is an algorithm that describes the bijection between labeled

trees and the Prufer code. A Prufer code of a labeled tree is unique up to isomorphism and has a length of $n - 2$.

Prufer sequences were first used by Heinz Prufer to give a proof of Cayley's formula.

References

[R19] (page 1260)

static edges(*runs)

Return a list of edges and the number of nodes from the given runs that connect nodes in an integer-labelled tree.

All node numbers will be shifted so that the minimum node is 0. It is not a problem if edges are repeated in the runs; only unique edges are returned. There is no assumption made about what the range of the node labels should be, but all nodes from the smallest through the largest must be present.

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer.edges([1, 2, 3], [2, 4, 5]) # a T
([[0, 1], [1, 2], [1, 3], [3, 4]], 5)
```

Duplicate edges are removed:

```
>>> Prufer.edges([0, 1, 2, 3], [1, 4, 5], [1, 4, 6]) # a K
([[0, 1], [1, 2], [1, 4], [2, 3], [4, 5], [4, 6]], 7)
```

next(delta=1)

Generates the Prufer sequence that is delta beyond the current one.

See also:

[prufer_rank](#) (page 221), [rank](#) (page 222), [prev](#) (page 221), [size](#) (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> b = a.next(1) # == a.next()
>>> b.tree_repr
[[0, 2], [0, 1], [1, 3]]
>>> b.rank
1
```

nodes

Returns the number of nodes in the tree.

Examples


```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).nodes
6
>>> Prufer([1, 0, 0]).nodes
5
```

prev(delta=1)

Generates the Prufer sequence that is -delta before the current one.

See also:

[prufer_rank](#) (page 221), [rank](#) (page 222), [next](#) (page 220), [size](#) (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [1, 2], [2, 3], [1, 4]])
>>> a.rank
36
>>> b = a.prev()
>>> b
Prufer([1, 2, 0])
>>> b.rank
35
```

prufer_rank()

Computes the rank of a Prufer sequence.

See also:

[rank](#) (page 222), [next](#) (page 220), [prev](#) (page 221), [size](#) (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_rank()
0
```

prufer_repr

Returns Prufer sequence for the Prufer object.

This sequence is found by removing the highest numbered vertex, recording the node it was attached to, and continuing until only two vertices remain. The Prufer sequence is the list of recorded nodes.

See also:

[to_prufer](#) (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).prufer_repr
[3, 3, 3, 4]
```

```
>>> Prufer([1, 0, 0]).prufer_repr
[1, 0, 0]
```

rank

Returns the rank of the Prufer sequence.

See also:

[prufer_rank](#) (page 221), [next](#) (page 220), [prev](#) (page 221), [size](#) (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> p = Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]])
>>> p.rank
778
>>> p.next(1).rank
779
>>> p.prev().rank
777
```

size

Return the number of possible trees of this Prufer object.

See also:

[prufer_rank](#) (page 221), [rank](#) (page 222), [next](#) (page 220), [prev](#) (page 221)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer([0]*4).size == Prufer([6]*4).size == 1296
True
```

static to_prufer(*tree*, *n*)

Return the Prufer sequence for a tree given as a list of edges where *n* is the number of nodes in the tree.

See also:

[prufer_repr](#) (page 221) returns Prufer sequence of a Prufer object.

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> a = Prufer([[0, 1], [0, 2], [0, 3]])
>>> a.prufer_repr
[0, 0]
>>> Prufer.to_prufer([[0, 1], [0, 2], [0, 3]], 4)
[0, 0]
```

static to_tree(*prufer*)

Return the tree (as a list of edges) of the given Prufer sequence.

See also:

`tree_repr` (page 223) returns tree representation of a Prufer object.

References

- <https://hamberg.no/erlend/posts/2010-11-06-prufer-sequence-compact-tree-representation.html>

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> a = Prufer([0, 2], 4)
>>> a.tree_repr
[[0, 1], [0, 2], [2, 3]]
>>> Prufer.to_tree([0, 2])
[[0, 1], [0, 2], [2, 3]]
```

`tree_repr`

Returns the tree representation of the Prufer object.

See also:

`to_tree` (page 222)

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer([[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]).tree_repr
[[0, 3], [1, 3], [2, 3], [3, 4], [4, 5]]
>>> Prufer([1, 0, 0]).tree_repr
[[1, 2], [0, 1], [0, 3], [0, 4]]
```

classmethod `unrank(rank, n)`

Finds the unranked Prufer sequence.

Examples

```
>>> from diofant.combinatorics.prufer import Prufer
>>> Prufer.unrank(0, 4)
Prufer([0, 0])
```

3.2.6 Subsets

class `diofant.combinatorics.subsets.Subset`

Represents a basic subset object.

We generate subsets using essentially two techniques, binary enumeration and lexicographic enumeration. The `Subset` class takes two arguments, the first one describes the initial subset to consider and the second describes the superset.

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a.prev_binary().subset
['c']
```

classmethod `bitlist_from_subset(subset, superset)`
Gets the bitlist corresponding to a subset.

See also:

[subset_from_bitlist](#) (page 228)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> Subset.bitlist_from_subset(['c', 'd'], ['a', 'b', 'c', 'd'])
'0011'
```

cardinality

Returns the number of all possible subsets.

See also:

[subset](#) (page 228), [superset](#) (page 229), [size](#) (page 227), [superset_size](#) (page 229)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.cardinality
16
```

iterate_binary(k)

This is a helper function. It iterates over the binary subsets by k steps. This variable can be both positive or negative.

See also:

[next_binary](#) (page 225), [prev_binary](#) (page 226)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(-2).subset
['d']
>>> a = Subset(['a', 'b', 'c'], ['a', 'b', 'c', 'd'])
>>> a.iterate_binary(2).subset
[]
```

iterate_graycode(*k*)

Helper function used for `prev_gray` and `next_gray`. It performs *k* step overs to get the respective Gray codes.

See also:

[next_gray](#) (page 225), [prev_gray](#) (page 226)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.iterate_graycode(3).subset
[1, 4]
>>> a.iterate_graycode(-2).subset
[1, 2, 4]
```

next_binary()

Generates the next binary ordered subset.

See also:

[prev_binary](#) (page 226), [iterate_binary](#) (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
['b']
>>> a = Subset(['a', 'b', 'c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_binary().subset
[]
```

next_gray()

Generates the next Gray code ordered subset.

See also:

[iterate_graycode](#) (page 224), [prev_gray](#) (page 226)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([1, 2, 3], [1, 2, 3, 4])
>>> a.next_gray().subset
[1, 3]
```

next_lexicographic()

Generates the next lexicographically ordered subset.

See also:

[prev_lexicographic](#) (page 226)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
['d']
>>> a = Subset(['d'], ['a', 'b', 'c', 'd'])
>>> a.next_lexicographic().subset
[]
```

prev_binary()

Generates the previous binary ordered subset.

See also:

[next_binary](#) (page 225), [iterate_binary](#) (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['a', 'b', 'c', 'd']
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.prev_binary().subset
['c']
```

prev_gray()

Generates the previous Gray code ordered subset.

See also:

[iterate_graycode](#) (page 224), [next_gray](#) (page 225)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([2, 3, 4], [1, 2, 3, 4, 5])
>>> a.prev_gray().subset
[2, 3, 4, 5]
```

prev_lexicographic()

Generates the previous lexicographically ordered subset.

See also:

[next_lexicographic](#) (page 225)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['d']
```

```
>>> a = Subset(['c','d'], ['a', 'b', 'c', 'd'])
>>> a.prev_lexicographic().subset
['c']
```

rank_binary

Computes the binary ordered rank.

See also:

[iterate_binary](#) (page 224), [unrank_binary](#) (page 229)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset([], ['a','b','c','d'])
>>> a.rank_binary
0
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_binary
3
```

rank_gray

Computes the Gray code ranking of the subset.

See also:

[iterate_graycode](#) (page 224), [unrank_gray](#) (page 229)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_gray
2
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_gray
27
```

rank_lexicographic

Computes the lexicographic ranking of the subset.

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.rank_lexicographic
14
>>> a = Subset([2, 4, 5], [1, 2, 3, 4, 5, 6])
>>> a.rank_lexicographic
43
```

size

Gets the size of the subset.

See also:

subset (page 228), *superset* (page 229), *superset_size* (page 229), *cardinality* (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.size
2
```

subset

Gets the subset represented by the current instance.

See also:

superset (page 229), *size* (page 227), *superset_size* (page 229), *cardinality* (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.subset
['c', 'd']
```

classmethod subset_from_bitlist(*super_set*, *bitlist*)

Gets the subset defined by the bitlist.

See also:

bitlist_from_subset (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> Subset.subset_from_bitlist(['a', 'b', 'c', 'd'], '0011').subset
['c', 'd']
```

classmethod subset_indices(*subset*, *superset*)

Return indices of subset in superset in a list; the list is empty if all elements of subset are not in superset.

Examples

```
>>> from diofant.combinatorics import Subset
>>> superset = [1, 3, 2, 5, 4]
>>> Subset.subset_indices([3, 2, 1], superset)
[1, 2, 0]
>>> Subset.subset_indices([1, 6], superset)
[]
```



```
>>> Subset.subset_indices([], superset)
[]
```

superset

Gets the superset of the subset.

See also:

subset (page 228), *size* (page 227), *superset_size* (page 229), *cardinality* (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset
['a', 'b', 'c', 'd']
```

superset_size

Returns the size of the superset.

See also:

subset (page 228), *superset* (page 229), *size* (page 227), *cardinality* (page 224)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> a = Subset(['c', 'd'], ['a', 'b', 'c', 'd'])
>>> a.superset_size
4
```

classmethod unrank_binary(*rank*, *superset*)

Gets the binary ordered subset of the specified rank.

See also:

iterate_binary (page 224), *rank_binary* (page 227)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> Subset.unrank_binary(4, ['a', 'b', 'c', 'd']).subset
['b']
```

classmethod unrank_gray(*rank*, *superset*)

Gets the Gray code ordered subset of the specified rank.

See also:

iterate_graycode (page 224), *rank_gray* (page 227)

Examples

```
>>> from diofant.combinatorics.subsets import Subset
>>> Subset.unrank_gray(4, ['a', 'b', 'c']).subset
['a', 'b']
>>> Subset.unrank_gray(0, ['a', 'b', 'c']).subset
[]
```

`subsets.ksubsets(superset, k)`

Finds the subsets of size *k* in lexicographic order.

This uses the `itertools` generator.

See also:

[Subset](#) (page 223)

Examples

```
>>> from diofant.combinatorics.subsets import ksubsets
>>> list(ksubsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
>>> list(ksubsets([1, 2, 3, 4, 5], 2))
[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]
```

3.2.7 Gray Code

class `diofant.combinatorics.graycode.GrayCode`

A Gray code is essentially a Hamiltonian walk on a *n*-dimensional cube with edge length of one. The vertices of the cube are represented by vectors whose values are binary. The Hamilton walk visits each vertex exactly once. The Gray code for a 3d cube is ['000', '100', '110', '010', '011', '111', '101', '001'].

A Gray code solves the problem of sequentially generating all possible subsets of *n* objects in such a way that each subset is obtained from the previous one by either deleting or adding a single object. In the above example, 1 indicates that the object is present, and 0 indicates that its absent.

Gray codes have applications in statistics as well when we want to compute various statistics related to subsets in an efficient manner.

References

[R4] (page 1260), [R5] (page 1260)

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
```

```
>>> a = GrayCode(4)
>>> list(a.generate_gray())
['0000', '0001', '0011', '0010', '0110', '0111', '0101', '0100',      '1100', '1101',
↵', '1111', '1110', '1010', '1011', '1001', '1000']
```

current

Returns the currently referenced Gray code as a bit string.

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> GrayCode(3, start='100').current
'100'
```

generate_gray(hints)**

Generates the sequence of bit vectors of a Gray Code.

[1] Knuth, D. (2011). The Art of Computer Programming, Vol 4, Addison Wesley

See also:

[skip](#) (page 232)

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(start='011'))
['011', '010', '110', '111', '101', '100']
>>> list(a.generate_gray(rank=4))
['110', '111', '101', '100']
```

n

Returns the dimension of the Gray code.

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(5)
>>> a.n
5
```

next(delta=1)

Returns the Gray code a distance delta (default = 1) from the current value in canonical order.

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3, start='110')
>>> a.next().current
'111'
>>> a.next(-1).current
'010'
```

rank

Ranks the Gray code.

A ranking algorithm determines the position (or rank) of a combinatorial object among all the objects w.r.t. a given order. For example, the 4 bit binary reflected Gray code (BRGC) '0101' has a rank of 6 as it appears in the 6th position in the canonical ordering of the family of 4 bit Gray codes.

See also:

[unrank](#) (page 233)

References

[R6] (page 1260)

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> list(a.generate_gray())
['000', '001', '011', '010', '110', '111', '101', '100']
>>> GrayCode(3, start='100').rank
7
>>> GrayCode(3, rank=7).current
'100'
```

selections

Returns the number of bit vectors in the Gray code.

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> a.selections
8
```

skip()

Skips the bit generation.

See also:

[generate_gray](#) (page 231)

Examples

```
>>> from diofant.combinatorics.graycode import GrayCode
>>> a = GrayCode(3)
>>> for i in a.generate_gray():
...     if i == '010':
...         a.skip()
...     print(i)
...
000
001
011
010
111
101
100
```

classmethod unrank(*n*, *rank*)

Unranks an *n*-bit sized Gray code of rank *k*. This method exists so that a derivative `GrayCode` class can define its own code of a given rank.

The string here is generated in reverse order to allow for tail-call optimization.

See also:

[rank](#) (page 232)

Examples

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

graycode.random_bitstring(*n*)

Generates a random bitlist of length *n*.

Examples

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

graycode.gray_to_bin(*bin_list*)

Convert from Gray coding to binary coding.

We assume big endian encoding.

See also:

[bin_to_gray](#) (page 234)

Examples

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

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

Convert from binary coding to gray coding.

We assume big endian encoding.

See also:

[gray_to_bin](#) (page 233)

Examples

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

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

Gets the subset defined by the bitstring.

See also:

[graycode_subsets](#) (page 234)

Examples

```
>>> from diofant.combinatorics.graycode import get_subset_from_bitstring
>>> get_subset_from_bitstring(['a', 'b', 'c', 'd'], '0011')
['c', 'd']
>>> get_subset_from_bitstring(['c', 'a', 'c', 'c'], '1100')
['c', 'a']
```

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

Generates the subsets as enumerated by a Gray code.

See also:

[get_subset_from_bitstring](#) (page 234)

Examples

```
>>> from diofant.combinatorics.graycode import graycode_subsets
>>> list(graycode_subsets(['a', 'b', 'c']))
[[], ['c'], ['b', 'c'], ['b'], ['a', 'b'], ['a', 'b', 'c'], ['a', 'c'], ['a']]
>>> list(graycode_subsets(['a', 'b', 'c', 'c']))
[[], ['c'], ['c', 'c'], ['c'], ['b', 'c'], ['b', 'c', 'c'], ['b', 'c'], ['b'],
↪ ['a', 'b'], ['a', 'b', 'c'], ['a', 'b', 'c', 'c'], ['a', 'b', 'c'], ['a',
↪ 'c'], ['a', 'c', 'c'], ['a', 'c'], ['a']]
```

3.2.8 Named Groups

`diofant.combinatorics.named_groups.SymmetricGroup(n)`

Generates the symmetric group on n elements as a permutation group.

The generators taken are the n -cycle $(0\ 1\ 2\ \dots\ n-1)$ and the transposition $(0\ 1)$ (in cycle notation). (See [1]). After the group is generated, some of its basic properties are set.

See also:

CyclicGroup (page 235), *DihedralGroup* (page 235), *AlternatingGroup* (page 236)

References

[1] http://en.wikipedia.org/wiki/Symmetric_group#Generators_and_relations

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> G = SymmetricGroup(4)
>>> G.is_group
True
>>> G.order()
24
>>> list(G.generate_schreier_sims(af=True))
[[0, 1, 2, 3], [1, 2, 3, 0], [2, 3, 0, 1], [3, 1, 2, 0], [0, 2, 3, 1],
 [1, 3, 0, 2], [2, 0, 1, 3], [3, 2, 0, 1], [0, 3, 1, 2], [1, 0, 2, 3],
 [2, 1, 3, 0], [3, 0, 1, 2], [0, 1, 3, 2], [1, 2, 0, 3], [2, 3, 1, 0],
 [3, 1, 0, 2], [0, 2, 1, 3], [1, 3, 2, 0], [2, 0, 3, 1], [3, 2, 1, 0],
 [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3], [3, 0, 2, 1]]
```

`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.

See also:

SymmetricGroup (page 235), *DihedralGroup* (page 235), *AlternatingGroup* (page 236)

Examples

```
>>> from diofant.combinatorics.named_groups import CyclicGroup
>>> 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]]
```

`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 rotation. It is easy to see that these satisfy $a^{**n} = b^{**2} = 1$ and $bab = \sim a$ so they indeed generate D_n (See [1]). After the group is generated, some of its basic properties are set.

See also:

[SymmetricGroup](#) (page 235), [CyclicGroup](#) (page 235), [AlternatingGroup](#) (page 236)

References

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

Examples

```
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> G = DihedralGroup(5)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> [perm.cyclic_form for perm in a]
[[[]], [[0, 1, 2, 3, 4]], [[0, 2, 4, 1, 3]],
 [[0, 3, 1, 4, 2]], [[0, 4, 3, 2, 1]], [[0, 4], [1, 3]],
 [[1, 4], [2, 3]], [[0, 1], [2, 4]], [[0, 2], [3, 4]],
 [[0, 3], [1, 2]]]
```

`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.

See also:

[SymmetricGroup](#) (page 235), [CyclicGroup](#) (page 235), [DihedralGroup](#) (page 235)

References

[1] Armstrong, M. "Groups and Symmetry"

Examples

```
>>> from diofant.combinatorics.named_groups import AlternatingGroup
>>> G = AlternatingGroup(4)
>>> G.is_group
True
>>> a = list(G.generate_dimino())
>>> len(a)
12
```



```
>>> all(perm.is_even for perm in a)
True
```

`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.

See also:

[*diofant.combinatorics.group_constructs.DirectProduct*](#) (page 243)

References

[R7] (page 1260)

Examples

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

3.2.9 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.

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

[R20] (page 1260)

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.util import _base_ordering
>>> S = SymmetricGroup(4)
>>> S.schreier_sims()
>>> _base_ordering(S.base, S.degree)
[0, 1, 2, 3]
```

`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`.

See also:

[`diofant.combinatorics.perm_groups.PermutationGroup.is_alt_sym`](#) (page 202)

Examples

```
>>> from diofant.combinatorics.util import _check_cycles_alt_sym
>>> from diofant.combinatorics.permutations import Permutation
>>> a = Permutation([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [11, 12]])
>>> _check_cycles_alt_sym(a)
False
>>> b = Permutation([[0, 1, 2, 3, 4, 5, 6], [7, 8, 9, 10]])
>>> _check_cycles_alt_sym(b)
True
```

`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, ..., n-1}’

“`gens`” - a list of elements of a permutation group of degree ‘ n ’.

Returns List of length k , where k is

the length of `base`. The i -th entry contains those elements in `gens` which fix the first i elements of `base` (so that the 0-th entry is equal to `gens` itself). If no element fixes the first i elements of `base`, the i -th element is set to a list containing the identity element.

See also:

[`_strong_gens_from_distr`](#) (page 242), [`_orbits_transversals_from_bsgs`](#) (page 240), [`_handle_precomputed_bsgs`](#) (page 239)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> from diofant.combinatorics.util import _distribute_gens_by_base
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> D.strong_gens
[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)]
>>> D.base
[0, 1]
>>> _distribute_gens_by_base(D.base, D.strong_gens)
[[Permutation(0, 1, 2), Permutation(0, 2), Permutation(1, 2)],
 [Permutation(1, 2)]]
```

```
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.

See also:

[_orbits_transversals_from_bsgs](#) (page 240), [_distribute_gens_by_base](#) (page 238)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.named_groups import DihedralGroup
>>> from diofant.combinatorics.util import _handle_precomputed_bsgs
>>> D = DihedralGroup(3)
>>> D.schreier_sims()
>>> _handle_precomputed_bsgs(D.base, D.strong_gens,
```

```

... basic_orbits=D.basic_orbits)
({0: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2)},
 {1: Permutation(2), 2: Permutation(1, 2)}],
 [[0, 1, 2], [1, 2]], [[Permutation(0, 1, 2),
                        Permutation(0, 2),
                        Permutation(1, 2)],
                       [Permutation(1, 2)]]

```

`diofant.combinatorics.util._orbits_transversals_from_bsgs`(*base*,
strong_gens_distr,
transversals_only=False)

Compute basic orbits and transversals from a base and strong generating set.

The generators are provided as distributed across the basic stabilizers. If the optional argument `transversals_only` is set to `True`, only the transversals are returned.

Parameters “base” - the base

“strong_gens_distr” - strong generators distributed by membership in basic

stabilizers

“transversals_only” - a flag switching between returning only the transversals/ both orbits and transversals

See also:

[`_distribute_gens_by_base`](#) (page 238), [`_handle_precomputed_bsgs`](#) (page 239)

Examples

```

>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.util import _orbits_transversals_from_bsgs
>>> from diofant.combinatorics.util import (_orbits_transversals_from_bsgs,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> strong_gens_distr = _distribute_gens_by_base(S.base, S.strong_gens)
>>> _orbits_transversals_from_bsgs(S.base, strong_gens_distr)
([[0, 1, 2], [1, 2]],
 [{0: Permutation(2), 1: Permutation(0, 1, 2), 2: Permutation(0, 2, 1)},
 {1: Permutation(2), 2: Permutation(1, 2)}])

```

`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`.

Notes

This procedure is outlined in [1],p.95.

References

[1] Holt, D., Eick, B., O'Brien, E. "Handbook of computational group theory"

Examples

```
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.util import _remove_gens
>>> 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
```

`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"`.

See also:

`diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims` (page 211),
`diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims_random`
 (page 212)

Notes

The algorithm is described in [1],pp.89-90. The reason for returning both the current state of the element being decomposed and the level at which the sifting ends is that they provide important information for the randomized version of the Schreier-Sims algorithm.

References

[1] Holt, D., Eick, B., O’Brien, E. “Handbook of computational group theory”

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.util import _strip
>>> S = SymmetricGroup(5)
>>> S.schreier_sims()
>>> g = Permutation([0, 2, 3, 1, 4])
>>> _strip(g, S.base, S.basic_orbits, S.basic_transversals)
(Permutation(4), 5)
```

`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

See also:

`_distribute_gens_by_base` (page 238)

Examples

```
>>> from diofant.combinatorics import Permutation
>>> Permutation.print_cyclic = True
>>> from diofant.combinatorics.named_groups import SymmetricGroup
>>> from diofant.combinatorics.util import (_strong_gens_from_distr,
... _distribute_gens_by_base)
>>> S = SymmetricGroup(3)
>>> S.schreier_sims()
>>> S.strong_gens
```

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

3.2.10 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 190) procedure for taking the direct product of two permutation groups, but the idea of shifting the generators is realized in the case of an arbitrary number of groups. A call to `DirectProduct(G1, G2, ..., Gn)` is generally expected to be faster than a call to `G1*G2*...*Gn` (and thus the need for this algorithm).

See also:

`diofant.combinatorics.perm_groups.PermutationGroup.__mul__` (page 190)

Examples

```
>>> from diofant.combinatorics.group_constructs import DirectProduct
>>> from diofant.combinatorics.named_groups import CyclicGroup
>>> C = CyclicGroup(4)
>>> G = DirectProduct(C, C, C)
>>> G.order()
64
```

3.2.11 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

```
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.testutil import _cmp_perm_lists
>>> a = Permutation([0, 2, 3, 4, 1])
>>> b = Permutation([1, 2, 0, 4, 3])
>>> c = Permutation([3, 4, 0, 1, 2])
>>> ls1 = [a, b, c]
>>> ls2 = [b, c, a]
>>> _cmp_perm_lists(ls1, ls2)
True
```

`diofant.combinatorics.testutil._naive_list_centralizer(self, other, af=False)`

`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.

See also:

[`diofant.combinatorics.perm_groups.PermutationGroup.schreier_sims`](#) (page 211)

Examples

```
>>> from diofant.combinatorics.named_groups import AlternatingGroup
>>> from diofant.combinatorics.testutil import _verify_bsgs
>>> A = AlternatingGroup(4)
>>> A.schreier_sims()
>>> _verify_bsgs(A, A.base, A.strong_gens)
True
```

`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`

See also:

[`_naive_list_centralizer`](#) (page 243), [`diofant.combinatorics.perm_groups.PermutationGroup.ce`](#) (page 195), [`_cmp_perm_lists`](#) (page 243)

Examples

```
>>> from diofant.combinatorics.named_groups import (SymmetricGroup,
... AlternatingGroup)
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.testutil import _verify_centralizer
>>> S = SymmetricGroup(5)
>>> A = AlternatingGroup(5)
>>> centr = PermutationGroup([Permutation([0, 1, 2, 3, 4])])
>>> _verify_centralizer(S, A, centr)
True
```

`diofant.combinatorics.testutil._verify_normal_closure(group, arg, closure=None)`

3.2.12 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 [d0, -d0, d1, -d1, ...]

msym : symmetry of the metric(s)

it can be an integer or a list; in the first case it is the symmetry of the dummy index metric; in the second case it is the list of the symmetries of the index metric for each type

v : list, (base_i, gens_i, n_i, sym_i) for tensors of type *i*

base_i, gens_i : BSGS for tensors of this type.

The BSGS should have minimal base under lexicographic ordering; if not, an attempt is made to get the minimal BSGS; in case of failure, canonicalize_naive is used, which is much slower.

n_i : number of tensors of type *i*.

sym_i : symmetry under exchange of component tensors of type *i*.

Both for msym and sym_i the cases are

- None no symmetry
- 0 commuting
- 1 anticommuting

Returns 0 if the tensor is zero, else return the array form of the permutation representing the canonical form of the tensor.

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

$$T = A_{d_0 d_1} * B^{d_0}_{d_2} * B^{d_2 d_1}$$

ord = [d0, -d0, d1, -d1, d2, -d2] order of the indices

$$g = [1, 3, 0, 5, 4, 2, 6, 7]$$

$$T_c = 0$$

```
>>> from diofant.combinatorics.tensor_can import get_symmetric_group_sgs,
↳ canonicalize, bsgs_direct_product
>>> from diofant.combinatorics import Permutation
>>> base2a, gens2a = get_symmetric_group_sgs(2, 1)
>>> t0 = (base2a, gens2a, 1, 0)
>>> t1 = (base2a, gens2a, 2, 0)
>>> g = Permutation([1, 3, 0, 5, 4, 2, 6, 7])
>>> canonicalize(g, range(6), 0, t0, t1)
0
```

same as above, but with B_{ab} anticommuting

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

can = [0,2,1,4,3,5,7,6]

```
>>> t1 = (base2a, gens2a, 2, 1)
>>> canonicalize(g, range(6), 0, t0, t1)
[0, 2, 1, 4, 3, 5, 7, 6]
```

two types of indices $[a, b, c, d, e, f]$ and $[m, n]$, in this order, both with commuting metric

f^{abc} antisymmetric, commuting

A_{ma} no symmetry, commuting

$$T = f^c_{da} * f^f_{eb} * A_m^d * A^{mb} * A_n^a * A^{ne}$$

ord = [c,f,a,-a,b,-b,d,-d,e,-e,m,-m,n,-n]

g = [0,7,3, 1,9,5, 11,6, 10,4, 13,2, 12,8, 14,15]

The canonical tensor is $T_c = -f^{cab} * f^{fde} * A^m_a * A_{md} * A^n_b * A_{ne}$

can = [0,2,4, 1,6,8, 10,3, 11,7, 12,5, 13,9, 15,14]

```
>>> base_f, gens_f = get_symmetric_group_sgs(3, 1)
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base_A, gens_A = bsgs_direct_product(base1, gens1, base1, gens1)
>>> t0 = (base_f, gens_f, 2, 0)
>>> t1 = (base_A, gens_A, 4, 0)
>>> dummies = [range(2, 10), range(10, 14)]
>>> g = Permutation([0, 7, 3, 1, 9, 5, 11, 6, 10, 4, 13, 2, 12, 8, 14, 15])
>>> canonicalize(g, dummies, [0, 0], t0, t1)
[0, 2, 4, 1, 6, 8, 10, 3, 11, 7, 12, 5, 13, 9, 15, 14]
```

`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 $n = len(ind)$; g has size $n + 2$, the last two indices for the sign of the tensor (trick introduced in [4]).

A slot symmetry transformation s is a permutation acting on the slots $t \rightarrow T(ind[(g * s)[0]], \dots, ind[(g * s)[n - 1]])$

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

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

$$g \rightarrow d * g * s, \text{ so it belongs to the coset } D * g * S.$$

Let us explain the conventions by an example.

Given a tensor T^{d3d2d1}_{d1d2d3} **with the slot symmetries** $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$
 $T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$

and symmetric metric, find the tensor equivalent to it which is the lowest under the ordering of indices: lexicographic ordering $d1, d2, d3$ then and contravariant index before covariant index; that is the canonical form of the tensor.

The canonical form is $-T^{d1d2d3}_{d1d2d3}$ obtained using $T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$.

To convert this problem in the input for this function, use the following labelling of the index names (- for covariant for short) $d1, -d1, d2, -d2, d3, -d3$

T^{d3d2d1}_{d1d2d3} corresponds to $g = [4, 2, 0, 1, 3, 5, 6, 7]$ where the last two indices are for the sign

$sgens = [Permutation(0, 2)(6, 7), Permutation(0, 4)(6, 7)]$

$sgens[0]$ is the slot symmetry $-(0, 2) T^{a0a1a2a3a4a5} = -T^{a2a1a0a3a4a5}$

sgens[1] is the slot symmetry $-(0, 4) T^{a0a1a2a3a4a5} = -T^{a4a1a2a3a0a5}$

The dummy symmetry group D is generated by the strong base generators $[(0, 1), (2, 3), (4, 5), (0, 1)(2, 3), (2, 3)(4, 5)]$

The dummy symmetry acts from the left $d = [1, 0, 2, 3, 4, 5, 6, 7]$ exchange $d1- > -d1$
 $T^{d3d2d1}_{d1d2d3} == T^{d3d2}_{d1}^{d1}_{d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7]- > [4, 2, 1, 0, 3, 5, 6, 7] =_a f_rmul(d, g)$ which differs from $_a f_rmul(g, d)$.

The slot symmetry acts from the right $s = [2, 1, 0, 3, 4, 5, 7, 6]$ exchanges slots 0 and 2 and changes sign $T^{d3d2d1}_{d1d2d3} == -T^{d1d2d3}_{d1d2d3}$

$g = [4, 2, 0, 1, 3, 5, 6, 7]- > [0, 2, 4, 1, 3, 5, 7, 6] =_a f_rmul(g, s)$

Example in which the tensor is zero, same slot symmetries as above: $T^{d3}_{d1,d2}^{d1}_{d3}^{d2}$

$= -T^{d3}_{d1,d3}^{d1}_{d2}^{d2}$ under slot symmetry $-(2, 4)$;

$= T_{d3d1}^{d3d1}_{d2}^{d2}$ under slot symmetry $-(0, 2)$;

$= T^{d3}_{d1d3}^{d1}_{d2}^{d2}$ symmetric metric;

$= 0$ since two of these lines have tensors differ only for the sign.

The double coset $D * g * S$ consists of permutations $h = d * g * s$ corresponding to equivalent tensors; if there are two h which are the same apart from the sign, return zero; otherwise choose as representative the tensor with indices ordered lexicographically according to $[d1, -d1, d2, -d2, d3, -d3]$ that is $rep = \min(D * g * S) = \min([d * g * s \text{ for } d \text{ in } D \text{ for } s \text{ in } S])$

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 $j < i$ $d_i * g * s_i * S_{b_0, \dots, b_{i-1}} * b_j = D_{p_0, \dots, p_{i-1}} * p_j$ so that for dx in $D_{p_0, \dots, p_{i-1}}$ 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 $s_i * sx * b_j = J, dx * d_i * g * J = p_j$ $sx * b_j = s_i * ** - 1 * J; sx = \text{trace}(s_i * ** - 1, S_{b_0, \dots, b_{i-1}}) dx * ** - 1 * p_j = d_i * g * J; dx = \text{trace}(d_i * g * J, D_{p_0, \dots, p_{i-1}})$

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

$h_n * b_j = p_j$ for all j , so that h_n is the solution.

Add the found (s, d, h) to TAB1.

At the end of the iteration sort TAB1 with respect to the h ; if there are two consecutive h in TAB1 which differ only for the sign, the tensor is zero, so return 0; if there are two consecutive h which are equal, keep only one.

Then stabilize the slot generators under i and the dummy generators under p_i .

Assign $TAB = TAB1$ at the end of the iteration step.

At the end TAB contains a unique (s, d, h) , since all the slots of the tensor h have been fixed to have the minimum value according to the symmetries. The algorithm returns h .

It is important that the slot BSGS has lexicographic minimal base, otherwise there is an i which does not belong to the slot base for which p_i is fixed by the dummy symmetry only, while i is not invariant from the slot stabilizer, so p_i is not in general the minimal value.

This algorithm differs slightly from the original algorithm [3]: the canonical form is minimal lexicographically, and the BSGS has minimal base under lexicographic order. Equal tensors h are eliminated from TAB .

Examples

```
>>> from diofant.combinatorics.permutations import Permutation
>>> from diofant.combinatorics.perm_groups import PermutationGroup
>>> from diofant.combinatorics.tensor_can import double_coset_can_rep, get_
↳ transversals
>>> gens = [Permutation(x) for x in [[2, 1, 0, 3, 4, 5, 7, 6], [4, 1, 2, 3, 0, 5,
↳ 7, 6]]]
>>> base = [0, 2]
>>> g = Permutation([4, 2, 0, 1, 3, 5, 6, 7])
>>> transversals = get_transversals(base, gens)
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
[0, 1, 2, 3, 4, 5, 7, 6]
```

```
>>> g = Permutation([4, 1, 3, 0, 5, 2, 6, 7])
>>> double_coset_can_rep([list(range(6))], [0], base, gens, transversals, g)
0
```

`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

```
>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.tensor_can import get_symmetric_group_sgs
>>> Permutation.print_cyclic = True
>>> get_symmetric_group_sgs(3)
([0, 1], [Permutation(4)(0, 1), Permutation(4)(1, 2)])
```

`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

```
>>> from diofant.combinatorics import Permutation
>>> from diofant.combinatorics.tensor_can import (get_symmetric_group_sgs, bsgs_
↳direct_product)
>>> Permutation.print_cyclic = True
>>> base1, gens1 = get_symmetric_group_sgs(1)
>>> base2, gens2 = get_symmetric_group_sgs(2)
>>> bsgs_direct_product(base1, gens1, base2, gens2)
([1], [Permutation(4)(1, 2)])
```

3.3 Number Theory

3.3.1 Ntheory Class Reference

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

```
>>> 25 in sieve
False
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

extend(*n*)

Grow the sieve to cover all primes $\leq n$ (a real number).

Examples

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

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

extend_to_no(*i*)

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

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

`primerange(a, b)`

Generate all prime numbers in the range [a, b).

Examples

```
>>> from diofant import sieve
>>> print([i for i in sieve.primerange(7, 18)])
[7, 11, 13, 17]
```

`search(n)`

Return the indices i, j of the primes that bound n.

If n is prime then i == j.

Although n can be an expression, if ceiling cannot convert it to an integer then an error will be raised.

Examples

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

3.3.2 Ntheory Functions Reference

`diofant.ntheory.generate.prime(nth)`

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

See also:

[`diofant.ntheory.primetest.isprime` \(page 270\)](#) Test if n is prime

[`primerange` \(page 253\)](#) Generate all primes in a given range

[`primepi` \(page 252\)](#) Return the number of primes less than or equal to n

References

[R368] (page 1260)

Examples

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

`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 .

See also:

[`diofant.ntheory.primetest.isprime` \(page 270\)](#) Test if n is prime

[`primerange` \(page 253\)](#) Generate all primes in a given range

[`prime` \(page 251\)](#) Return the n th prime

Examples

```
>>> from diofant import primepi
>>> primepi(25)
9
```

`diofant.ntheory.generate.nextprime(n, ith=1)`

Return the i th prime greater than n .

i must be an integer.

See also:

[`prevprime` \(page 252\)](#) Return the largest prime smaller than n

[`primerange` \(page 253\)](#) Generate all primes in a given range

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from diofant import nextprime
>>> [(i, nextprime(i)) for i in range(10, 15)]
[(10, 11), (11, 13), (12, 13), (13, 17), (14, 17)]
>>> nextprime(2, ith=2) # the 2nd prime after 2
5
```

`diofant.ntheory.generate.prevprime(n)`

Return the largest prime smaller than n .

See also:

[`nextprime` \(page 252\)](#) Return the i th prime greater than n

[`primerange` \(page 253\)](#) Generates all primes in a given range

Notes

Potential primes are located at $6*j \pm 1$. This property is used during searching.

```
>>> from diofant import prevprime
>>> [(i, prevprime(i)) for i in range(10, 15)]
[(10, 7), (11, 7), (12, 11), (13, 11), (14, 13)]
```

`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.

See also:

`nextprime` (page 252) Return the *i*th prime greater than *n*

`prevprime` (page 252) Return the largest prime smaller than *n*

`randprime` (page 254) Returns a random prime in a given range

`primorial` (page 254) Returns the product of primes based on condition

`Sieve.primerange` (page 251) return range from already computed primes or extend the sieve to contain the requested range.

Notes

Some famous conjectures about the occurrence of primes in a given range are [\[R369\]](#) (page 1260):

- **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 $\log(n)$ [\[R370\]](#) (page 1260); 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

[\[R369\]](#) (page 1260), [\[R370\]](#) (page 1260)

Examples

```
>>> from diofant import primerange, sieve
>>> print([i for i in primerange(1, 30)])
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

The Sieve method, `primerange`, is generally faster but it will occupy more memory as the sieve stores values. The default instance of Sieve, named `sieve`, can be used:

```
>>> list(sieve.primerange(1, 30))
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

`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 $a > 1$.

See also:

[*primerange* \(page 253\)](#) Generate all primes in a given range

References

[R371] (page 1261)

Examples

```
>>> from diofant import randprime, isprime
>>> randprime(1, 30)
13
>>> isprime(randprime(1, 30))
True
```

`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`).

```
>>> from diofant.ntheory.generate import primorial, randprime, primerange
>>> from diofant import factorint, Mul, primefactors, sqrt
>>> primorial(4) # the first 4 primes are 2, 3, 5, 7
210
>>> primorial(4, nth=False) # primes <= 4 are 2 and 3
6
>>> primorial(1)
2
>>> primorial(1, nth=False)
1
>>> primorial(sqrt(101), nth=False)
210
```

One can argue that the primes are infinite since if you take a set of primes and multiply them together (e.g. the primorial) and then add or subtract 1, the result cannot be divided by any of the original factors, hence either 1 or more new primes must divide this product of primes.

In this case, the number itself is a new prime:

```
>>> factorint(primorial(4) + 1)
{211: 1}
```

In this case two new primes are the factors:

```
>>> factorint(primorial(4) - 1) == {11: 1, 19: 1}
True
```

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 253)** Generate all primes in a given range

`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 (λ) and the length of terms before the cycle begins (μ); if `values` is `True` then the terms of the sequence will be returned instead. The sequence is started with value x_0 .

Note: more than the first $\lambda + \mu$ terms may be returned and this is the cost of cycle detection with Brent's method; there are, however, generally less terms calculated than would have been calculated if the proper ending point were determined, e.g. by using Floyd's method.

```
>>> from diofant.ntheory.generate import cycle_length
```

This will yield successive values of $i \leftarrow \text{func}(i)$:

```
>>> def iter(func, i):
...     while 1:
...         ii = func(i)
...         yield ii
...         i = ii
... 
```

A function is defined:

```
>>> func = lambda i: (i**2 + 1) % 51
```

and given a seed of 4 and the μ and λ terms calculated:

```
>>> next(cycle_length(func, 4))
(6, 2)
```

We can see what is meant by looking at the output:

```
>>> n = cycle_length(func, 4, values=True)
>>> list(ni for ni in n)
[17, 35, 2, 5, 26, 14, 44, 50, 2, 5, 26, 14]
```

There are 6 repeating values after the first 2.

If a sequence is suspected of being longer than you might wish, `nmax` can be used to exit early (and μ will be returned as `None`):

```
>>> next(cycle_length(func, 4, nmax = 4))
(4, None)
>>> [ni for ni in cycle_length(func, 4, nmax = 4, values=True)]
[17, 35, 2, 5]
```

References

[R372] (page 1261)

`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.

```
>>> from diofant.ntheory.factor_ import smoothness
>>> smoothness(2**7*3**2)
(3, 128)
>>> smoothness(2**4*13)
(13, 16)
>>> smoothness(2)
(2, 2)
```

See also:

`factorint` (page 261), `smoothness_p` (page 256)

`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 +/- 1 type factoring methods.

```
>>> from diofant.ntheory.factor_ import smoothness_p, factorint
>>> smoothness_p(10431, m=1)
(1, [(3, (2, 2, 4)), (19, (1, 5, 5)), (61, (1, 31, 31))])
>>> smoothness_p(10431)
(-1, [(3, (2, 2, 2)), (19, (1, 3, 9)), (61, (1, 5, 5))])
>>> smoothness_p(10431, power=1)
(-1, [(3, (2, 2, 2)), (61, (1, 5, 5)), (19, (1, 3, 9))])
```

If visual=True then an annotated string will be returned:

```
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

This string can also be generated directly from a factorization dictionary and vice versa:

```
>>> f = factorint(17*9)
>>> f == {3: 2, 17: 1}
True
>>> 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}
True
```

The table of the output logic is:

	Visual		
Input	True	False	other
dict	str	tuple	str
str	str	tuple	dict
tuple	str	tuple	str
n	str	tuple	tuple
mul	str	tuple	tuple

See also:

[factorint](#) (page 261), [smoothness](#) (page 256)

`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

```
>>> from diofant import trailing
>>> trailing(128)
7
>>> trailing(63)
0
```

`diofant.ntheory.factor_.multiplicity(p, n)`

Find the greatest integer m such that p^{**m} divides n .

Examples

```
>>> from diofant.ntheory import multiplicity
>>> from diofant.core.numbers import Rational as R
>>> [multiplicity(5, n) for n in [8, 5, 25, 125, 250]]
[0, 1, 2, 3, 3]
>>> multiplicity(3, R(1, 9))
-2
```

`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

```
>>> from diofant import perfect_power
>>> perfect_power(16)
(2, 4)
>>> perfect_power(16, big = False)
(4, 2)
```

```
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 a lead-up to some number and then loop around back to that number and begin to repeat the sequence, e.g. 1, 2, 3, 4, 5, 3, 4, 5 - this leader and loop look a bit like the Greek letter rho, and thus the name, 'rho'.

For a given function, very different leader-loop values can be obtained so it is a good idea to allow for retries:

```
>>> from diofant.ntheory.generate import cycle_length
>>> n = 16843009
>>> F = lambda x:(2048*pow(x, 2, n) + 32767) % n
>>> for s in range(5):
...     print('loop length = %4i; leader length = %3i' % next(cycle_length(F, s)))
...
loop length = 2489; leader length = 42
loop length = 78; leader length = 120
loop length = 1482; leader length = 99
loop length = 1482; leader length = 285
loop length = 1482; leader length = 100
```

Here is an explicit example where there is a two element leadup to a sequence of 3 numbers (11, 14, 4) that then repeat:

```
>>> x=2
>>> for i in range(9):
...     x=(x**2+12)%17
...     print(x)
...
16
13
11
14
4
11
14
4
11
>>> next(cycle_length(lambda x: (x**2+12)%17, 2))
(3, 2)
>>> list(cycle_length(lambda x: (x**2+12)%17, 2, values=True))
```

```
[16, 13, 11, 14, 4]
```

Instead of checking the differences of all generated values for a gcd with n , only the k th and $2*k$ th numbers are checked, e.g. 1st and 2nd, 2nd and 4th, 3rd and 6th until it has been detected that the loop has been traversed. Loops may be many thousands of steps long before rho finds a factor or reports failure. If `max_steps` is specified, the iteration is cancelled with a failure after the specified number of steps.

References

[R373] (page 1261)

Examples

```
>>> from diofant import pollard_rho
>>> n=16843009
>>> F=lambda x:(2048*pow(x,2,n) + 32767) % n
>>> pollard_rho(n, F=F)
257
```

Use the default setting with a bad value of a and no retries:

```
>>> pollard_rho(n, a=n-2, retries=0)
```

If `retries` is > 0 then perhaps the problem will correct itself when new values are generated for a :

```
>>> pollard_rho(n, a=n-2, retries=1)
257
```

`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 $\text{lcm}(1..B) = \text{reduce}(\text{ilcm}, \text{range}(2, B + 1))$.

A search is made for factors next to even numbers having a power smoothness less than B . Choosing a larger B increases the likelihood of finding a larger factor but takes longer. Whether a factor of n is found or not depends on a and the power smoothness of the even number just less than the factor p (hence the name $p - 1$).

Although some discussion of what constitutes a good a some descriptions are hard to interpret. At the modular.math site referenced below it is stated that if $\gcd(a^{**M} - 1, n) = N$ then $a^{**M} \% q^{**r}$ is 1 for every prime power divisor of N . But consider the following:

```
>>> from diofant.ntheory.factor_ import smoothness_p, pollard_pm1
>>> n=257*1009
>>> smoothness_p(n)
(-1, [(257, (1, 2, 256)), (1009, (1, 7, 16))])
```

So we should (and can) find a root with $B=16$:

```
>>> pollard_pm1(n, B=16, a=3)
1009
```

If we attempt to increase B to 256 we find that it doesn't work:

```
>>> pollard_pm1(n, B=256)
>>>
```

But if the value of a is changed we find that only multiples of 257 work, e.g.:

```
>>> pollard_pm1(n, B=256, a=257)
1009
```

Checking different a values shows that all the ones that didn't work had a gcd value not equal to n but equal to one of the factors:

```
>>> from diofant.core.numbers import ilcm, igcd
>>> from diofant import factorint, Pow
>>> M = 1
>>> for i in range(2, 256):
...     M = ilcm(M, i)
...
>>> {igcd(pow(a, M, n) - 1, n) for a in range(2, 256) if
...     igcd(pow(a, M, n) - 1, n) != n} == {1009}
True
```

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.

References

[R374] (page 1261), [R375] (page 1261), [R376] (page 1261)

Examples

With the default smoothness bound, this number can't be cracked:

```
>>> from diofant.ntheory import pollard_pm1, primefactors
>>> pollard_pm1(21477639576571)
```

Increasing the smoothness bound helps:

```
>>> pollard_pm1(21477639576571, B=2000)
4410317
```


Looking at the smoothness of the factors of this number we find:

```
>>> from diofant.utilities import flatten
>>> from diofant.ntheory.factor_ import smoothness_p, factorint
>>> print(smoothness_p(21477639576571, visual=1))
p**i=4410317**1 has p-1 B=1787, B-pow=1787
p**i=4869863**1 has p-1 B=2434931, B-pow=2434931
```

The B and B-pow are the same for the p - 1 factorizations of the divisors because those factorizations had a very large prime factor:

```
>>> factorint(4410317 - 1) == {2: 2, 617: 1, 1787: 1}
True
>>> 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.

`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:

```
>>> from diofant.ntheory import factorint
>>> factorint(2000) # 2000 = (2**4) * (5**3)
{2: 4, 5: 3}
>>> factorint(65537) # This number is prime
{65537: 1}
```

For input less than 2, `factorint` behaves as follows:

- `factorint(1)` returns the empty factorization, {}
- `factorint(0)` returns {0:1}
- `factorint(-n)` adds -1:1 to the factors and then factors n

Partial Factorization:

If `limit (> 3)` is specified, the search is stopped after performing trial division up to (and including) the limit (or taking a corresponding number of rho/p-1 steps). This is useful if one has a large number and only is interested in finding small factors (if any). Note that setting a limit does not prevent larger factors from being found early; it simply means that the largest factor may be composite. Since checking for perfect power is relatively cheap, it is done regardless of the limit setting.

This number, for example, has two small factors and a huge semi-prime factor that cannot be reduced easily:

```
>>> from diofant.ntheory import isprime
>>> a = 1407633717262338957430697921446883
>>> f = factorint(a, limit=10000)
>>> f == {991: 1, int(202916782076162456022877024859): 1, 7: 1}
True
>>> isprime(max(f))
False
```

This number has a small factor and a residual perfect power whose base is greater than the limit:

```
>>> factorint(3*101**7, limit=5)
{3: 1, 101: 7}
```

Visual Factorization:

If `visual` is set to `True`, then it will return a visual factorization of the integer. For example:

```
>>> from diofant import pprint
>>> pprint(factorint(4200, visual=True), use_unicode=False)
 3  1  2  1
2 *3 *5 *7
```

Note that this is achieved by using the `evaluate=False` flag in `Mul` and `Pow`. If you do other manipulations with an expression where `evaluate=False`, it may evaluate. Therefore, you should use the `visual` option only for visualization, and use the normal dictionary returned by `visual=False` if you want to perform operations on the factors.

You can easily switch between the two forms by sending them back to `factorint`:

```
>>> from diofant import Mul, Pow
>>> regular = factorint(1764); regular
{2: 2, 3: 2, 7: 2}
>>> pprint(factorint(regular), use_unicode=False)
 2  2  2
2 *3 *7
```

```
>>> visual = factorint(1764, visual=True); pprint(visual, use_unicode=False)
 2  2  2
2 *3 *7
>>> print(factorint(visual))
{2: 2, 3: 2, 7: 2}
```

If you want to send a number to be factored in a partially factored form you can do so with a dictionary or unevaluated expression:

```
>>> factorint(factorint({4: 2, 12: 3})) # twice to toggle to dict form
{2: 10, 3: 3}
>>> factorint(Mul(4, 12, evaluate=False))
{2: 4, 3: 1}
```

The table of the output logic is:

Input	True	False	other
dict	mul	dict	mul
n	mul	dict	dict
mul	mul	dict	dict

See also:

[smoothness](#) (page 256), [smoothness_p](#) (page 256), [divisors](#) (page 264)

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('%s %s' % (base, exp))
...
2 2
2507191691 1
1231026625769 1
```

Any of these methods can optionally be disabled with the following boolean parameters:

- use_trial: Toggle use of trial division
- use_rho: Toggle use of Pollard’s rho method
- use_pm1: Toggle use of Pollard’s p-1 method

factorint also periodically checks if the remaining part is a prime number or a perfect power, and in those cases stops.

If verbose is set to True, detailed progress is printed.

`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.

See also:

[divisors](#) (page 264)

Examples

```
>>> from diofant.ntheory import primefactors, factorint, isprime
>>> primefactors(6)
[2, 3]
>>> primefactors(-5)
[5]
```

```
>>> sorted(factorint(123456).items())
[(2, 6), (3, 1), (643, 1)]
```

```
>>> primefactors(123456)
[2, 3, 643]
```

```
>>> sorted(factorint(10000000001, limit=200).items())
[(101, 1), (99009901, 1)]
>>> isprime(99009901)
False
>>> primefactors(10000000001, limit=300)
[101]
```

`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)`.

See also:

[primefactors](#) (page 263), [factorint](#) (page 261), [divisor_count](#) (page 264)

References

[R377] (page 1261)

Examples

```
>>> from diofant import divisors, divisor_count
>>> divisors(24)
[1, 2, 3, 4, 6, 8, 12, 24]
>>> divisor_count(24)
8
```

```
>>> list(divisors(120, generator=True))
[1, 2, 4, 8, 3, 6, 12, 24, 5, 10, 20, 40, 15, 30, 60, 120]
```

`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.

See also:

[factorint](#) (page 261), [divisors](#) (page 264), [totient](#) (page 265)

References

[R378] (page 1261)

Examples

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

`diofant.ntheory.factor_.totient(cls, *args, **options)`
Calculate the Euler totient function $\phi(n)$

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

See also:

[`divisor_count`](#) (page 264)

`diofant.ntheory.factor_.core(n, t=2)`
Calculate $\text{core}(n,t) = \text{core}_t(n)$ of a positive integer n .
`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 : `core(n,t)` calculates the t -th power free part of n

`core(n, 2)` is the squarefree part of n `core(n, 3)` is the cubefree part of n

Default for t is 2.

See also:

[`factorint`](#) (page 261)

References

[R379] (page 1261)

Examples

```
>>> from diofant.ntheory.factor_ import core
>>> core(24, 2)
6
>>> core(9424, 3)
1178
>>> core(379238)
```

```
379238
>>> core(15**11, 10)
15
```

`diofant.ntheory.modular.symmetric_residue(a, m)`

Return the residual mod m such that it is within half of the modulus.

```
>>> from diofant.ntheory.modular import symmetric_residue
>>> symmetric_residue(1, 6)
1
>>> symmetric_residue(4, 6)
-2
```

`diofant.ntheory.modular.crt(m, v, symmetric=False, check=True)`

Chinese Remainder Theorem.

The moduli in m are assumed to be pairwise coprime. The output is then an integer f , such that $f = v_i \pmod{m_i}$ for each pair out of v and m . If `symmetric` is `False` a positive integer will be returned, else $|f|$ will be less than or equal to the LCM of the moduli, and thus f may be negative.

If the moduli are not co-prime the correct result will be returned if/when the test of the result is found to be incorrect. This result will be `None` if there is no solution.

The keyword `check` can be set to `False` if it is known that the moduli are coprime.

As an example consider a set of residues $U = [49, 76, 65]$ and a set of moduli $M = [99, 97, 95]$. Then we have:

```
>>> from diofant.ntheory.modular import crt, solve_congruence
>>> crt([99, 97, 95], [49, 76, 65])
(639985, 912285)
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

If the moduli are not co-prime, you may receive an incorrect result if you use `check=False`:

```
>>> crt([12, 6, 17], [3, 4, 2], check=False)
(954, 1224)
>>> [954 % m for m in [12, 6, 17]]
[6, 0, 2]
>>> crt([12, 6, 17], [3, 4, 2]) is None
True
>>> crt([3, 6], [2, 5])
(5, 6)
```

Note: the order of `gf_crt`'s arguments is reversed relative to `crt`, and that `solve_congruence` takes residue, modulus pairs.

Programmer's note: rather than checking that all pairs of moduli share no GCD (an $O(n^2)$ test) and rather than factoring all moduli and seeing that there is no factor in common, a check that the result gives the indicated residuals is performed - an $O(n)$ operation.

See also:

`solve_congruence` (page 267)

`diofant.polys.galoistools.gf_crt` (page 810) low level crt routine used by this routine

`diofant.ntheory.modular.crt1(m)`

First part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from diofant.ntheory.modular import crt1
>>> crt1([18, 42, 6])
(4536, [252, 108, 756], [0, 2, 0])
```

`diofant.ntheory.modular.crt2(m, v, mm, e, s, symmetric=False)`

Second part of Chinese Remainder Theorem, for multiple application.

Examples

```
>>> from diofant.ntheory.modular import crt1, crt2
>>> mm, e, s = crt1([18, 42, 6])
>>> crt2([18, 42, 6], [0, 0, 0], mm, e, s)
(0, 4536)
```

`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.

See also:

`crt` (page 266) high level routine implementing the Chinese Remainder Theorem

Examples

```
>>> from diofant.ntheory.modular import solve_congruence
```

What number is 2 mod 3, 3 mod 5 and 2 mod 7?

```
>>> solve_congruence((2, 3), (3, 5), (2, 7))
(23, 105)
>>> [23 % m for m in [3, 5, 7]]
[2, 3, 2]
```

If you prefer to work with all remainder in one list and all moduli in another, send the arguments like this:

```
>>> solve_congruence(*zip((2, 3, 2), (3, 5, 7)))
(23, 105)
```

The moduli need not be co-prime; in this case there may or may not be a solution:

```
>>> solve_congruence((2, 3), (4, 6)) is None
True
```

```
>>> solve_congruence((2, 3), (5, 6))
(5, 6)
```

The symmetric flag will make the result be within 1/2 of the modulus:

```
>>> solve_congruence((2, 3), (5, 6), symmetric=True)
(-1, 6)
```

`diofant.ntheory.multinomial.binomial_coefficients(n)`

Return a dictionary containing pairs $(k_1, k_2) : C_{kn}$ where C_{kn} are binomial coefficients and $n = k_1 + k_2$.

See also:

[binomial_coefficients_list](#) (page 268), [multinomial_coefficients](#) (page 268)

Examples

```
>>> from diofant.ntheory import binomial_coefficients
>>> binomial_coefficients(9) == {(0, 9): 1, (1, 8): 9, (2, 7): 36,
... (3, 6): 84, (4, 5): 126, (5, 4): 126, (6, 3): 84,
... (7, 2): 36, (8, 1): 9, (9, 0): 1}
True
```

`diofant.ntheory.multinomial.binomial_coefficients_list(n)`

Return a list of binomial coefficients as rows of the Pascal's triangle.

See also:

[binomial_coefficients](#) (page 268), [multinomial_coefficients](#) (page 268)

Examples

```
>>> from diofant.ntheory import binomial_coefficients_list
>>> binomial_coefficients_list(9)
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```

`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 $n = k_1 + k_2 + \dots + k_m$.

See also:

[binomial_coefficients_list](#) (page 268), [binomial_coefficients](#) (page 268)

Notes

The algorithm is based on the following result:

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

Examples

```
>>> from diofant.ntheory import multinomial_coefficients
>>> multinomial_coefficients(2, 5) == {(0, 5): 1, (1, 4): 5,
...   (2, 3): 10, (3, 2): 10, (4, 1): 5, (5, 0): 1}
True
```

`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.

```
>>> from diofant.ntheory.multinomial import multinomial_coefficients
>>> m53, m33 = multinomial_coefficients(5,3), multinomial_coefficients(3,3)
>>> m53[(0,0,0,1,2)] == m53[(0,0,1,0,2)] == m53[(1,0,2,0,0)] == m33[(0,1,2)]
True
```

Examples

```
>>> from diofant.ntheory.multinomial import multinomial_coefficients_iterator
>>> it = multinomial_coefficients_iterator(20,3)
>>> next(it)
((3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), 1)
```

`diofant.ntheory.partitions_.npartitions`(*n*, *verbose=False*)

Calculate the partition function $P(n)$, i.e. the number of ways that *n* can be written as a sum of positive integers.

$P(n)$ is computed using the Hardy-Ramanujan-Rademacher formula [R380] (page 1261).

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

References

[R380] (page 1261)

Examples

```
>>> from diofant.ntheory import npartitions
>>> npartitions(25)
1958
```

`diofant.ntheory.primetest.mr(n, bases)`

Perform a Miller-Rabin strong pseudoprime test on *n* using a given list of bases/witnesses.

References

[R381] (page 1261), [R382] (page 1261)

Examples

```
>>> from diofant.ntheory.primetest import mr
>>> mr(1373651, [2, 3])
False
>>> mr(479001599, [31, 73])
True
```

`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} .

See also:

[`diofant.ntheory.generate.primerange` \(page 253\)](#) Generates all primes in a given range

[`diofant.ntheory.generate.primepi` \(page 252\)](#) Return the number of primes less than or equal to *n*

[`diofant.ntheory.generate.prime` \(page 251\)](#) Return the *n*th prime

Examples

```
>>> from diofant.ntheory import isprime
>>> isprime(13)
True
>>> isprime(15)
False
```

`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

```
>>> from diofant.ntheory import n_order
>>> n_order(3, 7)
6
>>> n_order(4, 7)
3
```

`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

```
>>> from diofant.ntheory import is_primitive_root, n_order, totient
>>> is_primitive_root(3, 10)
True
>>> is_primitive_root(9, 10)
False
>>> n_order(3, 10) == totient(10)
True
>>> n_order(9, 10) == totient(10)
False
```

`diofant.ntheory.residue_ntheory.primitive_root(p)`

Returns the smallest primitive root or None.

Parameters p : positive integer

References

[R383] (page 1261), [R384] (page 1261)

Examples

```
>>> from diofant.ntheory.residue_ntheory import primitive_root
>>> primitive_root(19)
2
```

`diofant.ntheory.residue_ntheory.sqrt_mod(a, p, all_roots=False)`

Find a root of $x^{**2} = a \text{ mod } p$.

Parameters **a** : integer

p : positive integer

all_roots : if True the list of roots is returned or None

Notes

If there is no root it is returned None; else the returned root is less or equal to $p // 2$; in general is not the smallest one. It is returned $p // 2$ only if it is the only root.

Use `all_roots` only when it is expected that all the roots fit in memory; otherwise use `sqrt_mod_iter`.

Examples

```
>>> from diofant.ntheory import sqrt_mod
>>> sqrt_mod(11, 43)
21
>>> sqrt_mod(17, 32, True)
[7, 9, 23, 25]
```

`diofant.ntheory.residue_ntheory.quadratic_residues(p)`

Returns the list of quadratic residues.

Examples

```
>>> from diofant.ntheory.residue_ntheory import quadratic_residues
>>> quadratic_residues(7)
[0, 1, 2, 4]
```

`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

```
>>> from diofant.ntheory.residue_ntheory import nthroot_mod
>>> nthroot_mod(11, 4, 19)
8
>>> nthroot_mod(11, 4, 19, True)
[8, 11]
>>> nthroot_mod(68, 3, 109)
23
```

`diofant.ntheory.residue_ntheory.is_nthpow_residue(a, n, m)`

Returns True if $x^n = a \pmod m$ has solutions.

References

[R385] (page 1261)

`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 $a \pmod{p}$ in $\{i^2 \pmod{p} \text{ for } i \text{ in } \text{range}(p)\}$. If p is an odd prime, an iterative method is used to make the determination:

```
>>> from diofant.ntheory import is_quad_residue
>>> 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 273), [jacobi_symbol](#) (page 273)

`diofant.ntheory.residue_ntheory.legendre_symbol(a, p)`

Legendre symbol function [R386] (page 1261).

Returns

1. 0 if a is multiple of p
 2. 1 if a is a quadratic residue of p
 3. -1 otherwise
- p should be an odd prime by definition

See also:

[is_quad_residue](#) (page 273), [jacobi_symbol](#) (page 273)

References

[R386] (page 1261)

Examples

```
>>> from diofant.ntheory import legendre_symbol
>>> [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]
```

`diofant.ntheory.residue_ntheory.jacobi_symbol(m, n)`

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

Returns

1. 0 if $m \text{ cong } 0 \pmod{n}$
2. 1 if $x^2 \text{ cong } m \pmod{n}$ has a solution
3. -1 otherwise

See also:

[is_quad_residue](#) (page 273), [legendre_symbol](#) (page 273)

Examples

```
>>> from diofant.ntheory import jacobi_symbol, legendre_symbol
>>> from diofant import Mul, S, Integer
>>> 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
```

`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.

See also:

[continued_fraction_iterator](#) (page 274)

Examples

```
>>> from diofant.core import Rational, pi
>>> from diofant import S
>>> from diofant.ntheory.continued_fraction import continued_fraction_
↳convergents, continued_fraction_iterator
```

```
>>> list(continued_fraction_convergents([0, 2, 1, 2]))
[0, 1/2, 1/3, 3/8]
```

```
>>> list(continued_fraction_convergents([1, 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
```

`diofant.ntheory.continued_fraction.continued_fraction_iterator(x)`
 Return continued fraction expansion of `x` as iterator.

References

[R387] (page 1261)

Examples

```
>>> from diofant.core import Rational, pi
>>> from diofant.ntheory.continued_fraction import continued_fraction_iterator
```

```
>>> list(continued_fraction_iterator(Rational(3, 8)))
[0, 2, 1, 2]
>>> list(continued_fraction_iterator(Rational(-3, 8)))
[-1, 1, 1, 1, 2]
```

```
>>> for i, v in enumerate(continued_fraction_iterator(pi)):
...     if i > 7:
...         break
...     print(v)
3
7
15
1
292
1
1
1
```

`diofant.ntheory.continued_fraction.continued_fraction_periodic(p, q, d=0)`
 Find the periodic continued fraction [R388] (page 1261) 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.

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

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.

See also:

[continued_fraction_iterator](#) (page 274), [continued_fraction_reduce](#) (page 276)

References

[R388] (page 1261), [R389] (page 1261)

Examples

```
>>> from diofant.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_periodic(3, 2, 7)
[2, [1, 4, 1, 1]]
```

Golden ratio has the simplest continued fraction expansion:

```
>>> continued_fraction_periodic(1, 2, 5)
[[1]]
```

If the discriminator is zero or a perfect square then the number will be a rational number:

```
>>> continued_fraction_periodic(4, 3, 0)
[1, 3]
>>> continued_fraction_periodic(4, 3, 49)
[3, 1, 2]
```

`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).

See also:

[*continued_fraction_periodic*](#) (page 275)

Examples

```
>>> from diofant.ntheory.continued_fraction import continued_fraction_reduce
>>> continued_fraction_reduce([1, 2, 3, 4, 5])
225/157
>>> continued_fraction_reduce([-2, 1, 9, 7, 1, 2])
-256/233
>>> continued_fraction_reduce([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8]).n(10)
2.718281835
>>> continued_fraction_reduce([1, 4, 2, [3, 1]])
(sqrt(21) + 287)/238
>>> continued_fraction_reduce([[1]])
1/2 + sqrt(5)/2
>>> from diofant.ntheory.continued_fraction import continued_fraction_periodic
>>> continued_fraction_reduce(continued_fraction_periodic(8, 5, 13))
(sqrt(13) + 8)/5
```


class diofant.ntheory.mobius

Möbius function maps natural number to $\{-1, 0, 1\}$

It is defined as follows:

1. 1 if $n = 1$.
2. 0 if n has a squared prime factor.
3. $(-1)^k$ if n is a square-free positive integer with k number of prime factors.

It is an important multiplicative function in number theory and combinatorics. It has applications in mathematical series, algebraic number theory and also physics (Fermion operator has very concrete realization with Möbius Function model).

Parameters n : positive integer

References

[R390] (page 1261), [R391] (page 1261)

Examples

```
>>> from diofant.ntheory import mobius
>>> mobius(13*7)
1
>>> mobius(1)
1
>>> mobius(13*7*5)
-1
>>> mobius(13**2)
0
```

diofant.ntheory.egyptian_fraction.egyptian_fraction(r , $algorithm='Greedy'$)

Compute an Egyptian fraction of the rational r .

Parameters r : Rational

a positive rational number.

algorithm : { "Greedy", "Graham Jewett", "Takenouchi", "Golomb" }, optional

Denotes the algorithm to be used (the default is "Greedy").

Returns list

The list of denominators of an Egyptian fraction expansion [R392] (page 1261).

See also:

[diofant.core.numbers.Rational](#) (page 89)

Notes

Currently the following algorithms are supported:

1. Greedy Algorithm

Also called the Fibonacci-Sylvester algorithm [R393] (page 1261). At each step, extract the largest unit fraction less than the target and replace the target with the remainder.

It has some distinct properties:

- (a) Given p/q in lowest terms, generates an expansion of maximum length p . Even as the numerators get large, the number of terms is seldom more than a handful.
- (b) Uses minimal memory.
- (c) The terms can blow up (standard examples of this are $5/121$ and $31/311$). The denominator is at most squared at each step (doubly-exponential growth) and typically exhibits singly-exponential growth.

2. Graham Jewett Algorithm

The algorithm suggested by the result of Graham and Jewett. Note that this has a tendency to blow up: the length of the resulting expansion is always $2^{*(x/\gcd(x,y)) - 1}$. See [R394] (page 1261).

3. Takenouchi Algorithm

The algorithm suggested by Takenouchi (1921). Differs from the Graham-Jewett algorithm only in the handling of duplicates. See [R394] (page 1261).

4. Golomb's Algorithm

A method given by Golomb (1962), using modular arithmetic and inverses. It yields the same results as a method using continued fractions proposed by Bleicher (1972). See [R395] (page 1261).

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

[R392] (page 1261), [R393] (page 1261), [R394] (page 1261), [R395] (page 1261)

Examples

```
>>> from diofant import Rational
>>> from diofant.ntheory.egyptian_fraction import egyptian_fraction
>>> egyptian_fraction(Rational(3, 7))
[3, 11, 231]
>>> egyptian_fraction(Rational(3, 7), "Graham Jewett")
[7, 8, 9, 56, 57, 72, 3192]
>>> egyptian_fraction(Rational(3, 7), "Takenouchi")
[4, 7, 28]
>>> egyptian_fraction(Rational(3, 7), "Golomb")
```

```
[3, 15, 35]
>>> egyptian_fraction(Rational(11, 5), "Golomb")
[1, 2, 3, 4, 9, 234, 1118, 2580]
```

3.4 Concrete Mathematics

3.4.1 Hypergeometric terms

The center stage, in recurrence solving and summations, play hypergeometric terms. Formally these are sequences annihilated by first order linear recurrence operators. In simple words if we are given term $a(n)$ then it is hypergeometric if its consecutive term ratio is a rational function in n .

To check if a sequence is of this type you can use the `is_hypergeometric` method which is available in Basic class. Here is simple example involving a polynomial:

```
>>> from diofant import *
>>> n, k = symbols('n,k')
>>> (n**2 + 1).is_hypergeometric(n)
True
```

Of course polynomials are hypergeometric but are there any more complicated sequences of this type? Here are some trivial examples:

```
>>> factorial(n).is_hypergeometric(n)
True
>>> binomial(n, k).is_hypergeometric(n)
True
>>> rf(n, k).is_hypergeometric(n)
True
>>> ff(n, k).is_hypergeometric(n)
True
>>> gamma(n).is_hypergeometric(n)
True
>>> (2**n).is_hypergeometric(n)
True
```

We see that all species used in summations and other parts of concrete mathematics are hypergeometric. Note also that binomial coefficients and both rising and falling factorials are hypergeometric in both their arguments:

```
>>> binomial(n, k).is_hypergeometric(k)
True
>>> rf(n, k).is_hypergeometric(k)
True
>>> ff(n, k).is_hypergeometric(k)
True
```

To say more, all previously shown examples are valid for integer linear arguments:

```
>>> factorial(2*n).is_hypergeometric(n)
True
>>> binomial(3*n+1, k).is_hypergeometric(n)
True
>>> rf(n+1, k-1).is_hypergeometric(n)
```

```
True
>>> ff(n-1, k+1).is_hypergeometric(n)
True
>>> gamma(5*n).is_hypergeometric(n)
True
>>> (2**(n-7)).is_hypergeometric(n)
True
```

However nonlinear arguments make those sequences fail to be hypergeometric:

```
>>> factorial(n**2).is_hypergeometric(n)
False
>>> (2**(n**3 + 1)).is_hypergeometric(n)
False
```

If not only the knowledge of being hypergeometric or not is needed, you can use `hypersimp()` function. It will try to simplify combinatorial expression and if the term given is hypergeometric it will return a quotient of polynomials of minimal degree. Otherwise it will return `None` to say that sequence is not hypergeometric:

```
>>> hypersimp(factorial(2*n), n)
2*(n + 1)*(2*n + 1)
>>> hypersimp(factorial(n**2), n)
```

3.4.2 Concrete Class Reference

class `diofant.concrete.summations.Sum`

Represents unevaluated summation.

`Sum` represents a finite or infinite series, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the summation.

For finite sums (and sums with symbolic limits assumed to be finite) we follow the summation convention described by Karr [1], especially definition 3 of section 1.4. The sum:

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

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

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

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

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

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

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

It is important to note that Karr defines all sums with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the summation convention. Indeed we have:

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

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

See also:

[diofant.concrete.summations.summation](#) (page 293), [diofant.concrete.products.Product](#) (page 284), [diofant.concrete.products.product](#) (page 294)

References

[R21] (page 1261), [R22] (page 1261), [R23] (page 1261)

Examples

```
>>> from diofant.abc import i, k, m, n, x
>>> from diofant import factorial, oo, IndexedBase, Function
```

```
>>> Sum(k, (k, 1, m))
Sum(k, (k, 1, m))
>>> Sum(k, (k, 1, m)).doit()
m**2/2 + m/2
>>> Sum(k**2, (k, 1, m))
Sum(k**2, (k, 1, m))
>>> Sum(k**2, (k, 1, m)).doit()
m**3/3 + m**2/2 + m/6
>>> Sum(x**k, (k, 0, oo))
Sum(x**k, (k, 0, oo))
>>> Sum(x**k, (k, 0, oo)).doit()
Piecewise((1/(-x + 1), Abs(x) < 1), (Sum(x**k, (k, 0, oo)), true))
>>> Sum(x**k/factorial(k), (k, 0, oo)).doit()
E**x
```

Here are examples to do summation with symbolic indices. You can use either Function of IndexedBase classes:

```
>>> f = Function('f')
```

```
>>> Sum(f(n), (n, 0, 3)).doit()
f(0) + f(1) + f(2) + f(3)
>>> Sum(f(n), (n, 0, oo)).doit()
Sum(f(n), (n, 0, oo))
>>> f = IndexedBase('f')
>>> Sum(f[n]**2, (n, 0, 3)).doit()
f[0]**2 + f[1]**2 + f[2]**2 + f[3]**2
```

An example showing that the symbolic result of a summation is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those sums by interchanging the limits according to the above rules:

```
>>> S = Sum(i, (i, 1, n)).doit()
>>> S
n**2/2 + n/2
>>> S.subs(n, -4)
6
>>> Sum(i, (i, 1, -4)).doit()
6
>>> Sum(-i, (i, -3, 0)).doit()
6
```

An explicit example of the Karr summation convention:

```
>>> S1 = Sum(i**2, (i, m, m+n-1)).doit()
>>> S1
m**2*n + m*n**2 - m*n + n**3/3 - n**2/2 + n/6
>>> S2 = Sum(i**2, (i, m+n, m-1)).doit()
>>> S2
-m**2*n - m*n**2 + m*n - n**3/3 + n**2/2 - n/6
>>> S1 + S2
0
>>> S3 = Sum(i, (i, m, m-1)).doit()
>>> S3
0
```

euler_maclaurin(*m=0, n=0, eps=0, eval_integral=True*)

Return an Euler-Maclaurin approximation of self, where *m* is the number of leading terms to sum directly and *n* is the number of terms in the tail.

With *m = n = 0*, this is simply the corresponding integral plus a first-order endpoint correction.

Returns (*s, e*) where *s* is the Euler-Maclaurin approximation and *e* is the estimated error (taken to be the magnitude of the first omitted term in the tail):

```
>>> from diofant.abc import k, a, b
```

```
>>> Sum(1/k, (k, 2, 5)).doit().evalf()
1.28333333333333
>>> s, e = Sum(1/k, (k, 2, 5)).euler_maclaurin()
>>> s
-log(2) + 7/20 + log(5)
>>> from diofant import sstr
>>> print(sstr((s.evalf(), e.evalf()), full_prec=True))
(1.26629073187415, 0.0175000000000000)
```

The endpoints may be symbolic:

```
>>> s, e = Sum(1/k, (k, a, b)).euler_maclaurin()
>>> s
-log(a) + log(b) + 1/(2*b) + 1/(2*a)
>>> e
Abs(1/(12*b**2) - 1/(12*a**2))
```

If the function is a polynomial of degree at most $2n+1$, the Euler-Maclaurin formula becomes exact (and $e = 0$ is returned):

```
>>> Sum(k, (k, 2, b)).euler_maclaurin()
(b**2/2 + b/2 - 1, 0)
>>> Sum(k, (k, 2, b)).doit()
b**2/2 + b/2 - 1
```

With a nonzero `eps` specified, the summation is ended as soon as the remainder term is less than the `epsilon`.

findrecur(*F=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

$$\sum_{i=0}^I \sum_{j=0}^J a_{i,j} F(n-j, k-i) = 0$$

Notes

We use Sister Celine's algorithm, see [\[R24\]](#) (page 1261).

References

[\[R24\]](#) (page 1261)

Examples

```
>>> from diofant import symbols, factorial, oo
```

```
>>> n, k = symbols('n, k', integer=True)
>>> 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)
```

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.

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 291), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 293), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 292)

References

[R25] (page 1261)

Examples

```
>>> from diofant.abc import x, y, a, b, c, d
```

```
>>> Sum(x, (x, 0, 3)).reverse_order(x)
Sum(-x, (x, 4, -1))
>>> Sum(x*y, (x, 1, 5), (y, 0, 6)).reverse_order(x, y)
Sum(x*y, (x, 6, 0), (y, 7, -1))
>>> Sum(x, (x, a, b)).reverse_order(x)
Sum(-x, (x, b + 1, a - 1))
>>> Sum(x, (x, a, b)).reverse_order(0)
Sum(-x, (x, b + 1, a - 1))
```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```
>>> S = Sum(x**2, (x, a, b), (x, c, d))
>>> S
Sum(x**2, (x, a, b), (x, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x**2, (x, b + 1, a - 1), (x, c, d))
>>> S1 = S0.reverse_order(1)
>>> S1
Sum(x**2, (x, b + 1, a - 1), (x, d + 1, c - 1))
```

Of course we can mix both notations:

```
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(x, 1)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
>>> Sum(x*y, (x, a, b), (y, 2, 5)).reverse_order(y, x)
Sum(x*y, (x, b + 1, a - 1), (y, 6, 1))
```

class diofant.concrete.products.Product

Represents unevaluated products.

`Product` represents a finite or infinite product, with the first argument being the general form of terms in the series, and the second argument being `(dummy_variable, start, end)`, with `dummy_variable` taking all integer values from `start` through `end`. In accordance with long-standing mathematical convention, the end term is included in the product.

For finite products (and products with symbolic limits assumed to be finite) we follow the analogue of the summation convention described by Karr [1], especially definition 3 of section 1.4. The product:

$$\prod_{m \leq i < n} f(i)$$

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

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

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

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

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

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

It is important to note that above we define all products with the upper limit being exclusive. This is in contrast to the usual mathematical notation, but does not affect the product convention. Indeed we have:

$$\prod_{m \leq i < n} f(i) = \prod_{i=m}^{n-1} f(i)$$

where the difference in notation is intentional to emphasize the meaning, with limits typeset on the top being inclusive.

See also:

[diofant.concrete.summations.Sum](#) (page 280), [diofant.concrete.summations.summation](#) (page 293), [diofant.concrete.products.product](#) (page 294)

References

[R26] (page 1261), [R27] (page 1261), [R28] (page 1261)

Examples

```
>>> from diofant.abc import a, b, i, k, m, n, x
>>> from diofant import factorial, oo
```

```
>>> Product(k, (k, 1, m))
Product(k, (k, 1, m))
>>> Product(k, (k, 1, m)).doit()
factorial(m)
>>> Product(k**2, (k, 1, m))
Product(k**2, (k, 1, m))
>>> Product(k**2, (k, 1, m)).doit()
(factorial(m))**2
```

Wallis' product for pi:

```
>>> W = Product(2*i/(2*i-1) * 2*i/(2*i+1), (i, 1, oo))
>>> W
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

Direct computation currently fails:

```
>>> W.doit()
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, oo))
```

But we can approach the infinite product by a limit of finite products:

```
>>> from diofant import limit
>>> W2 = Product(2*i/(2*i-1)*2*i/(2*i+1), (i, 1, n))
>>> W2
Product(4*i**2/((2*i - 1)*(2*i + 1)), (i, 1, n))
>>> W2e = W2.doit()
>>> W2e
2**(-2*n)*4**n*(factorial(n))**2/(RisingFactorial(1/2, n)*RisingFactorial(3/2, n))
>>> limit(W2e, n, oo)
pi/2
```

By the same formula we can compute $\sin(\pi/2)$:

```
>>> from diofant import pi, gamma, simplify
```

```
>>> P = pi * x * Product(1 - x**2/k**2, (k, 1, n))
>>> P = P.subs(x, pi/2)
>>> P
pi**2*Product(1 - pi**2/(4*k**2), (k, 1, n))/2
>>> Pe = P.doit()
>>> Pe
pi**2*RisingFactorial(1 + pi/2, n)*RisingFactorial(-pi/2 + 1, n)/
↳ (2*(factorial(n))**2)
>>> Pe = Pe.rewrite(gamma)
>>> Pe
pi**2*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/(2*gamma(1 + pi/2)*gamma(-pi/2 +
↳ 1)*gamma(n + 1)**2)
>>> Pe = simplify(Pe)
>>> Pe
sin(pi**2/2)*gamma(n + 1 + pi/2)*gamma(n - pi/2 + 1)/gamma(n + 1)**2
>>> limit(Pe, n, oo)
sin(pi**2/2)
```

Products with the lower limit being larger than the upper one:

```
>>> Product(1/i, (i, 6, 1)).doit()
120
>>> Product(i, (i, 2, 5)).doit()
120
```

The empty product:

```
>>> Product(i, (i, n, n-1)).doit()
1
```

An example showing that the symbolic result of a product is still valid for seemingly nonsensical values of the limits. Then the Karr convention allows us to give a perfectly valid interpretation to those products by interchanging the limits according to the above rules:

```
>>> P = Product(2, (i, 10, n)).doit()
>>> P
2**(n - 9)
>>> P.subs(n, 5)
1/16
>>> Product(2, (i, 10, 5)).doit()
1/16
```

```
>>> 1/Product(2, (i, 6, 9)).doit()
1/16
```

An explicit example of the Karr summation convention applied to products:

```
>>> P1 = Product(x, (i, a, b)).doit()
>>> P1
x**(-a + b + 1)
>>> P2 = Product(x, (i, b+1, a-1)).doit()
>>> P2
x**(a - b - 1)
>>> simplify(P1 * P2)
1
```

And another one:

```
>>> P1 = Product(i, (i, b, a)).doit()
>>> P1
RisingFactorial(b, a - b + 1)
>>> P2 = Product(i, (i, a+1, b-1)).doit()
>>> P2
RisingFactorial(a + 1, -a + b - 1)
>>> P1 * P2
RisingFactorial(b, a - b + 1)*RisingFactorial(a + 1, -a + b - 1)
>>> simplify(P1 * P2)
1
```

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.

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 291), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 293), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 292)

References

[R29] (page 1262)

Examples

```
>>> from diofant import simplify, RisingFactorial, gamma, Sum
>>> from diofant.abc import x, y, a, b, c, d
```

```
>>> P = Product(x, (x, a, b))
>>> Pr = P.reverse_order(x)
>>> Pr
```

```

Product(1/x, (x, b + 1, a - 1))
>>> Pr = Pr.doit()
>>> Pr
1/RisingFactorial(b + 1, a - b - 1)
>>> simplify(Pr)
gamma(b + 1)/gamma(a)
>>> P = P.doit()
>>> P
RisingFactorial(a, -a + b + 1)
>>> simplify(P)
gamma(b + 1)/gamma(a)

```

While one should prefer variable names when specifying which limits to reverse, the index counting notation comes in handy in case there are several symbols with the same name.

```

>>> S = Sum(x*y, (x, a, b), (y, c, d))
>>> S
Sum(x*y, (x, a, b), (y, c, d))
>>> S0 = S.reverse_order(0)
>>> S0
Sum(-x*y, (x, b + 1, a - 1), (y, c, d))
>>> S1 = S0.reverse_order(1)
>>> 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))

```

class `diofant.concrete.expr_with_limits.ExprWithLimits`

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.

See also:

[*diofant.concrete.expr_with_limits.ExprWithLimits.variables* \(page 289\)](#)

Lists the integration variables

Examples

```

>>> from diofant import Integral
>>> from diofant.abc import x, y
>>> 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.

free_symbols

This method returns the symbols in the object, excluding those that take on a specific value (i.e. the dummy symbols).

Examples

```
>>> from diofant import Sum
>>> from diofant.abc import x, y
>>> Sum(x, (x, y, 1)).free_symbols
{y}
```

function

Return the function applied across limits.

See also:

diofant.concrete.expr_with_limits.ExprWithLimits.limits (page 289), *diofant.concrete.expr_with_limits.ExprWithLimits.variables* (page 289), *diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols* (page 289)

Examples

```
>>> from diofant import Integral
>>> from diofant.abc import x
>>> Integral(x**2, (x,)).function
x**2
```

is_number

Return True if the Sum has no free symbols, else False.

limits

Return the limits of expression.

See also:

diofant.concrete.expr_with_limits.ExprWithLimits.function (page 289), *diofant.concrete.expr_with_limits.ExprWithLimits.variables* (page 289), *diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols* (page 289)

Examples

```
>>> from diofant import Integral
>>> from diofant.abc import x, i
>>> Integral(x**i, (i, 1, 3)).limits
((i, 1, 3),)
```

variables

Return a list of the dummy variables

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

See also:

[diofant.concrete.expr_with_limits.ExprWithLimits.function](#) (page 289), [diofant.concrete.expr_with_limits.ExprWithLimits.limits](#) (page 289), [diofant.concrete.expr_with_limits.ExprWithLimits.free_symbols](#) (page 289)

[diofant.concrete.expr_with_limits.ExprWithLimits.as_dummy](#) (page 288)

Rename dummy variables

class diofant.concrete.expr_with_intlimits.ExprWithIntLimits

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.

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 291), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 293), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 292), [diofant.concrete.summations.Sum.reverse_order](#) (page 283), [diofant.concrete.products.Product.reverse_order](#) (page 287)

Examples

```
>>> from diofant import Sum, Product, simplify
>>> from diofant.abc import x, y, a, b, c, d, u, v, i, j, k, l
```

```
>>> 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 .

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

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit](#) (page 293), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#)

(page 292), *diofant.concrete.summations.Sum.reverse_order* (page 283), *diofant.concrete.products.Product.reverse_order* (page 287)

Examples

```
>>> from diofant.abc import x, y, a, b, c, d
>>> from diofant import Sum, Product
>>> 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
```

`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.

See also:

diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index (page 291), *diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder_limit* (page 293), *diofant.concrete.summations.Sum.reverse_order* (page 283), *diofant.concrete.products.Product.reverse_order* (page 287)

Examples

```
>>> from diofant import Sum, Product
>>> from diofant.abc import x, y, z, a, b, c, d, 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))
```

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.

See also:

[diofant.concrete.expr_with_intlimits.ExprWithIntLimits.index](#) (page 291), [diofant.concrete.expr_with_intlimits.ExprWithIntLimits.reorder](#) (page 292), [diofant.concrete.summations.Sum.reverse_order](#) (page 283), [diofant.concrete.products.Product.reverse_order](#) (page 287)

Examples

```
>>> from diofant.abc import x, y, z, a, b, c, d, e, f
>>> from diofant import Sum, Product
```

```
>>> 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))
```

3.4.3 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:

```
>>> from diofant import oo, symbols, log
```

```
>>> i, n, m = symbols('i n m', integer=True)
```

```
>>> summation(2*i - 1, (i, 1, n))
n**2
>>> summation(1/2**i, (i, 0, oo))
2
>>> summation(1/log(n)**n, (n, 2, oo))
Sum(log(n)**(-n), (n, 2, oo))
>>> summation(i, (i, 0, n), (n, 0, m))
m**3/6 + m**2/2 + m/3
```

```
>>> from diofant.abc import x
>>> from diofant import factorial
```

```
>>> summation(x**n/factorial(n), (n, 0, oo))
E**x
```

See also:

[diofant.concrete.summations.Sum](#) (page 280), [diofant.concrete.products.Product](#) (page 284), [diofant.concrete.products.product](#) (page 294)

`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:

```
>>> from diofant import symbols
>>> i, n, m, k = symbols('i n m k', integer=True)
```

```
>>> product(i, (i, 1, k))
factorial(k)
>>> product(m, (i, 1, k))
m**k
>>> product(i, (i, 1, k), (k, 1, n))
Product(factorial(k), (k, 1, n))
```

`diofant.concrete.gosper.gosper_normal(f, g, n, polys=True)`

Compute the Gosper's normal form of `f` and `g`.

Given relatively prime univariate polynomials `f` and `g`, rewrite their quotient to a normal form defined as follows:

$$\frac{f(n)}{g(n)} = Z \cdot \frac{A(n)C(n+1)}{B(n)C(n)}$$

where `Z` is an arbitrary constant and `A`, `B`, `C` are monic polynomials in `n` with the following properties:

1. $\gcd(A(n), B(n+h)) = 1 \forall h \in \mathbb{N}$
2. $\gcd(B(n), C(n+1)) = 1$
3. $\gcd(A(n), C(n)) = 1$

This normal form, or rational factorization in other words, is a crucial step in Gosper's algorithm and in solving of difference equations. It can be also used to decide if two hypergeometric terms are similar or not.

This procedure will return a tuple containing elements of this factorization in the form $(Z*A, B, C)$.

Examples

```
>>> from diofant.abc import n
```

```
>>> gosper_normal(4*n + 5, 2*(4*n + 1)*(2*n + 3), n, polys=False)
(1/4, n + 3/2, n + 1/4)
```

`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

```
>>> from diofant.functions import factorial
>>> from diofant.abc import n
```

```
>>> gosper_term((4*n + 1)*factorial(n)/factorial(2*n + 1), n)
(-n - 1/2)/(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.

References

[R30] (page 1262)

Examples

```
>>> from diofant.functions import factorial
>>> from diofant.abc import i, n, k
```

```
>>> f = (4*k + 1)*factorial(k)/factorial(2*k + 1)
>>> gosper_sum(f, (k, 0, n))
(-factorial(n) + 2*factorial(2*n + 1))/factorial(2*n + 1)
>>> _.subs(n, 2) == sum(f.subs(k, i) for i in [0, 1, 2])
True
>>> gosper_sum(f, (k, 3, n))
(-60*factorial(n) + factorial(2*n + 1))/(60*factorial(2*n + 1))
>>> _.subs(n, 5) == sum(f.subs(k, i) for i in [3, 4, 5])
True
```

3.5 Numerical evaluation

3.5.1 Basics

Exact Diofant expressions can be converted to floating-point approximations (decimal numbers) using either the `.evalf()` method or the `N()` function. `N(expr, <args>)` is equivalent to `sympify(expr).evalf(<args>)`.

```
>>> from diofant import *
>>> N(sqrt(2)*pi)
4.44288293815837
>>> (sqrt(2)*pi).evalf()
4.44288293815837
```

By default, numerical evaluation is performed to an accuracy of 15 decimal digits. You can optionally pass a desired accuracy (which should be a positive integer) as an argument to `evalf` or `N`:

```
>>> N(sqrt(2)*pi, 5)
4.4429
>>> N(sqrt(2)*pi, 50)
4.4428829381583662470158809900606936986146216893757
```

Complex numbers are supported:

```
>>> N(1/(pi + I), 20)
0.28902548222223624241 - 0.091999668350375232456*I
```

If the expression contains symbols or for some other reason cannot be evaluated numerically, calling `.evalf()` or `N()` returns the original expression, or in some cases a partially evaluated expression. For example, when the expression is a polynomial in expanded form, the coefficients are evaluated:

```
>>> x = Symbol('x')
>>> (pi*x**2 + x/3).evalf()
3.14159265358979*x**2 + 0.333333333333333*x
```

You can also use the standard Python functions `float()`, `complex()` to convert Diofant expressions to regular Python numbers:


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

So the displayed precision should not be used as a model of error propagation or significance arithmetic; rather, this scheme is employed to ensure stability of numerical algorithms.

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

```
>>> N(3.5)
3.5000000000000000
>>> N(3.5, 5)
3.5000
>>> N(3.5, 30)
3.50000000000000000000000000000000
```

3.5.3 Accuracy and error handling

When the input to `N` or `evalf` is a complicated expression, numerical error propagation becomes a concern. As an example, consider the 100'th Fibonacci number and the excellent (but not exact) approximation $\varphi^{100}/\sqrt{5}$ where φ is the golden ratio. With ordinary floating-point arithmetic, subtracting these numbers from each other erroneously results in a complete cancellation:

```
>>> a, b = GoldenRatio**1000/sqrt(5), fibonacci(1000)
>>> float(a)
4.3466557686937455e+208
>>> float(b)
4.3466557686937455e+208
>>> float(a) - float(b)
0.0
```

`N` and `evalf` keep track of errors and automatically increase the precision used internally in order to obtain a correct result:

```
>>> N(fibonacci(100) - GoldenRatio**100/sqrt(5))
-5.64613129282185e-22
```

Unfortunately, numerical evaluation cannot tell an expression that is exactly zero apart from one that is merely very small. The working precision is therefore capped, by default to around 100 digits. If we try with the 1000'th Fibonacci number, the following happens:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5))
0.e+85
```

The lack of digits in the returned number indicates that `N` failed to achieve full accuracy. The result indicates that the magnitude of the expression is something less than 10^{84} , but that is not a particularly good answer. To force a higher working precision, the `maxn` keyword argument can be used:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), maxn=500)
-4.60123853010113e-210
```

Normally, `maxn` can be set very high (thousands of digits), but be aware that this may cause significant slowdown in extreme cases. Alternatively, the `strict=True` option can be set to force an exception instead of silently returning a value with less than the requested accuracy:

```
>>> N(fibonacci(1000) - (GoldenRatio)**1000/sqrt(5), strict=True)
Traceback (most recent call last):
...
PrecisionExhausted: Failed to distinguish the expression:
-sqrt(5)*GoldenRatio**1000/5 +
└─4346655768693745643568852767504062580256466051737178040248172908953655541794905189040387984007925
from zero. Try simplifying the input, using chop=True, or providing a higher maxn for
└─evalf
```

If we add a term so that the Fibonacci approximation becomes exact (the full form of Binet's formula), we get an expression that is exactly zero, but N does not know this:

```
>>> f = fibonacci(100) - (GoldenRatio**100 - (GoldenRatio-1)**100)/sqrt(5)
>>> N(f)
0.e-104
>>> N(f, maxn=1000)
0.e-1336
```

In situations where such cancellations are known to occur, the chop options is useful. This basically replaces very small numbers in the real or imaginary portions of a number with exact zeros:

```
>>> N(f, chop=True)
0
>>> N(3 + I*f, chop=True)
3.000000000000000
```

In situations where you wish to remove meaningless digits, re-evaluation or the use of the round method are useful:

```
>>> Float('.1', '')*Float('.12345', '')
0.012297
>>> ans = _
>>> N(ans, 1)
0.01
>>> ans.round(2)
0.01
```

If you are dealing with a numeric expression that contains no floats, it can be evaluated to arbitrary precision. To round the result relative to a given decimal, the round method is useful:

```
>>> v = 10*pi + cos(1)
>>> N(v)
31.9562288417661
>>> v.round(3)
31.956
```

3.5.4 Sums and integrals

Sums (in particular, infinite series) and integrals can be used like regular closed-form expressions, and support arbitrary-precision evaluation:

```
>>> var('n x')
(n, x)
>>> Sum(1/n**n, (n, 1, oo)).evalf()
1.29128599706266
>>> Integral(x**(-x), (x, 0, 1)).evalf()
1.29128599706266
>>> Sum(1/n**n, (n, 1, oo)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> Integral(x**(-x), (x, 0, 1)).evalf(50)
1.2912859970626635404072825905956005414986193682745
>>> (Integral(exp(-x**2), (x, -oo, oo)) ** 2).evalf(30)
3.14159265358979323846264338328
```

By default, the tanh-sinh quadrature algorithm is used to evaluate integrals. This algorithm is very efficient and robust for smooth integrands (and even integrals with endpoint singularities), but may struggle with integrals that are highly oscillatory or have mid-interval discontinuities. In many cases, `evalf/N` will correctly estimate the error. With the following integral, the result is accurate but only good to four digits:

```
>>> f = abs(sin(x))
>>> Integral(abs(sin(x)), (x, 0, 4)).evalf()
2.346
```

It is better to split this integral into two pieces:

```
>>> (Integral(f, (x, 0, pi)) + Integral(f, (x, pi, 4))).evalf()
2.34635637913639
```

A similar example is the following oscillatory integral:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(maxn=20)
0.5
```

It can be dealt with much more efficiently by telling `evalf` or `N` to use an oscillatory quadrature algorithm:

```
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(quad='osc')
0.504067061906928
>>> Integral(sin(x)/x**2, (x, 1, oo)).evalf(20, quad='osc')
0.50406706190692837199
```

Oscillatory quadrature requires an integrand containing a factor $\cos(ax+b)$ or $\sin(ax+b)$. Note that many other oscillatory integrals can be transformed to this form with a change of variables:

```
>>> init_printing(pretty_print=True, use_unicode=False,
...               wrap_line=False, no_global=True)
>>> intgr1 = Integral(sin(1/x), (x, 0, 1)).transform(x, 1/x)
>>> intgr1
oo
/
|
| sin(x)
| ----- dx
|      2
|      x
|
/
```



```
1
>>> N(intgr1, quad='osc')
0.504067061906928
```

Infinite series use direct summation if the series converges quickly enough. Otherwise, extrapolation methods (generally the Euler-Maclaurin formula but also Richardson extrapolation) are used to speed up convergence. This allows high-precision evaluation of slowly convergent series:

```
>>> var('k')
k
>>> Sum(1/k**2, (k, 1, oo)).evalf()
1.64493406684823
>>> zeta(2).evalf()
1.64493406684823
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf()
0.577215664901533
>>> Sum(1/k-log(1+1/k), (k, 1, oo)).evalf(50)
0.57721566490153286060651209008240243104215933593992
>>> EulerGamma.evalf(50)
0.57721566490153286060651209008240243104215933593992
```

The Euler-Maclaurin formula is also used for finite series, allowing them to be approximated quickly without evaluating all terms:

```
>>> Sum(1/k, (k, 10000000, 20000000)).evalf()
0.693147255559946
```

Note that `evalf` makes some assumptions that are not always optimal. For fine-tuned control over numerical summation, it might be worthwhile to manually use the method `Sum.euler_maclaurin`.

Special optimizations are used for rational hypergeometric series (where the term is a product of polynomials, powers, factorials, binomial coefficients and the like). `N/evalf` sum series of this type very rapidly to high precision. For example, this Ramanujan formula for π can be summed to 10,000 digits in a fraction of a second with a simple command:

```
>>> f = factorial
>>> n = Symbol('n', integer=True)
>>> R = 9801/sqrt(8)/Sum(f(4*n)*(1103+26390*n)/f(n)**4/396**(4*n),
...                      (n, 0, oo))
>>> N(R, 10000)
3.141592653589793238462643383279502884197169399375105820974944592307...
```

3.5.5 Numerical simplification

The function `nsimplify` attempts to find a formula that is numerically equal to the given input. This feature can be used to guess an exact formula for an approximate floating-point input, or to guess a simpler formula for a complicated symbolic input. The algorithm used by `nsimplify` is capable of identifying simple fractions, simple algebraic expressions, linear combinations of given constants, and certain elementary functional transformations of any of the preceding.

Optionally, `nsimplify` can be passed a list of constants to include (e.g. π) and a minimum numerical tolerance. Here are some elementary examples:

```

>>> nsimplify(0.1)
1/10
>>> nsimplify(6.28, [pi], tolerance=0.01)
2*pi
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(pi, tolerance=0.001)
355
---
113
>>> nsimplify(0.33333, tolerance=1e-4)
1/3
>>> 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*GoldenRatio
>>> nsimplify(2 + exp(2*atan('1/4')*I))
49 8*I
-- + ---
17 17
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1      / \sqrt{5}  1
- - I* /  ----- + -
2      \sqrt{10}  4
>>> nsimplify(I**I, [pi])
-pi
----
 2
E
>>> n = Symbol('n')
>>> nsimplify(Sum(1/n**2, (n, 1, oo)), [pi])
2
pi
---
 6
>>> nsimplify(gamma('1/4')*gamma('3/4'), [pi])

```

```
\ / 2 *pi
```

3.6 Numeric Computation

Symbolic computer algebra systems like Diofant facilitate the construction and manipulation of mathematical expressions. Unfortunately when it comes time to evaluate these expressions on numerical data, symbolic systems often have poor performance.

Fortunately Diofant offers a number of easy-to-use hooks into other numeric systems, allowing you to create mathematical expressions in Diofant and then ship them off to the numeric system of your choice. This page documents many of the options available including the `math` library, the popular array computing package `numpy`, code generation in Fortran or C, and the use of the array compiler Theano.

3.6.1 Subs/evalf

`Subs` is the slowest but simplest option. It runs at Diofant speeds. The `.subs(...).evalf()` method can substitute a numeric value for a symbolic one and then evaluate the result within Diofant.

```
>>> from diofant import *
>>> from diofant.abc import x
>>> expr = sin(x)/x
>>> expr.evalf(subs={x: 3.14})
0.000507214304613640
```

This method is slow. You should use this method production only if performance is not an issue. You can expect `.subs` to take tens of microseconds. It can be useful while prototyping or if you just want to see a value once.

3.6.2 Lambdify

The `lambdify` function translates Diofant expressions into Python functions, leveraging a variety of numerical libraries. It is used as follows:

```
>>> from diofant import *
>>> from diofant.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr)
>>> f(3.14)
0.0005072143046136395
```

Here `lambdify` makes a function that computes $f(x) = \sin(x)/x$. By default `lambdify` relies on implementations in the `math` standard library. This numerical evaluation takes on the order of hundreds of nanoseconds, roughly two orders of magnitude faster than the `.subs` method. This is the speed difference between Diofant and raw Python.

`Lambdify` can leverage a variety of numerical backends. By default it uses the `math` library. However it also supports `mpmath` and most notably, `numpy`. Using the `numpy` library gives the generated function access to powerful vectorized ufuncs that are backed by compiled C code.

```
>>> from diofant import *
>>> from diofant.abc import x
>>> expr = sin(x)/x
>>> f = lambdify(x, expr, "numpy")
```

```
>>> import numpy
>>> data = numpy.linspace(1, 10, 10000)
>>> pprint(f(data))
[ 0.84147098  0.84119981  0.84092844 ..., -0.05426074 -0.05433146
  -0.05440211]
```

If you have array-based data this can confer a considerable speedup, on the order of 10 nano-seconds per element. Unfortunately numpy incurs some start-up time and introduces an overhead of a few microseconds.

3.6.3 uFuncify

While NumPy operations are very efficient for vectorized data they sometimes incur unnecessary costs when chained together. Consider the following operation

```
x = get_numpy_array(...)
y = sin(x)/x
```

The operators `sin` and `/` call routines that execute tight for loops in C. The resulting computation looks something like this

```
for(int i = 0; i < n; i++)
{
    temp[i] = sin(x[i]);
}
for(int i = i; i < n; i++)
{
    y[i] = temp[i] / x[i];
}
```

This is slightly sub-optimal because

1. We allocate an extra `temp` array
2. We walk over `x` memory twice when once would have been sufficient

A better solution would fuse both element-wise operations into a single for loop

```
for(int i = i; i < n; i++)
{
    y[i] = sin(x[i]) / x[i];
}
```

Statically compiled projects like NumPy are unable to take advantage of such optimizations. Fortunately, 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

```
>>> from diofant import *
>>> from diofant.abc import x
>>> 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.

3.6.4 Theano

Diofant has a strong connection with [Theano](#), a mathematical array compiler. Diofant expressions can be easily translated to Theano graphs and then compiled using the Theano compiler chain.

```
>>> from diofant import *
>>> from diofant.abc import x
>>> expr = sin(x)/x
```

```
>>> from diofant.printing.theanocode import theano_function
>>> f = theano_function([x], [expr])
```

If array broadcasting or types are desired then Theano requires this extra information

```
>>> f = theano_function([x], [expr], dims={x: 1}, dtypes={x: 'float64'})
```

Theano has a more sophisticated code generation system than Diofant's C/Fortran code printers. Among other things it handles common sub-expressions and compilation onto the GPU. Theano also supports Diofant Matrix and Matrix Expression objects.

3.6.5 So Which Should I Use?

The options here were listed in order from slowest and least dependencies to fastest and most dependencies. For example, if you have Theano installed then that will often be the best choice. If you don't have Theano but do have `f2py` then you should use `ufuncify`.

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

3.7 Mathematical Functions

All functions support the methods documented below, inherited from `diofant.core.function.Function` (page 134).

class `diofant.core.function.Function`

Base class for applied mathematical functions.

It also serves as a constructor for undefined function classes.

Examples

First example shows how to use Function as a constructor for undefined function classes:

```
>>> from diofant import Function, Symbol
>>> x = Symbol('x')
>>> f = Function('f')
>>> g = Function('g')(x)
>>> f
f
>>> f(x)
f(x)
>>> g
g(x)
>>> f(x).diff(x)
Derivative(f(x), x)
>>> g.diff(x)
Derivative(g(x), x)
```

In the following example Function is used as a base class for `my_func` that represents a mathematical function *my_func*. Suppose that it is well known, that *my_func*(0) is 1 and *my_func* at infinity goes to 0, so we want those two simplifications to occur automatically. Suppose also that *my_func*(x) is real exactly when x is real. Here is an implementation that honours those requirements:

```
>>> from diofant import Function, S, oo, I, sin, Symbol
>>> class my_func(Function):
...     @classmethod
...     def eval(cls, x):
...         if x.is_Number:
...             if x is S.Zero:
...                 return S.One
...             elif x is S.Infinity:
...                 return S.Zero
...         def _eval_is_real(self):
...             return self.args[0].is_real
...
>>> x = Symbol('x')
>>> my_func(0) + sin(0)
1
>>> my_func(oo)
0
>>> my_func(3.54).n() # Not yet implemented for my_func.
my_func(3.54)
>>> my_func(I).is_real
False
```

In order for `my_func` to become useful, several other methods would need to be implemented. See source code of some of the already implemented functions for more complete examples.

Also, if the function can take more than one argument, then `nargs` must be defined, e.g. if `my_func` can take one or two arguments then,

```
>>> class my_func(Function):
...     nargs = (1, 2)
```

```
...
>>>
```

as_base_exp()

Returns the method as the 2-tuple (base, exponent).

classmethod class_key()

Nice order of classes.

fdiff(*argindex=1*)

Returns the first derivative of the function.

3.7.1 Elementary

This module implements elementary functions such as trigonometric, hyperbolic as well as functions like Abs, Max, sqrt etc.

3.7.2 diofant.functions.elementary.complexes

re

class diofant.functions.elementary.complexes.**re**

Returns real part of expression.

This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use Basic.as_real_imag() or perform complex expansion on instance of this function.

See also:

[diofant.functions.elementary.complexes.im](#) (page 307)

Examples

```
>>> from diofant import re, im, I, E
>>> from diofant.abc import x, y
>>> re(2*E)
2*E
>>> re(2*I + 17)
17
>>> re(2*I)
0
>>> re(im(x) + x*I + 2)
2
```

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

Returns the real number with a zero imaginary part.

im

class diofant.functions.elementary.complexes.**im**

Returns imaginary part of expression.

This function performs only elementary analysis and so it will fail to decompose properly more complicated expressions. If completely simplified result is needed then use `Basic.as_real_imag()` or perform complex expansion on instance of this function.

See also:

[diofant.functions.elementary.complexes.re](#) (page 307)

Examples

```
>>> from diofant import re, im, E, I
>>> from diofant.abc import x, y
>>> im(2*E)
0
>>> re(2*I + 17)
17
>>> im(x*I)
re(x)
>>> im(re(x) + y)
im(y)
```

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

Return the imaginary part with a zero real part.

Examples

```
>>> from diofant.functions import im
>>> from diofant import I
>>> im(2 + 3*I).as_real_imag()
(3, 0)
```

sign

class `diofant.functions.elementary.complexes.sign`

Returns the complex sign of an expression.

If the expression is real the sign will be:

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

If the expression is imaginary the sign will be:

- I if `im(expression)` is positive
- -I if `im(expression)` is negative

Otherwise an unevaluated expression will be returned. When evaluated, the result (in general) will be $\cos(\arg(\text{expr})) + I\sin(\arg(\text{expr}))$.

See also:

[diofant.functions.elementary.complexes.Abs](#) (page 309), [diofant.functions.elementary.complexes.conjugate](#) (page 310)

Examples

```
>>> from diofant.functions import sign
>>> from diofant.core.numbers import I
```

```
>>> sign(-1)
-1
>>> sign(0)
0
>>> sign(-3*I)
-I
>>> sign(1 + I)
sign(1 + I)
>>> _.evalf()
0.707106781186548 + 0.707106781186548*I
```

Abs

class `diofant.functions.elementary.complexes.Abs`

Return the absolute value of the argument.

This is an extension of the built-in function `abs()` to accept symbolic values. If you pass a Diofant expression to the built-in `abs()`, it will pass it automatically to `Abs()`.

See also:

[`diofant.functions.elementary.complexes.sign`](#) (page 308), [`diofant.functions.elementary.complexes.conjugate`](#) (page 310)

Examples

```
>>> from diofant import Abs, Symbol, S
>>> Abs(-1)
1
>>> x = Symbol('x', extended_real=True)
>>> Abs(-x)
Abs(x)
>>> Abs(x**2)
x**2
>>> abs(-x) # The Python built-in
Abs(x)
```

Note that the Python built-in will return either an `Expr` or `int` depending on the argument:

```
>>> type(abs(-1))
<... 'int'>
>>> type(abs(S.NegativeOne))
<class 'diofant.core.numbers.One'>
```

`Abs` will always return a diofant object.

fdiff(*argindex*=1)

Get the first derivative of the argument to `Abs()`.

Examples

```
>>> from diofant.abc import x
>>> from diofant.functions import Abs
>>> Abs(-x).fdiff()
sign(x)
```

adjoint

class `diofant.functions.elementary.complexes.adjoint`
Conjugate transpose or Hermite conjugation.

arg

class `diofant.functions.elementary.complexes.arg`
Returns the argument (in radians) of a complex number.
For a real number, the argument is always 0.

Examples

```
>>> from diofant.functions import arg
>>> from diofant import I, sqrt
>>> arg(2.0)
0
>>> arg(I)
pi/2
>>> arg(sqrt(2) + I*sqrt(2))
pi/4
```

conjugate

class `diofant.functions.elementary.complexes.conjugate`
Returns the complex conjugate of an argument.

In mathematics, the complex conjugate [R111] (page 1262) of a complex number is given by changing the sign of the imaginary part.

Thus, the conjugate of the complex number $a + ib$ (where a and b are real numbers) is $a - ib$

See also:

[*diofant.functions.elementary.complexes.sign*](#) (page 308), [*diofant.functions.elementary.complexes.Abs*](#) (page 309)

References

[R111] (page 1262)

Examples

```
>>> from diofant import conjugate, I
>>> conjugate(2)
2
>>> conjugate(I)
-I
```

polar_lift

class `diofant.functions.elementary.complexes.polar_lift`

Lift argument to the Riemann surface of the logarithm, using the standard branch.

```
>>> from diofant import Symbol, polar_lift, I
>>> p = Symbol('p', polar=True)
>>> x = Symbol('x')
>>> polar_lift(4)
4*exp_polar(0)
>>> polar_lift(-4)
4*exp_polar(I*pi)
>>> polar_lift(-I)
exp_polar(-I*pi/2)
>>> polar_lift(I + 2)
polar_lift(2 + I)
```

```
>>> polar_lift(4*x)
4*polar_lift(x)
>>> polar_lift(4*p)
4*p
```

See also:

[diofant.functions.elementary.exponential.exp_polar](#) (page 330), [diofant.functions.elementary.complexes.periodic_argument](#) (page 311)

periodic_argument

class `diofant.functions.elementary.complexes.periodic_argument`

Represent the argument on a quotient of the Riemann surface of the logarithm. That is, given a period P , always return a value in $(-P/2, P/2]$, by using $\exp(P \cdot I) == 1$.

```
>>> from diofant import exp, exp_polar, periodic_argument, unbranched_argument
>>> from diofant import I, pi
>>> unbranched_argument(exp(5*I*pi))
pi
>>> unbranched_argument(exp_polar(5*I*pi))
5*pi
>>> periodic_argument(exp_polar(5*I*pi), 2*pi)
pi
>>> periodic_argument(exp_polar(5*I*pi), 3*pi)
-pi
>>> periodic_argument(exp_polar(5*I*pi), pi)
0
```

See also:

[diofant.functions.elementary.exponential.exp_polar](#) (page 330)

[diofant.functions.elementary.complexes.polar_lift](#) (**page 311**) Lift argument to the Riemann surface of the logarithm

[diofant.functions.elementary.complexes.principal_branch](#) (page 312)

principal_branch**class** diofant.functions.elementary.complexes.principal_branch

Represent a polar number reduced to its principal branch on a quotient of the Riemann surface of the logarithm.

This is a function of two arguments. The first argument is a polar number z , and the second one a positive real number of infinity, p . The result is “ $z \bmod \exp_polar(I*p)$ ”.

```
>>> from diofant import exp_polar, principal_branch, oo, I, pi
>>> from diofant.abc import z
>>> principal_branch(z, oo)
z
>>> principal_branch(exp_polar(2*pi*I)*3, 2*pi)
3*exp_polar(0)
>>> principal_branch(exp_polar(2*pi*I)*3*z, 2*pi)
3*principal_branch(z, 2*pi)
```

See also:

[diofant.functions.elementary.exponential.exp_polar](#) (page 330)

[diofant.functions.elementary.complexes.polar_lift](#) (**page 311**) Lift argument to the Riemann surface of the logarithm

[diofant.functions.elementary.complexes.periodic_argument](#) (page 311)

transpose**class** diofant.functions.elementary.complexes.transpose

Linear map transposition.

3.7.3 diofant.functions.elementary.trigonometric**3.7.4 Trigonometric Functions****sin****class** diofant.functions.elementary.trigonometric.sin

The sine function.

Returns the sine of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

This function will evaluate automatically in the case x/π is some rational number [\[R115\]](#) (page 1262). For example, if x is a multiple of π , $\pi/2$, $\pi/3$, $\pi/4$ and $\pi/6$.

References

[\[R112\]](#) (page 1262), [\[R113\]](#) (page 1262), [\[R114\]](#) (page 1262), [\[R115\]](#) (page 1262)

Examples

```
>>> from diofant import sin, pi
>>> from diofant.abc import x
>>> 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
```

cos

class `diofant.functions.elementary.trigonometric.cos`

The cosine function.

Returns the cosine of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>

fant.functions.elementary.trigonometric.acsc (page 322), *dio-*
fant.functions.elementary.trigonometric.acos (page 318), *dio-*
fant.functions.elementary.trigonometric.asec (page 319), *dio-*
fant.functions.elementary.trigonometric.atan (page 320), *dio-*
fant.functions.elementary.trigonometric.acot (page 321), *dio-*
fant.functions.elementary.trigonometric.atan2 (page 323)

Notes

See *sin()* (page 312) for notes about automatic evaluation.

References

[R116] (page 1262), [R117] (page 1262), [R118] (page 1262)

Examples

```
>>> from diofant import cos, pi
>>> from diofant.abc import x
>>> 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
```

tan

class `diofant.functions.elementary.trigonometric.tan`

The tangent function.

Returns the tangent of *x* (measured in radians).

See also:

diofant.functions.elementary.trigonometric.sin (page 312), *dio-*
fant.functions.elementary.trigonometric.csc (page 317), *dio-*
fant.functions.elementary.trigonometric.cos (page 313), *dio-*
fant.functions.elementary.trigonometric.sec (page 316), *dio-*
fant.functions.elementary.trigonometric.cot (page 315), *dio-*
fant.functions.elementary.trigonometric.asin (page 317), *dio-*
fant.functions.elementary.trigonometric.acsc (page 322), *dio-*
fant.functions.elementary.trigonometric.acos (page 318), *dio-*
fant.functions.elementary.trigonometric.asec (page 319), *dio-*
fant.functions.elementary.trigonometric.atan (page 320), *dio-*
fant.functions.elementary.trigonometric.acot (page 321), *dio-*
fant.functions.elementary.trigonometric.atan2 (page 323)

Notes

See `sin()` (page 312) for notes about automatic evaluation.

References

[R119] (page 1262), [R120] (page 1262), [R121] (page 1262)

Examples

```

>>> from diofant import tan, pi
>>> from diofant.abc import x
>>> tan(x**2).diff(x)
2*x*(tan(x**2)**2 + 1)
>>> tan(pi/8).expand()
-1 + sqrt(2)

```

inverse(*argindex*=1)

Returns the inverse of this function.

cot

class `diofant.functions.elementary.trigonometric.cot`

The cotangent function.

Returns the cotangent of *x* (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

See `sin()` (page 312) for notes about automatic evaluation.

References

[R122] (page 1262), [R123] (page 1262), [R124] (page 1262)

Examples

```
>>> from diofant import cot
>>> from diofant.abc import x
>>> cot(x**2).diff(x)
2*x*(-cot(x**2)**2 - 1)
```

inverse(*argindex*=1)

Returns the inverse of this function.

sec

class `diofant.functions.elementary.trigonometric.sec`

The secant function.

Returns the secant of *x* (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

See `sin()` (page 312) for notes about automatic evaluation.

References

[R125] (page 1262), [R126] (page 1262), [R127] (page 1262)

Examples

```
>>> from diofant import sec
>>> from diofant.abc import x
>>> sec(x**2).diff(x)
2*x*tan(x**2)*sec(x**2)
```


csc

class `diofant.functions.elementary.trigonometric.csc`

The cosecant function.

Returns the cosecant of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

See `sin()` (page 312) for notes about automatic evaluation.

References

[R128] (page 1262), [R129] (page 1262), [R130] (page 1262)

Examples

```
>>> from diofant import csc
>>> from diofant.abc import x
>>> csc(x**2).diff(x)
-2*x*cot(x**2)*csc(x**2)
```

3.7.5 Trigonometric Inverses**asin**

class `diofant.functions.elementary.trigonometric.asin`

The inverse sine function.

Returns the arcsine of x in radians.

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>

fant.functions.elementary.trigonometric.tan (page 314), *dio-*
fant.functions.elementary.trigonometric.cot (page 315), *dio-*
fant.functions.elementary.trigonometric.acsc (page 322), *dio-*
fant.functions.elementary.trigonometric.acos (page 318), *dio-*
fant.functions.elementary.trigonometric.asec (page 319), *dio-*
fant.functions.elementary.trigonometric.atan (page 320), *dio-*
fant.functions.elementary.trigonometric.acot (page 321), *dio-*
fant.functions.elementary.trigonometric.atan2 (page 323)

Notes

`asin(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1` and for some instances when the result is a rational multiple of `pi` (see the `eval` class method).

References

[R131] (page 1262), [R132] (page 1262), [R133] (page 1262)

Examples

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

inverse(*argindex=1*)

Returns the inverse of this function.

acos

class `diofant.functions.elementary.trigonometric.acos`

The inverse cosine function.

Returns the arc cosine of `x` (measured in radians).

See also:

diofant.functions.elementary.trigonometric.sin (page 312), *dio-*
fant.functions.elementary.trigonometric.csc (page 317), *dio-*
fant.functions.elementary.trigonometric.cos (page 313), *dio-*
fant.functions.elementary.trigonometric.sec (page 316), *dio-*
fant.functions.elementary.trigonometric.tan (page 314), *dio-*
fant.functions.elementary.trigonometric.cot (page 315), *dio-*
fant.functions.elementary.trigonometric.asin (page 317), *dio-*
fant.functions.elementary.trigonometric.acsc (page 322), *dio-*
fant.functions.elementary.trigonometric.asec (page 319), *dio-*
fant.functions.elementary.trigonometric.atan (page 320), *dio-*
fant.functions.elementary.trigonometric.acot (page 321), *dio-*
fant.functions.elementary.trigonometric.atan2 (page 323)

Notes

$\text{acos}(x)$ will evaluate automatically in the cases ∞ , $-\infty$, 0 , 1 , -1 .

$\text{acos}(zoo)$ evaluates to zoo (see note in `:py:class`diofant.functions.elementary.trigonometric.asec``)

References

[R134] (page 1262), [R135] (page 1262), [R136] (page 1262)

Examples

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

inverse(*argindex=1*)

Returns the inverse of this function.

asec

class `diofant.functions.elementary.trigonometric.asec`

The inverse secant function.

Returns the arc secant of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

$\text{asec}(x)$ will evaluate automatically in the cases ∞ , $-\infty$, 0 , 1 , -1 .

$\text{asec}(x)$ has branch cut in the interval $[-1, 1]$. For complex arguments, it can be defined [R140] (page 1262) as

$$\sec^{-1}(z) = -i * (\log(\sqrt{1 - z^2} + 1)/z)$$

At $x = 0$, for positive branch cut, the limit evaluates to ∞ . For negative branch cut, the limit

$$\lim_{z \rightarrow 0} -i * (\log(-\sqrt{1 - z^2} + 1)/z)$$

simplifies to $-i * \log(z/2 + O(z^3))$ which ultimately evaluates to ∞ .

As $\text{asec}(x) = \text{asec}(1/x)$, a similar argument can be given for $\text{acos}(x)$.

References

[R137] (page 1262), [R138] (page 1262), [R139] (page 1262), [R140] (page 1262)

Examples

```
>>> from diofant import asec, oo, pi
>>> asec(1)
0
>>> asec(-1)
pi
```

inverse(*argindex*=1)

Returns the inverse of this function.

atan

class `diofant.functions.elementary.trigonometric.atan`

The inverse tangent function.

Returns the arc tangent of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

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

References

[R141] (page 1262), [R142] (page 1262), [R143] (page 1262)

Examples

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

inverse(*argindex*=1)

Returns the inverse of this function.

acot

class diofant.functions.elementary.trigonometric.acot

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:

<i>diofant.functions.elementary.trigonometric.sin</i>	(page 312),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.csc</i>	(page 317),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.cos</i>	(page 313),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.sec</i>	(page 316),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.tan</i>	(page 314),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.cot</i>	(page 315),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.asin</i>	(page 317),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.acsc</i>	(page 322),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.acos</i>	(page 318),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.asec</i>	(page 319),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.atan</i>	(page 320),	<i>dio-</i>
<i>fant.functions.elementary.trigonometric.atan2</i>	(page 323)	

References

[R144] (page 1262), [R145] (page 1262), [R146] (page 1263)

inverse(*argindex*=1)

Returns the inverse of this function.

asec

class diofant.functions.elementary.trigonometric.asec

The inverse secant function.

Returns the arc secant of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

`asec(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1`.

`asec(x)` has branch cut in the interval `[-1, 1]`. For complex arguments, it can be defined [R150] (page 1263) as

$$\text{sec}^{-1}(z) = -i * (\log(\sqrt{1 - z^2} + 1)/z)$$

At `x = 0`, for positive branch cut, the limit evaluates to `zoo`. For negative branch cut, the limit

$$\lim_{z \rightarrow 0} -i * (\log(-\sqrt{1 - z^2} + 1)/z)$$

simplifies to `-i * log(z/2 + O(z3))` which ultimately evaluates to `zoo`.

As `asec(x) = asec(1/x)`, a similar argument can be given for `acos(x)`.

References

[R147] (page 1263), [R148] (page 1263), [R149] (page 1263), [R150] (page 1263)

Examples

```
>>> from diofant import asec, oo, pi
>>> asec(1)
0
>>> asec(-1)
pi
```

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

acsc

`class diofant.functions.elementary.trigonometric.acsc`
The inverse cosecant function.

Returns the arc cosecant of x (measured in radians).

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.acot</code>	(page 321),	<code>dio-</code>
<code>fant.functions.elementary.trigonometric.atan2</code>	(page 323)	

Notes

`acsc(x)` will evaluate automatically in the cases `oo`, `-oo`, `0`, `1`, `-1`.

References

[R151] (page 1263), [R152] (page 1263), [R153] (page 1263)

Examples

```
>>> from diofant import acsc, oo, pi
>>> acsc(1)
pi/2
>>> acsc(-1)
-pi/2
```

inverse(*argindex*=1)

Returns the inverse of this function.

atan2

class `diofant.functions.elementary.trigonometric.atan2`

The function `atan2(y,x)` computes `atan(y/x)` taking two arguments y and x . Signs of both y and x are considered to determine the appropriate quadrant of `atan(y/x)`. The range is $(-\pi, \pi]$. The complete definition reads as follows:

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases}$$

Attention: Note the role reversal of both arguments. The y -coordinate is the first argument and the x -coordinate the second.

See also:

<code>diofant.functions.elementary.trigonometric.sin</code>	(page 312),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.csc</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.cos</code>	(page 313),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.sec</code>	(page 316),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.tan</code>	(page 314),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.cot</code>	(page 315),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asin</code>	(page 317),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acsc</code>	(page 322),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acos</code>	(page 318),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.asec</code>	(page 319),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.atan</code>	(page 320),	<code>dio-</code>
<code>diofant.functions.elementary.trigonometric.acot</code>	(page 321)	

References

[R154] (page 1263), [R155] (page 1263), [R156] (page 1263)

Examples

Going counter-clock wise around the origin we find the following angles:

```
>>> from diofant import atan2
>>> atan2(0, 1)
0
>>> atan2(1, 1)
pi/4
>>> atan2(1, 0)
pi/2
>>> atan2(1, -1)
3*pi/4
>>> atan2(0, -1)
pi
>>> atan2(-1, -1)
-3*pi/4
>>> atan2(-1, 0)
-pi/2
>>> atan2(-1, 1)
-pi/4
```

which are all correct. Compare this to the results of the ordinary `atan` function for the point $(x, y) = (-1, 1)$

```
>>> from diofant import atan, S
>>> 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:


```
>>> from diofant import diff
>>> from diofant.abc import x, y
>>> diff(atan2(y, x), x)
-y/(x**2 + y**2)
```

```
>>> diff(atan2(y, x), y)
x/(x**2 + y**2)
```

We can express the `atan2` function in terms of complex logarithms:

```
>>> from diofant import log
>>> atan2(y, x).rewrite(log)
-I*log((x + I*y)/sqrt(x**2 + y**2))
```

and in terms of (`atan`):

```
>>> from diofant import atan
>>> atan2(y, x).rewrite(atan)
2*atan(y/(x + sqrt(x**2 + y**2)))
```

but note that this form is undefined on the negative real axis.

3.7.6 diofant.functions.elementary.hyperbolic

3.7.7 Hyperbolic Functions

HyperbolicFunction

class `diofant.functions.elementary.hyperbolic.HyperbolicFunction`

Base class for hyperbolic functions.

See also:

`diofant.functions.elementary.hyperbolic.sinh` (page 325), `dio-`
`fant.functions.elementary.hyperbolic.cosh` (page 326), `dio-`
`fant.functions.elementary.hyperbolic.tanh` (page 326), `dio-`
`fant.functions.elementary.hyperbolic.coth` (page 326)

`sinh`

class `diofant.functions.elementary.hyperbolic.sinh`

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 326), `dio-`
`fant.functions.elementary.hyperbolic.tanh` (page 326), `dio-`
`fant.functions.elementary.hyperbolic.asinh` (page 327)

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`

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 325), `dio-`
`fant.functions.elementary.hyperbolic.tanh` (page 326), `dio-`
`fant.functions.elementary.hyperbolic.acosh` (page 327)

tanh

class `diofant.functions.elementary.hyperbolic.tanh`

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 325), `dio-`
`fant.functions.elementary.hyperbolic.cosh` (page 326), `dio-`
`fant.functions.elementary.hyperbolic.atanh` (page 328)

inverse(*argindex*=1)

Returns the inverse of this function.

coth

class `diofant.functions.elementary.hyperbolic.coth`

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

• `coth(x)` -> Returns the hyperbolic cotangent of *x*

inverse(*argindex*=1)

Returns the inverse of this function.

sech

class `diofant.functions.elementary.hyperbolic.sech`

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 325), *dio-*
fant.functions.elementary.hyperbolic.cosh (page 326), *dio-*
fant.functions.elementary.hyperbolic.tanh (page 326), *dio-*
fant.functions.elementary.hyperbolic.coth (page 326), *dio-*
fant.functions.elementary.hyperbolic.csch (page 327), *dio-*
fant.functions.elementary.hyperbolic.asinh (page 327), *dio-*
fant.functions.elementary.hyperbolic.acosh (page 327)

csch

class `diofant.functions.elementary.hyperbolic.csch`

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 325), *dio-*
fant.functions.elementary.hyperbolic.cosh (page 326), *dio-*
fant.functions.elementary.hyperbolic.tanh (page 326), *dio-*
fant.functions.elementary.hyperbolic.sech (page 326), *dio-*
fant.functions.elementary.hyperbolic.asinh (page 327), *dio-*
fant.functions.elementary.hyperbolic.acosh (page 327)

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

3.7.8 Hyperbolic Inverses

asinh

class `diofant.functions.elementary.hyperbolic.asinh`

The inverse hyperbolic sine function.

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

See also:

diofant.functions.elementary.hyperbolic.cosh (page 326), *dio-*
fant.functions.elementary.hyperbolic.tanh (page 326), *dio-*
fant.functions.elementary.hyperbolic.sinh (page 325)

inverse(*argindex=1*)

Returns the inverse of this function.

acosh

class `diofant.functions.elementary.hyperbolic.acosh`

The inverse hyperbolic cosine function.

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

See also:

diofant.functions.elementary.hyperbolic.asinh (page 327), *dio-*
fant.functions.elementary.hyperbolic.atanh (page 328), *dio-*
fant.functions.elementary.hyperbolic.cosh (page 326)

inverse(*argindex=1*)

Returns the inverse of this function.

atanh

class *diofant.functions.elementary.hyperbolic.atanh*

The inverse hyperbolic tangent function.

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

See also:

diofant.functions.elementary.hyperbolic.asinh (page 327), *dio-*
fant.functions.elementary.hyperbolic.acosh (page 327), *dio-*
fant.functions.elementary.hyperbolic.tanh (page 326)

inverse(*argindex=1*)

Returns the inverse of this function.

acoth

class *diofant.functions.elementary.hyperbolic.acoth*

The inverse hyperbolic cotangent function.

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

inverse(*argindex=1*)

Returns the inverse of this function.

3.7.9 *diofant.functions.elementary.integers*

ceiling

class *diofant.functions.elementary.integers.ceiling*

Ceiling is a univariate function which returns the smallest integer value not less than its argument. Ceiling function is generalized in this implementation to complex numbers.

See also:

diofant.functions.elementary.integers.floor (page 329)

References

[R157] (page 1263), [R158] (page 1263)

Examples

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

floor

class `diofant.functions.elementary.integers.floor`

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

See also:

[diofant.functions.elementary.integers.ceiling](#) (page 328)

References

[R159] (page 1263), [R160] (page 1263)

Examples

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

RoundFunction

class `diofant.functions.elementary.integers.RoundFunction`

The base class for rounding functions.

3.7.10 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 331)

exp_polar

class `diofant.functions.elementary.exponential.exp_polar`

Represent a ‘polar number’ (see `g-function` Sphinx documentation).

`exp_polar` represents the function $Exp : \mathbb{C} \rightarrow \mathcal{S}$, 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.

```
>>> from diofant import exp_polar, pi, I, exp
```

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 937), [`diofant.functions.elementary.complexes.polar`](#) (page 311), [`diofant.functions.elementary.complexes.periodic_argument`](#) (page 311), [`diofant.functions.elementary.complexes.principal_branch`](#) (page 312)

as_numer_denom()

Returns this with a positive exponent as a 2-tuple (a fraction).

Examples

```
>>> from diofant.functions import exp
>>> from diofant.abc import x
>>> exp(-x).as_numer_denom()
(1, E**x)
>>> exp(x).as_numer_denom()
(E**x, 1)
```

exp

Returns the exponent of the function.

inverse(*argindex=1*)

Returns the inverse function of $\exp(x)$.

LambertW

class diofant.functions.elementary.exponential.LambertW

The Lambert W function $W(z)$ is defined as the inverse function of $w \exp(w)$ [R161] (page 1263).

In other words, the value of $W(z)$ is such that $z = W(z) \exp(W(z))$ for any complex number z . The Lambert W function is a multivalued function with infinitely many branches $W_k(z)$, indexed by $k \in \mathbb{Z}$. Each branch gives a different solution w of the equation $z = w \exp(w)$.

The Lambert W function has two partially real branches: the principal branch ($k = 0$) is real for real $z > -1/e$, and the $k = -1$ branch is real for $-1/e < z < 0$. All branches except $k = 0$ have a logarithmic singularity at $z = 0$.

References

[R161] (page 1263)

Examples

```
>>> from diofant import LambertW
>>> LambertW(1.2)
0.635564016364870
>>> LambertW(1.2, -1).n()
-1.34747534407696 - 4.41624341514535*I
>>> LambertW(-1).is_extended_real
False
```

fdiff(*argindex=1*)

Return the first derivative of this function.

log

class diofant.functions.elementary.exponential.Log

The natural logarithm function $\ln(x)$ or $\log(x)$. Logarithms are taken with the natural base, e . To get a logarithm of a different base b , use $\log(x, b)$, which is essentially short-hand for $\log(x)/\log(b)$.

See also:

diofant.functions.elementary.exponential.exp (page 330)

as_base_exp()

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

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

Returns this function as a complex coordinate.

Examples

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

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

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

3.7.11 diofant.functions.elementary.piecewise

ExprCondPair

class diofant.functions.elementary.piecewise.ExprCondPair

Represents an expression, condition pair.

cond
Returns the condition of this pair.

expr
Returns the expression of this pair.

free_symbols
Return the free symbols of this pair.

Piecewise

class diofant.functions.elementary.piecewise.Piecewise

Represents a piecewise function.

Usage:

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

- Each argument is a 2-tuple defining an expression and condition
- The conds are evaluated in turn returning the first that is True. If any of the evaluated conds are not determined explicitly False, e.g. $x < 1$, the function is returned in symbolic form.
- If the function is evaluated at a place where all conditions are False, a ValueError exception will be raised.
- Pairs where the cond is explicitly False, will be removed.

See also:

[*diofant.functions.elementary.piecewise.piecewise_fold*](#) (page 333)

Examples

```
>>> from diofant import Piecewise, log
>>> from diofant.abc import x
>>> f = x**2
>>> g = log(x)
>>> p = Piecewise( (0, x<-1), (f, x<=1), (g, True))
>>> p.subs(x,1)
1
>>> p.subs(x,5)
log(5)
```

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.

See also:

[*diofant.functions.elementary.piecewise.Piecewise*](#) (page 332)

Examples

```
>>> from diofant import Piecewise, piecewise_fold, Integer
>>> from diofant.abc import x
>>> p = Piecewise((x, x < 1), (1, Integer(1) <= x))
>>> piecewise_fold(x*p)
Piecewise((x**2, x < 1), (x, 1 <= x))
```

3.7.12 diofant.functions.elementary.miscellaneous

IdentityFunction

class `diofant.functions.elementary.miscellaneous.IdentityFunction`

The identity function

Examples

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

Min

class `diofant.functions.elementary.miscellaneous.Min`

Return, if possible, the minimum value of the list.

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

See also:

[diofant.functions.elementary.miscellaneous.Max](#) (page 334) find maximum values

Examples

```
>>> from diofant import Min, Symbol, oo
```

```
>>> from diofant.abc import x, y
>>> 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)
```

Max

class `diofant.functions.elementary.miscellaneous.Max`

Return, if possible, the maximum value of the list.

When number of arguments is equal one, then return this argument.

When number of arguments is equal two, then return, if possible, the value from (a, b) that is \geq the other.

In common case, when the length of list greater than 2, the task is more complicated. Return only the arguments, which are greater than others, if it is possible to determine directional relation.

If is not possible to determine such a relation, return a partially evaluated result.

Assumptions are used to make the decision too.

Also, only comparable arguments are permitted.

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

See also:

[diofant.functions.elementary.miscellaneous.Min](#) (page 333) find minimum values

Notes

The task can be considered as searching of supremums in the directed complete partial orders [R162] (page 1263).

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. ∞ . 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

[R162] (page 1263), [R163] (page 1263)

Examples

```
>>> from diofant import Max, Symbol, oo
```

```
>>> from diofant.abc import x, y, z
>>> 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
```

root

`diofant.functions.elementary.miscellaneous.root(x, n, k)` → Returns the k -th n -th root of x , defaulting to the principle root ($k=0$).

See also:

diofant.polys.rootoftools.RootOf (page 734), *diofant.core.power.integer_nthroot* (page 103), *diofant.functions.elementary.miscellaneous.sqrt* (page 338), *diofant.functions.elementary.miscellaneous.real_root* (page 337)

References

- http://en.wikipedia.org/wiki/Square_root
- http://en.wikipedia.org/wiki/Real_root
- http://en.wikipedia.org/wiki/Root_of_unity
- http://en.wikipedia.org/wiki/Principal_value
- <http://mathworld.wolfram.com/CubeRoot.html>

Examples

```
>>> from diofant import root, Rational
>>> from diofant.abc import x, n
```

```
>>> root(x, 2)
sqrt(x)
```

```
>>> root(x, 3)
x**(1/3)
```

```
>>> root(x, n)
x**(1/n)
```

```
>>> root(x, -Rational(2, 3))
x**(-3/2)
```

To get the k-th n-th root, specify k:

```
>>> root(-2, 3, 2)
-(-1)**(2/3)*2**(1/3)
```

To get all n n-th roots you can use the `RootOf` function. The following examples show the roots of unity for n equal 2, 3 and 4:

```
>>> from diofant import RootOf, I
```

```
>>> [ RootOf(x**2 - 1, i) for i in range(2) ]
[-1, 1]
```

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

```
>>> [ RootOf(x**4 - 1, i) for i in range(4) ]
[-1, 1, -I, I]
```

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*(-1)**(1/3)
```

The `real_root` function can be used to either make the principle result real (or simply to return the real root directly):

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

Alternatively, the $n//2$ -th n -th root of a negative number can be computed with `root`:

```
>>> root(-32, 5, 5//2)
-2
```

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 $(-n)**(1/\text{odd})$ will be changed to $-n**(1/\text{odd})$; this will only create a real root of a principle root - the presence of other factors may cause the result to not be real.

See also:

[diofant.polys.rootoftools.RootOf](#) (page 734), [diofant.core.power.integer_nthroot](#) (page 103), [diofant.functions.elementary.miscellaneous.root](#) (page 335), [diofant.functions.elementary.miscellaneous.sqrt](#) (page 338)

Examples

```
>>> from diofant import root, real_root, Rational
>>> from diofant.abc import x, n
```

```
>>> real_root(-8, 3)
-2
>>> root(-8, 3)
2*(-1)**(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*(-1)**(2/3)
>>> real_root(_)
-2*(-1)**(2/3)
```

sqrt

`diofant.functions.elementary.miscellaneous.sqrt`(*arg*, ****kwargs**)

The square root function

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

See also:

[`diofant.polys.rootoftools.RootOf`](#) (page 734), [`diofant.functions.elementary.miscellaneous.r`](#)
(page 335), [`diofant.functions.elementary.miscellaneous.real_root`](#) (page 337)

References

[R164] (page 1263), [R165] (page 1263)

Examples

```
>>> from diofant import sqrt, Symbol
>>> x = Symbol('x')
```

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

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

Note that `sqrt(x**2)` does not simplify to *x*.

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

This is because the two are not equal to each other in general. For example, consider *x* == -1:

```
>>> from diofant import Eq
>>> Eq(sqrt(x**2), x).subs(x, -1)
false
```

This is because `sqrt` computes the principal square root, so the square may put the argument in a different branch. This identity does hold if *x* is positive:

```
>>> y = Symbol('y', positive=True)
>>> sqrt(y**2)
y
```

You can force this simplification by using the `powdenest()` function with the `force` option set to `True`:

```
>>> from diofant import powdenest
>>> sqrt(x**2)
sqrt(x**2)
>>> powdenest(sqrt(x**2), force=True)
x
```

To get both branches of the square root you can use the RootOf function:

```
>>> from diofant import RootOf
```

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

3.7.13 Combinatorial

This module implements various combinatorial functions.

bell

class diofant.functions.combinatorial.numbers.bell

Bell numbers / Bell polynomials

The Bell numbers satisfy $B_0 = 1$ and

$$B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k.$$

They are also given by:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

The Bell polynomials are given by $B_0(x) = 1$ and

$$B_n(x) = x \sum_{k=1}^{n-1} \binom{n-1}{k-1} B_{k-1}(x).$$

The second kind of Bell polynomials (are sometimes called “partial” Bell polynomials or incomplete Bell polynomials) are defined as

$$B_{n,k}(x_1, x_2, x_{n-k+1}) = \sum_{\substack{j_1+j_2+j_3+\dots+k \\ j_1+2j_2+3j_3+\dots=n}} \frac{n!}{j_1!j_2!j_3+\dots+k!} \left(\frac{x_1}{1!}\right)^{j_1} \left(\frac{x_2}{2!}\right)^{j_2} \left(\frac{x_{n-k+1}}{(n-k+1)!}\right)^{j_{n-k+1}}.$$

- 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, $B_{n,k}(x_1, x_2, , x_{n-k+1})$.

See also:

diofant.functions.combinatorial.numbers.bernoulli (page 340), *dio-*
fant.functions.combinatorial.numbers.catalan (page 343), *dio-*
fant.functions.combinatorial.numbers.euler (page 344), *dio-*
fant.functions.combinatorial.numbers.fibonacci (page 348), *dio-*
fant.functions.combinatorial.numbers.harmonic (page 349), *dio-*
fant.functions.combinatorial.numbers.lucas (page 351)

Notes

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

References

[R80] (page 1263), [R81] (page 1263), [R82] (page 1263)

Examples

```
>>> from diofant import bell, Symbol, symbols
```

```
>>> [bell(n) for n in range(11)]
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975]
>>> bell(30)
846749014511809332450147
>>> bell(4, Symbol('t'))
t**4 + 6*t**3 + 7*t**2 + t
>>> bell(6, 2, symbols('x:6')[1:])
6*x1*x5 + 15*x2*x4 + 10*x3**2
```

bernoulli

class diofant.functions.combinatorial.numbers.bernoulli

Bernoulli numbers / Bernoulli polynomials

The Bernoulli numbers are a sequence of rational numbers defined by $B_0 = 1$ and the recursive relation ($n > 0$):

$$0 = \sum_{k=0}^n \binom{n}{k} B_k = 0$$

They are also commonly defined by their exponential generating function, which is $x/(\exp(x) - 1)$. For odd indices > 1 , the Bernoulli numbers are zero.

The Bernoulli polynomials satisfy the analogous formula:

$$B_n(x) = \sum_{k=0}^n \binom{n}{k} B_k x^{n-k}$$

Bernoulli numbers and Bernoulli polynomials are related as $B_n(0) = B_n$.

We compute Bernoulli numbers using Ramanujan's formula:

$$B_n = (A(n) - S(n)) / \binom{n+3}{n}$$

where $A(n) = (n+3)/3$ when $n = 0$ or $2 \pmod{6}$, $A(n) = -(n+3)/6$ when $n = 4 \pmod{6}$, and:

$$S(n) = \sum_{k=1}^{\lfloor n/6 \rfloor} \binom{n+3}{n-6*k} * B_{n-6*k}$$

This formula is similar to the sum given in the definition, but cuts 2/3 of the terms. For Bernoulli polynomials, we use the formula in the definition.

- `bernoulli(n)` gives the nth Bernoulli number, B_n
- `bernoulli(n, x)` gives the nth Bernoulli polynomial in x , $B_n(x)$

See also:

diofant.functions.combinatorial.numbers.bell	(page 339),	dio-
fant.functions.combinatorial.numbers.catalan	(page 343),	dio-
fant.functions.combinatorial.numbers.euler	(page 344),	dio-
fant.functions.combinatorial.numbers.fibonacci	(page 348),	dio-
fant.functions.combinatorial.numbers.harmonic	(page 349),	dio-
fant.functions.combinatorial.numbers.lucas	(page 351)	

References

[R83] (page 1263), [R84] (page 1263), [R85] (page 1263), [R86] (page 1263)

Examples

```
>>> from diofant import bernoulli
```

```
>>> [bernoulli(n) for n in range(11)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66]
>>> bernoulli(1000001)
0
```

binomial

class diofant.functions.combinatorial.factorials.binomial

Implementation of the binomial coefficient. It can be defined in two ways depending on its desired interpretation:

$$C(n,k) = n!/(k!(n-k)!) \text{ or } C(n, k) = \text{ff}(n, k)/k!$$

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

```
>>> from diofant import Symbol, Rational, binomial, expand_func
>>> n = Symbol('n', integer=True, positive=True)
```

```
>>> binomial(15, 8)
6435
```

```
>>> binomial(n, -1)
0
```

Rows of Pascal's triangle can be generated with the binomial function:

```
>>> for N in range(8):
...     [ binomial(N, i) for i in range(N + 1)]
...
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
```

As can a given diagonal, e.g. the 4th diagonal:

```
>>> N = -4
>>> [ binomial(N, i) for i in range(1 - N)]
[1, -4, 10, -20, 35]
```

```
>>> binomial(Rational(5, 4), 3)
-5/128
>>> binomial(Rational(-5, 4), 3)
-195/128
```

```
>>> binomial(n, 3)
binomial(n, 3)
```

```
>>> binomial(n, 3).expand(func=True)
n**3/6 - n**2/2 + n/3
```

```
>>> expand_func(binomial(n, 3))
n*(n - 2)*(n - 1)/6
```

catalan

class diofant.functions.combinatorial.numbers.catalan

Catalan numbers

The n-th catalan number is given by:

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

- `catalan(n)` gives the n-th Catalan number, `C_n`

See also:

`diofant.functions.combinatorial.numbers.bell` (page 339), `diofant.functions.combinatorial.numbers.bernoulli` (page 340), `diofant.functions.combinatorial.numbers.euler` (page 344), `diofant.functions.combinatorial.numbers.fibonacci` (page 348), `diofant.functions.combinatorial.numbers.harmonic` (page 349), `diofant.functions.combinatorial.numbers.lucas` (page 351), `diofant.functions.combinatorial.factorials.binomial` (page 341)

References

[R87] (page 1263), [R88] (page 1263), [R89] (page 1263), [R90] (page 1263)

Examples

```
>>> from diofant import (Symbol, binomial, gamma, hyper, polygamma,
...                       catalan, diff, combsimp, Rational, I)
```

```
>>> [ catalan(i) for i in range(1,10) ]
[1, 2, 5, 14, 42, 132, 429, 1430, 4862]
```

```
>>> n = Symbol("n", integer=True)
```

```
>>> catalan(n)
catalan(n)
```

Catalan numbers can be transformed into several other, identical expressions involving other mathematical functions

```
>>> catalan(n).rewrite(binomial)
binomial(2*n, n)/(n + 1)
```

```
>>> catalan(n).rewrite(gamma)
4**n*gamma(n + 1/2)/(sqrt(pi)*gamma(n + 2))
```

```
>>> catalan(n).rewrite(hyper)
hyper((-n + 1, -n), (2,), 1)
```

For some non-integer values of n we can get closed form expressions by rewriting in terms of gamma functions:

```
>>> catalan(Rational(1,2)).rewrite(gamma)
8/(3*pi)
```

We can differentiate the Catalan numbers $C(n)$ interpreted as a continuous real function in n :

```
>>> diff(catalan(n), n)
(polygamma(0, n + 1/2) - polygamma(0, n + 2) + log(4))*catalan(n)
```

As a more advanced example consider the following ratio between consecutive numbers:

```
>>> combsimp((catalan(n + 1)/catalan(n)).rewrite(binomial))
2*(2*n + 1)/(n + 2)
```

The Catalan numbers can be generalized to complex numbers:

```
>>> catalan(I).rewrite(gamma)
4**I*gamma(1/2 + I)/(sqrt(pi)*gamma(2 + I))
```

and evaluated with arbitrary precision:

```
>>> catalan(I).evalf(20)
0.39764993382373624267 - 0.020884341620842555705*I
```

euler

class diofant.functions.combinatorial.numbers.euler

Euler numbers

The euler numbers are given by:

$$E_{2n} = \sum_{k=1}^{2n+1} \sum_{j=0}^k \frac{(-1)^j}{2^k} \binom{k}{j} \binom{2n+1}{k-2j}$$

$$E_{2n+1} = 0$$

- `euler(n)` gives the n -th Euler number, E_n

See also:

[diofant.functions.combinatorial.numbers.bell](#) (page 339), [diofant.functions.combinatorial.numbers.bernoulli](#) (page 340), [diofant.functions.combinatorial.numbers.fibonacci](#) (page 348), [diofant.functions.combinatorial.numbers.harmonic](#) (page 349), [diofant.functions.combinatorial.numbers.lucas](#) (page 351)

References

[R91] (page 1263), [R92] (page 1263), [R93] (page 1263), [R94] (page 1263)

Examples

```
>>> from diofant import Symbol
>>> from diofant.functions import euler
>>> [euler(n) for n in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
>>> n = Symbol("n")
>>> euler(n+2*n)
euler(3*n)
```

factorial

class diofant.functions.combinatorial.factorials.factorial

Implementation of factorial function over nonnegative integers. By convention (consistent with the gamma function and the binomial coefficients), factorial of a negative integer is complex infinity.

The factorial is very important in combinatorics where it gives the number of ways in which n objects can be permuted. It also arises in calculus, probability, number theory, etc.

There is strict relation of factorial with gamma function. In fact $n! = \text{gamma}(n+1)$ for nonnegative integers. Rewrite of this kind is very useful in case of combinatorial simplification.

Computation of the factorial is done using two algorithms. For small arguments naive product is evaluated. However for bigger input algorithm Prime-Swing is used. It is the fastest algorithm known and computes $n!$ via prime factorization of special class of numbers, called here the 'Swing Numbers'.

See also:

[diofant.functions.combinatorial.factorials.factorial2](#) (page 347), [diofant.functions.combinatorial.factorials.RisingFactorial](#) (page 352), [diofant.functions.combinatorial.factorials.FallingFactorial](#) (page 347)

Examples

```
>>> from diofant import Symbol, factorial, Rational
>>> n = Symbol('n', integer=True)
```

```
>>> factorial(0)
1
```

```
>>> factorial(7)
5040
```

```
>>> factorial(-2)
zoo
```

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

```
>>> factorial(2*n)
factorial(2*n)
```

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

subfactorial

class diofant.functions.combinatorial.factorials.subfactorial

The subfactorial counts the derangements of n items and is defined for non-negative integers as:

$$!n = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{for } n = 1 \\ (n - 1) * (!n - 1 + !n - 2) & \text{for } n > 1 \end{cases}$$

It can also be written as $\text{int}(\text{round}(n!/\text{exp}(1)))$ but the recursive definition with caching is implemented for this function.

An interesting analytic expression is the following [\[R96\]](#) (page 1263)

$$!x = \Gamma(x + 1, -1)/e$$

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.

See also:

[diofant.functions.combinatorial.factorials.factorial](#) (page 345), [diofant.utilities.iterables.generate_derangements](#) (page 1147), [diofant.functions.special.gamma_functions.uppergamma](#) (page 364)

References

[\[R95\]](#) (page 1263), [\[R96\]](#) (page 1263)

Examples

```
>>> from diofant import subfactorial
>>> from diofant.abc import n
>>> subfactorial(n + 1)
subfactorial(n + 1)
>>> subfactorial(5)
44
```

factorial2 / double factorial

class diofant.functions.combinatorial.factorials.factorial2

The double factorial $n!!$, not to be confused with $(n)!$

The double factorial is defined for nonnegative integers and for odd negative integers as:

$$n!! = \begin{cases} n*(n-2)*(n-4)*\dots*1 & \text{for } n \text{ positive odd} \\ n*(n-2)*(n-4)*\dots*2 & \text{for } n \text{ positive even} \\ 1 & \text{for } n = 0 \\ (n+2)!! / (n+2) & \text{for } n \text{ negative odd} \end{cases}$$

See also:

diofant.functions.combinatorial.factorials.factorial (page 345), *diofant.functions.combinatorial.factorials.RisingFactorial* (page 352), *diofant.functions.combinatorial.factorials.FallingFactorial* (page 347)

References

[R97] (page 1263)

Examples

```
>>> from diofant import factorial2
>>> from diofant.abc import n
```

```
>>> factorial2(n + 1)
factorial2(n + 1)
>>> factorial2(5)
15
>>> factorial2(-1)
1
>>> factorial2(-5)
1/3
```

FallingFactorial

class diofant.functions.combinatorial.factorials.FallingFactorial

Falling factorial (related to rising factorial) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by

$$ff(x, k) = x * (x-1) * \dots * (x - k+1)$$

where 'x' can be arbitrary expression and 'k' is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit <http://mathworld.wolfram.com/FallingFactorial.html> page.

```
>>> from diofant import ff
>>> from diofant.abc import x
```

```
>>> ff(x, 0)
1
```

```
>>> ff(5, 5)
120
```

```
>>> ff(x, 5) == x*(x-1)*(x-2)*(x-3)*(x-4)
True
```

See also:

diofant.functions.combinatorial.factorials.factorial (page 345), *diofant.functions.combinatorial.factorials.factorial2* (page 347), *diofant.functions.combinatorial.factorials.RisingFactorial* (page 352)

fibonacci

class `diofant.functions.combinatorial.numbers.fibonacci`

Fibonacci numbers / Fibonacci polynomials

The Fibonacci numbers are the integer sequence defined by the initial terms $F_0 = 0$, $F_1 = 1$ and the two-term recurrence relation $F_n = F_{n-1} + F_{n-2}$. This definition extended to arbitrary real and complex arguments using the formula

$$F_z = \frac{\phi^z - \cos(\pi z)\phi^{-z}}{\sqrt{5}}$$

The Fibonacci polynomials are defined by $F_1(x) = 1$, $F_2(x) = x$, and $F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$ for $n > 2$. For all positive integers n , $F_n(1) = F_n$.

- `fibonacci(n)` gives the n th Fibonacci number, F_n
- `fibonacci(n, x)` gives the n th Fibonacci polynomial in x , $F_n(x)$

See also:

diofant.functions.combinatorial.numbers.bell (page 339), *diofant.functions.combinatorial.numbers.bernoulli* (page 340), *diofant.functions.combinatorial.numbers.catalan* (page 343), *diofant.functions.combinatorial.numbers.euler* (page 344), *diofant.functions.combinatorial.numbers.harmonic* (page 349), *diofant.functions.combinatorial.numbers.lucas* (page 351)

References

[R98] (page 1264), [R99] (page 1264)

Examples

```
>>> from diofant import fibonacci, Symbol
```



```
>>> [fibonacci(x) for x in range(11)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> fibonacci(5, Symbol('t'))
t**4 + 3*t**2 + 1
```

harmonic

class diofant.functions.combinatorial.numbers.harmonic

Harmonic numbers

The n th harmonic number is given by $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$.

More generally:

$$H_{n,m} = \sum_{k=1}^n \frac{1}{k^m}$$

As $n \rightarrow \infty$, $H_{n,m} \rightarrow \zeta(m)$, the Riemann zeta function.

- `harmonic(n)` gives the n th harmonic number, H_n
- `harmonic(n,m)` gives the n th generalized harmonic number of order m , $H_{n,m}$, where `harmonic(n) == harmonic(n,1)`

See also:

<i>diofant.functions.combinatorial.numbers.bell</i>	(page 339),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.bernoulli</i>	(page 340),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.catalan</i>	(page 343),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.euler</i>	(page 344),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.fibonacci</i>	(page 348),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.lucas</i>	(page 351)	

References

[R100] (page 1264), [R101] (page 1264), [R102] (page 1264)

Examples

```
>>> from diofant import harmonic, oo
```

```
>>> [harmonic(n) for n in range(6)]
[0, 1, 3/2, 11/6, 25/12, 137/60]
>>> [harmonic(n, 2) for n in range(6)]
[0, 1, 5/4, 49/36, 205/144, 5269/3600]
>>> harmonic(oo, 2)
pi**2/6
```

```
>>> from diofant import Symbol, Sum
>>> n = Symbol("n")
```

```
>>> harmonic(n).rewrite(Sum)
Sum(1/_k, (_k, 1, n))
```

We can evaluate harmonic numbers for all integral and positive rational arguments:

```
>>> from diofant import S, expand_func, simplify
>>> harmonic(8)
761/280
>>> harmonic(11)
83711/27720
```

```
>>> H = harmonic(1/Integer(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(25/Integer(7))
>>> He = simplify(expand_func(H).doit())
>>> He
log(sin(pi/7)**(-2*cos(pi/7))*sin(2*pi/7)**(2*cos(16*pi/7))*cos(pi/14)**(-
↳2*sin(pi/14))/14)
+ pi*tan(pi/14)/2 + 30247/9900
>>> He.n(40)
1.983697455232980674869851942390639915940
>>> harmonic(25/Integer(7)).n(40)
1.983697455232980674869851942390639915940
```

We can rewrite harmonic numbers in terms of polygamma functions:

```
>>> from diofant import digamma, polygamma
>>> m = Symbol("m")
```

```
>>> harmonic(n).rewrite(digamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n).rewrite(polygamma)
polygamma(0, n + 1) + EulerGamma
```

```
>>> harmonic(n,3).rewrite(polygamma)
polygamma(2, n + 1)/2 - polygamma(2, 1)/2
```

```
>>> harmonic(n,m).rewrite(polygamma)
(-1)**m*(polygamma(m - 1, 1) - polygamma(m - 1, n + 1))/factorial(m - 1)
```

Integer offsets in the argument can be pulled out:

```
>>> from diofant import expand_func
```

```
>>> expand_func(harmonic(n+4))
harmonic(n) + 1/(n + 4) + 1/(n + 3) + 1/(n + 2) + 1/(n + 1)
```

```
>>> expand_func(harmonic(n-4))
harmonic(n) - 1/(n - 1) - 1/(n - 2) - 1/(n - 3) - 1/n
```

Some limits can be computed as well:

```
>>> from diofant import limit, oo
```

```
>>> limit(harmonic(n), n, oo)
oo
```

```
>>> limit(harmonic(n, 2), n, oo)
pi**2/6
```

```
>>> limit(harmonic(n, 3), n, oo)
-polygamma(2, 1)/2
```

However we can not compute the general relation yet:

```
>>> limit(harmonic(n, m), n, oo)
harmonic(oo, m)
```

which equals $\zeta(m)$ for $m > 1$.

lucas

class diofant.functions.combinatorial.numbers.Lucas

Lucas numbers

Lucas numbers satisfy a recurrence relation similar to that of the Fibonacci sequence, in which each term is the sum of the preceding two. They are generated by choosing the initial values $L_0 = 2$ and $L_1 = 1$.

- `lucas(n)` gives the n th Lucas number

See also:

<i>diofant.functions.combinatorial.numbers.bell</i>	(page 339),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.bernoulli</i>	(page 340),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.catalan</i>	(page 343),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.euler</i>	(page 344),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.fibonacci</i>	(page 348),	<i>dio-</i>
<i>fant.functions.combinatorial.numbers.harmonic</i>	(page 349)	

References

[R103] (page 1264), [R104] (page 1264)

Examples

```
>>> from diofant import lucas
```

```
>>> [lucas(x) for x in range(11)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123]
```

MultiFactorial

class diofant.functions.combinatorial.factorials.MultiFactorial

RisingFactorial

class diofant.functions.combinatorial.factorials.RisingFactorial

Rising factorial (also called Pochhammer symbol) is a double valued function arising in concrete mathematics, hypergeometric functions and series expansions. It is defined by:

$$\text{rf}(x, k) = x * (x+1) * \dots * (x + k-1)$$

where 'x' can be arbitrary expression and 'k' is an integer. For more information check "Concrete mathematics" by Graham, pp. 66 or visit <http://mathworld.wolfram.com/RisingFactorial.html> page.

See also:

diofant.functions.combinatorial.factorials.factorial (page 345), *diofant.functions.combinatorial.factorials.factorial2* (page 347), *diofant.functions.combinatorial.factorials.FallingFactorial* (page 347)

Examples

```
>>> from diofant import rf
>>> from diofant.abc import x
```

```
>>> rf(x, 0)
1
```

```
>>> rf(1, 5)
120
```

```
>>> rf(x, 5) == x*(1 + x)*(2 + x)*(3 + x)*(4 + x)
True
```

stirling

`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:

$$\begin{aligned} \{0\} &= 1; & \{n\} &= \{n\} = 0; & \{n+1\} &= j*\{n\} + \{n\} \\ \{0\} & & \{0\} & \{k\} & \{k\} & \{k-1\} \end{aligned}$$

where j is:: n for Stirling numbers of the first kind $-n$ for signed Stirling numbers of the first kind k for Stirling numbers of the second kind

The first kind of Stirling number counts the number of permutations of n distinct items that have k cycles; the second kind counts the ways in which n distinct items can be partitioned into k parts. If d is given, the "reduced Stirling number of the second kind" is returned: $S^{\{d\}}(n, k) = S(n - d + 1, k - d + 1)$ with $n \geq k \geq d$. (This counts the ways to partition n consecutive integers into k groups with no pairwise difference less than d . See example below.)

To obtain the signed Stirling numbers of the first kind, use keyword `signed=True`. Using this keyword automatically sets `kind` to 1.

See also:

[diofant.utilities.iterables.multiset_partitions](#) (page 1153)

References

[R105] (page 1264), [R106] (page 1264)

Examples

```
>>> from diofant.functions.combinatorial.numbers import stirling, bell
>>> from diofant.combinatorics import Permutation
>>> from diofant.utilities.iterables import (multiset_partitions,
...                                         permutations, subsets)
```

First kind (unsigned by default):

```
>>> [stirling(6, i, kind=1) for i in range(7)]
[0, 120, 274, 225, 85, 15, 1]
>>> perms = list(permutations(range(4)))
>>> [sum(Permutation(p).cycles == i for p in perms) for i in range(5)]
[0, 6, 11, 6, 1]
>>> [stirling(4, i, kind=1) for i in range(5)]
[0, 6, 11, 6, 1]
```

First kind (signed):

```
>>> [stirling(4, i, signed=True) for i in range(5)]
[0, -6, 11, -6, 1]
```

Second kind:

```
>>> [stirling(10, i) for i in range(12)]
[0, 1, 511, 9330, 34105, 42525, 22827, 5880, 750, 45, 1, 0]
>>> sum(_) == bell(10)
True
>>> len(list(multiset_partitions(range(4), 2))) == stirling(4, 2)
True
```

Reduced second kind:

```
>>> from diofant import oo
>>> def delta(p):
...     if len(p) == 1:
...         return oo
...     return min(abs(i[0] - i[1]) for i in subsets(p, 2))
>>> parts = multiset_partitions(range(5), 3)
>>> d = 2
>>> sum(1 for p in parts if all(delta(i) >= d for i in p))
7
>>> stirling(5, 3, 2)
7
```

3.7.14 Enumeration

Three functions are available. Each of them attempts to efficiently compute a given combinatorial quantity for a given set or multiset which can be entered as an integer, sequence or multiset (dictionary with elements as keys and multiplicities as values). The *k* parameter indicates the number of elements to pick (or the number of partitions to make). When *k* is *None*, the sum of the enumeration for all *k* (from 0 through the number of items represented by *n*) is returned. A replacement parameter is recognized for combinations and permutations; this indicates that any item may appear with multiplicity as high as the number of items in the original set.

```
>>> from diofant.functions.combinatorial.numbers import nC, nP, nT
>>> items = 'baby'
```

`diofant.functions.combinatorial.numbers.nC(n, k=None, replacement=False)`

Return the number of combinations of *n* items taken *k* at a time.

Possible values for *n*: integer - set of length *n* sequence - converted to a multiset internally multiset - {element: multiplicity}

If *k* is *None* then the total of all combinations of length 0 through the number of items represented in *n* will be returned.

If *replacement* is *True* then a given item can appear more than once in the *k* items. (For example, for 'ab' sets of 2 would include 'aa', 'ab', and 'bb'.) The multiplicity of elements in *n* is ignored when *replacement* is *True* but the total number of elements is considered since no element can appear more times than the number of elements in *n*.

See also:

[*diofant.utilities.iterables.multiset_combinations*](#) (page 1152)

References

[R107] (page 1264), [R108] (page 1264)

Examples

```
>>> from diofant.functions.combinatorial.numbers import nC
>>> from diofant.utilities.iterables import multiset_combinations
>>> nC(3, 2)
3
>>> nC('abc', 2)
3
>>> nC('aab', 2)
2
```

When *replacement* is *True*, each item can have multiplicity equal to the length represented by *n*:

```
>>> nC('aabc', replacement=True)
35
>>> [len(list(multiset_combinations('aaaabbbbcccc', i))) for i in range(5)]
[1, 3, 6, 10, 15]
>>> sum(_)
35
```

If there are k items with multiplicities m_1, m_2, \dots, m_k then the total of all combinations of length 0 through k is the product, $(m_1 + 1) * (m_2 + 1) * \dots * (m_k + 1)$. When the multiplicity of each item is 1 (i.e., k unique items) then there are $2^{**}k$ combinations. For example, if there are 4 unique items, the total number of combinations is 16:

```
>>> sum(nC(4, i) for i in range(5))
16
```

`diofant.functions.combinatorial.numbers.nP(n, k=None, replacement=False)`
Return the number of permutations of n items taken k at a time.

Possible values for n:: integer - set of length n sequence - converted to a multiset internally multiset - {element: multiplicity}

If k is None then the total of all permutations of length 0 through the number of items represented by n will be returned.

If `replacement` is True then a given item can appear more than once in the k items. (For example, for 'ab' permutations of 2 would include 'aa', 'ab', 'ba' and 'bb'.) The multiplicity of elements in n is ignored when `replacement` is True but the total number of elements is considered since no element can appear more times than the number of elements in n .

See also:

[*diofant.utilities.iterables.multiset_permutations*](#) (page 1154)

References

[R109] (page 1264)

Examples

```
>>> from diofant.functions.combinatorial.numbers import nP
>>> from diofant.utilities.iterables import multiset_permutations, multiset
>>> nP(3, 2)
6
>>> nP('abc', 2) == nP(multiset('abc'), 2) == 6
True
>>> nP('aab', 2)
3
>>> nP([1, 2, 2], 2)
3
>>> [nP(3, i) for i in range(4)]
[1, 3, 6, 6]
>>> nP(3) == sum(_)
True
```

When `replacement` is True, each item can have multiplicity equal to the length represented by n :

```
>>> nP('aabc', replacement=True)
121
>>> [len(list(multiset_permutations('aaaabbbbcccc', i))) for i in range(5)]
[1, 3, 9, 27, 81]
>>> sum(_)
121
```

```
diofant.functions.combinatorial.numbers.nT(n, k=None)
```

Return the number of k-sized partitions of n items.

Possible values for n:: integer - n identical items sequence - converted to a multiset internally multiset - {element: multiplicity}

Note: the convention for nT is different than that of nC and nP in that here an integer indicates n *identical* items instead of a set of length n; this is in keeping with the partitions function which treats its integer-n input like a list of n 1s. One can use range(n) for n to indicate n distinct items.

If k is None then the total number of ways to partition the elements represented in n will be returned.

See also:

[diofant.utilities.iterables.partitions](#) (page 1155), [diofant.utilities.iterables.multiset_partitions](#) (page 1153)

References

[R110] (page 1264)

Examples

```
>>> from diofant.functions.combinatorial.numbers import nT
```

Partitions of the given multiset:

```
>>> [nT('aabbc', i) for i in range(1, 7)]
[1, 8, 11, 5, 1, 0]
>>> nT('aabbc') == sum(_)
True
```

```
>>> [nT("mississippi", i) for i in range(1, 12)]
[1, 74, 609, 1521, 1768, 1224, 579, 197, 50, 9, 1]
```

Partitions when all items are identical:

```
>>> [nT(5, i) for i in range(1, 6)]
[1, 2, 2, 1, 1]
>>> nT('1'*5) == sum(_)
True
```

When all items are different:

```
>>> [nT(range(5), i) for i in range(1, 6)]
[1, 15, 25, 10, 1]
>>> nT(range(5)) == sum(_)
True
```

Note that the integer for n indicates *identical* items for nT but indicates n *different* items for nC and nP.

3.7.15 Special

DiracDelta

class `diofant.functions.special.delta_functions.DiracDelta`

The DiracDelta function and its derivatives.

DiracDelta function has the following properties:

1. `diff(Heaviside(x), x) = DiracDelta(x)`
2. `integrate(DiracDelta(x-a)*f(x), (x, -oo, oo)) = f(a)` and `integrate(DiracDelta(x-a)*f(x), (x, a-e, a+e)) = f(a)`
3. `DiracDelta(x) = 0` for all $x \neq 0$
4. `DiracDelta(g(x)) = Sum_i(DiracDelta(x-x_i)/abs(g'(x_i)))` Where x_i -s are the roots of g

Derivatives of k-th order of DiracDelta have the following property:

5. `DiracDelta(x, k) = 0`, for all $x \neq 0$

See also:

`diofant.functions.special.delta_functions.Heaviside` (page 358),
`diofant.simplify.simplify.simplify` (page 922), `diofant.functions.special.delta_functions.DiracDelta.is_simple` (page 357),
`diofant.functions.special.tensor_functions.KroneckerDelta` (page 425)

References

[R166] (page 1264)

is_simple(*self*, x)

Tells whether the argument(`args[0]`) of DiracDelta is a linear expression in x.

x can be:

- a symbol

See also:

`diofant.simplify.simplify.simplify` (page 922), `diofant.functions.special.delta_functions.DiracDelta` (page 357)

Examples

```
>>> from diofant import DiracDelta, cos
>>> from diofant.abc import x, y
```

```
>>> DiracDelta(x*y).is_simple(x)
True
>>> DiracDelta(x*y).is_simple(y)
True
```

```
>>> DiracDelta(x**2+x-2).is_simple(x)
False
```

```
>>> DiracDelta(cos(x)).is_simple(x)
False
```

simplify(self, x)

Compute a simplified representation of the function using property number 4.

x can be:

- a symbol

See also:

diofant.functions.special.delta_functions.DiracDelta.is_simple
(page 357), *diofant.functions.special.delta_functions.DiracDelta*
(page 357)

Examples

```
>>> from diofant import DiracDelta
>>> from diofant.abc import x, y
```

```
>>> DiracDelta(x*y).simplify(x)
DiracDelta(x)/Abs(y)
>>> DiracDelta(x*y).simplify(y)
DiracDelta(y)/Abs(x)
```

```
>>> DiracDelta(x**2 + x - 2).simplify(x)
DiracDelta(x - 1)/3 + DiracDelta(x + 2)/3
```

Heaviside

class `diofant.functions.special.delta_functions.Heaviside`

Heaviside step function [\[R167\]](#) (page 1264)

$$H(x) = \begin{cases} 0, & x < 0 \\ 1/2, & x = 0 \\ 1, & x > 0 \end{cases}$$

See also:

diofant.functions.special.delta_functions.DiracDelta (page 357)

References

[\[R167\]](#) (page 1264)

Gamma, Beta and related Functions

class `diofant.functions.special.gamma_functions.gamma`

The gamma function

$$\Gamma(x) := \int_0^{\infty} t^{x-1} e^{-t} dt.$$

The gamma function implements the function which passes through the values of the factorial function, i.e. $\Gamma(n) = (n - 1)!$ when n is an integer. More general, $\Gamma(z)$ is defined in the whole complex plane except at the negative integers where there are simple poles.

See also:

lowergamma (page 365) Lower incomplete gamma function.

uppergamma (page 364) Upper incomplete gamma function.

polygamma (page 361) Polygamma function.

loggamma (page 360) Log Gamma function.

digamma (page 363) Digamma function.

trigamma (page 364) Trigamma function.

diofant.functions.special.beta_functions.beta (page 366) Euler Beta function.

References

[R168] (page 1264), [R169] (page 1264), [R170] (page 1264), [R171] (page 1264)

Examples

```
>>> from diofant import I, pi, oo, gamma, Rational
>>> from diofant.abc import x
```

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:

```
>>> from diofant import conjugate
>>> conjugate(gamma(x))
gamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from diofant import diff
>>> diff(gamma(x), x)
gamma(x)*polygamma(0, x)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(gamma(x), x, 0, 3)
1/x - EulerGamma + x*(EulerGamma**2/2 + pi**2/12) + x**2*(-EulerGamma*pi**2/12 +
↳ polygamma(2, 1)/6 - EulerGamma**3/6) + O(x**3)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> gamma(pi).evalf(40)
2.288037795340032417959588909060233922890
>>> gamma(1+I).evalf(20)
0.49801566811835604271 - 0.15494982830181068512*I
```

class `diofant.functions.special.gamma_functions.Loggamma`

The loggamma function implements the logarithm of the gamma function i.e, $\log\Gamma(x)$.

See also:

gamma (**page 358**) Gamma function.

lowergamma (**page 365**) Lower incomplete gamma function.

uppergamma (**page 364**) Upper incomplete gamma function.

polygamma (**page 361**) Polygamma function.

digamma (**page 363**) Digamma function.

trigamma (**page 364**) Trigamma function.

diofant.functions.special.beta_functions.beta (**page 366**) Euler Beta function.

References

[R172] (page 1264), [R173] (page 1264), [R174] (page 1264), [R175] (page 1264)

Examples

Several special values are known. For numerical integral arguments we have:

```
>>> from diofant import loggamma
>>> loggamma(-2)
oo
>>> loggamma(0)
oo
>>> loggamma(1)
0
>>> loggamma(2)
0
>>> loggamma(3)
log(2)
```

and for symbolic values:

```
>>> from diofant import Symbol
>>> n = Symbol("n", integer=True, positive=True)
>>> loggamma(n)
log(gamma(n))
>>> loggamma(-n)
oo
```

for half-integral values:

```
>>> from diofant import pi, Rational
>>> 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:

```
>>> from diofant import expand_func
>>> 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:

```
>>> from diofant import oo
>>> loggamma(oo)
oo
>>> loggamma(-oo)
zoo
```

The loggamma function obeys the mirror symmetry if $x \in \mathbb{C} \setminus \{-\infty, 0\}$:

```
>>> from diofant.abc import x
>>> from diofant import conjugate
>>> conjugate(loggamma(x))
loggamma(conjugate(x))
```

Differentiation with respect to x is supported:

```
>>> from diofant import diff
>>> diff(loggamma(x), x)
polygamma(0, x)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(loggamma(x), x, 0, 4)
-log(x) - EulerGamma*x + pi**2*x**2/12 + x**3*polygamma(2, 1)/6 + 0(x**4)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from diofant import I
>>> loggamma(5).evalf(30)
3.17805383034794561964694160130
>>> loggamma(I).evalf(20)
-0.65092319930185633889 - 1.8724366472624298171*I
```

class `diofant.functions.special.gamma_functions.polygamma`

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:

$$\psi^{(n)}(z) := \frac{d^{n+1}}{dz^{n+1}} \log \Gamma(z).$$

See also:

gamma (page 358) Gamma function.

lowergamma (page 365) Lower incomplete gamma function.

uppergamma (page 364) Upper incomplete gamma function.

loggamma (page 360) Log Gamma function.

digamma (page 363) Digamma function.

trigamma (page 364) Trigamma function.

diofant.functions.special.beta_functions.beta (page 366) Euler Beta function.

References

[R176] (page 1264), [R177] (page 1264), [R178] (page 1264), [R179] (page 1264)

Examples

Several special values are known:

```
>>> from diofant import S, polygamma, Integer
>>> polygamma(0, 1)
-EulerGamma
>>> polygamma(0, 1/Integer(2))
-2*log(2) - EulerGamma
>>> polygamma(0, 1/Integer(3))
-3*log(3)/2 - sqrt(3)*pi/6 - EulerGamma
>>> polygamma(0, 1/Integer(4))
-3*log(2) - pi/2 - EulerGamma
>>> polygamma(0, 2)
-EulerGamma + 1
>>> polygamma(0, 23)
-EulerGamma + 19093197/5173168
```

```
>>> from diofant import oo, I
>>> 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:

```

>>> from diofant import Symbol, diff
>>> x = Symbol("x")
>>> diff(polygamma(0, x), x)
polygamma(1, x)
>>> diff(polygamma(0, x), x, 2)
polygamma(2, x)
>>> diff(polygamma(0, x), x, 3)
polygamma(3, x)
>>> diff(polygamma(1, x), x)
polygamma(2, x)
>>> diff(polygamma(1, x), x, 2)
polygamma(3, x)
>>> diff(polygamma(2, x), x)
polygamma(3, x)
>>> diff(polygamma(2, x), x, 2)
polygamma(4, x)

```

```

>>> n = Symbol("n")
>>> diff(polygamma(n, x), x)
polygamma(n + 1, x)
>>> diff(polygamma(n, x), x, 2)
polygamma(n + 2, x)

```

We can rewrite polygamma functions in terms of harmonic numbers:

```

>>> from diofant import harmonic
>>> polygamma(0, x).rewrite(harmonic)
harmonic(x - 1) - EulerGamma
>>> polygamma(2, x).rewrite(harmonic)
2*harmonic(x - 1, 3) - 2*zeta(3)
>>> ni = Symbol("n", integer=True)
>>> polygamma(ni, x).rewrite(harmonic)
(-1)**(n + 1)*(-harmonic(x - 1, n + 1) + zeta(n + 1))*factorial(n)

```

`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 358) Gamma function.

`lowergamma` (page 365) Lower incomplete gamma function.

`uppergamma` (page 364) Upper incomplete gamma function.

`polygamma` (page 361) Polygamma function.

`loggamma` (page 360) Log Gamma function.

`trigamma` (page 364) Trigamma function.

`diofant.functions.special.beta_functions.beta` (page 366) Euler Beta function.

References

[R180] (page 1264), [R181] (page 1264), [R182] (page 1264)

`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 358) Gamma function.

`lowergamma` (page 365) Lower incomplete gamma function.

`uppergamma` (page 364) Upper incomplete gamma function.

`polygamma` (page 361) Polygamma function.

`loggamma` (page 360) Log Gamma function.

`digamma` (page 363) Digamma function.

`diofant.functions.special.beta_functions.beta` (page 366) Euler Beta function.

References

[R183] (page 1264), [R184] (page 1264), [R185] (page 1264)

`class diofant.functions.special.gamma_functions.uppergamma`

The upper incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\Gamma(s, x) := \int_x^\infty t^{s-1} e^{-t} dt = \Gamma(s) - \gamma(s, x).$$

where $\gamma(s, x)$ is the lower incomplete gamma function, **`lowergamma`** (page 365). This can be shown to be the same as

$$\Gamma(s, x) = \Gamma(s) - \frac{x^s}{s} {}_1F_1 \left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x \right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

The upper incomplete gamma function is also essentially equivalent to the generalized exponential integral:

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} dt = x^{n-1} \Gamma(1-n, x).$$

See also:

`gamma` (page 358) Gamma function.

`lowergamma` (page 365) Lower incomplete gamma function.

`polygamma` (page 361) Polygamma function.

`loggamma` (page 360) Log Gamma function.

digamma (page 363) Digamma function.

trigamma (page 364) Trigamma function.

diofant.functions.special.beta_functions.beta (page 366) Euler Beta function.

References

[R186] (page 1264), [R187] (page 1264), [R188] (page 1264), [R189] (page 1264), [R190] (page 1265), [R191] (page 1265)

Examples

```
>>> from diofant import uppergamma, Rational
>>> from diofant.abc import s, x
>>> 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
```

class diofant.functions.special.gamma_functions.Lowergamma

The lower incomplete gamma function.

It can be defined as the meromorphic continuation of

$$\gamma(s, x) := \int_0^x t^{s-1} e^{-t} dt = \Gamma(s) - \Gamma(s, x).$$

This can be shown to be the same as

$$\gamma(s, x) = \frac{x^s}{s} {}_1F_1\left(\begin{matrix} s \\ s+1 \end{matrix} \middle| -x\right),$$

where ${}_1F_1$ is the (confluent) hypergeometric function.

See also:

gamma (page 358) Gamma function.

uppergamma (page 364) Upper incomplete gamma function.

polygamma (page 361) Polygamma function.

loggamma (page 360) Log Gamma function.

digamma (page 363) Digamma function.

trigamma (page 364) Trigamma function.

diofant.functions.special.beta_functions.beta (page 366) Euler Beta function.

References

[R192] (page 1265), [R193] (page 1265), [R194] (page 1265), [R195] (page 1265), [R196] (page 1265)

Examples

```
>>> from diofant import lowergamma, Rational
>>> from diofant.abc import s, x
>>> 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)
```

class diofant.functions.special.beta_functions.beta

The beta integral is called the Eulerian integral of the first kind by Legendre:

$$B(x, y) := \int_0^1 t^{x-1}(1-t)^{y-1} dt.$$

Beta function or Euler's first integral is closely associated with gamma function. The Beta function often used in probability theory and mathematical statistics. It satisfies properties like:

$$\begin{aligned} B(a, 1) &= \frac{1}{a} \\ B(a, b) &= B(b, a) \\ B(a, b) &= \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \end{aligned}$$

Therefore for integral values of a and b:

$$B = \frac{(a-1)!(b-1)!}{(a+b-1)!}$$

See also:

***diofant.functions.special.gamma_functions.gamma* (page 358)** Gamma function.

***diofant.functions.special.gamma_functions.uppergamma* (page 364)** Upper incomplete gamma function.

***diofant.functions.special.gamma_functions.lowergamma* (page 365)** Lower incomplete gamma function.

***diofant.functions.special.gamma_functions.polygamma* (page 361)** Polygamma function.

***diofant.functions.special.gamma_functions.loggamma* (page 360)** Log Gamma function.

***diofant.functions.special.gamma_functions.digamma* (page 363)** Digamma function.

***diofant.functions.special.gamma_functions.trigamma* (page 364)** Trigamma function.

References

[R197] (page 1265), [R198] (page 1265), [R199] (page 1265)

Examples

```
>>> from diofant import I, pi
>>> from diofant.abc import x,y
```

The Beta function obeys the mirror symmetry:

```
>>> from diofant import beta
>>> from diofant import conjugate
>>> conjugate(beta(x,y))
beta(conjugate(x), conjugate(y))
```

Differentiation with respect to both x and y is supported:

```
>>> from diofant import beta
>>> from diofant import diff
>>> diff(beta(x,y), x)
(polygamma(0, x) - polygamma(0, x + y))*beta(x, y)
```

```
>>> from diofant import beta
>>> from diofant import diff
>>> diff(beta(x,y), y)
(polygamma(0, y) - polygamma(0, x + y))*beta(x, y)
```

We can numerically evaluate the gamma function to arbitrary precision on the whole complex plane:

```
>>> from diofant import beta
>>> beta(pi,pi).evalf(40)
0.02671848900111377452242355235388489324562
```

```
>>> beta(1+I,1+I).evalf(20)
-0.2112723729365330143 - 0.7655283165378005676*I
```

Error Functions and Fresnel Integrals

class `diofant.functions.special.error_functions.erf`

The Gauss error function. This function is defined as:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

See also:

`erfc` (page 368) Complementary error function.

`erfi` (page 370) Imaginary error function.

`erf2` (page 371) Two-argument error function.

`erfinv` (page 372) Inverse error function.

`erfcinv` (page 373) Inverse Complementary error function.

`erf2inv` (page 373) Inverse two-argument error function.

References

[R200] (page 1265), [R201] (page 1265), [R202] (page 1265), [R203] (page 1265)

Examples

```
>>> from diofant import I, oo, erf
>>> from diofant.abc import z
```

Several special values are known:

```
>>> erf(0)
0
>>> erf(oo)
1
>>> erf(-oo)
-1
>>> erf(I*oo)
oo*I
>>> erf(-I*oo)
-oo*I
```

In general one can pull out factors of -1 and I from the argument:

```
>>> erf(-z)
-erf(z)
```

The error function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(erf(z))
erf(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> 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
```

class `diofant.functions.special.error_functions.erfc`
 Complementary Error Function. The function is defined as:

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

See also:

erf (page 367) Gaussian error function.
erfi (page 370) Imaginary error function.
erf2 (page 371) Two-argument error function.
erfinv (page 372) Inverse error function.
erfcinv (page 373) Inverse Complementary error function.
erf2inv (page 373) Inverse two-argument error function.

References

[R204] (page 1265), [R205] (page 1265), [R206] (page 1265), [R207] (page 1265)

Examples

```
>>> from diofant import I, oo, erfc
>>> from diofant.abc import z
```

Several special values are known:

```
>>> erfc(0)
1
>>> erfc(oo)
0
>>> erfc(-oo)
2
>>> erfc(I*oo)
-oo*I
>>> erfc(-I*oo)
oo*I
```

The error function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(erfc(z))
erfc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> 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
```

class `diofant.functions.special.error_functions.erfi`
 Imaginary error function. The function `erfi` is defined as:

$$\operatorname{erfi}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{t^2} dt$$

See also:

`erf` (page 367) Gaussian error function.

`erfc` (page 368) Complementary error function.

`erf2` (page 371) Two-argument error function.

`erfinv` (page 372) Inverse error function.

`erfcinv` (page 373) Inverse Complementary error function.

`erf2inv` (page 373) Inverse two-argument error function.

References

[R208] (page 1265), [R209] (page 1265), [R210] (page 1265)

Examples

```
>>> from diofant import I, oo, erfi
>>> from diofant.abc import z
```

Several special values are known:

```
>>> erfi(0)
0
>>> erfi(oo)
oo
>>> erfi(-oo)
-oo
>>> erfi(I*oo)
I
>>> erfi(-I*oo)
-I
```

In general one can pull out factors of `-1` and `I` from the argument:

```
>>> erfi(-z)
-erfi(z)
```

```
>>> from diofant import conjugate
>>> conjugate(erfi(z))
erfi(conjugate(z))
```

Differentiation with respect to `z` is supported:

```
>>> from diofant import diff
>>> 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
```

class `diofant.functions.special.error_functions.erf2`
Two-argument error function. This function is defined as:

$$\operatorname{erf2}(x, y) = \frac{2}{\sqrt{\pi}} \int_x^y e^{-t^2} dt$$

See also:

- `erf` (page 367)** Gaussian error function.
- `erfc` (page 368)** Complementary error function.
- `erfi` (page 370)** Imaginary error function.
- `erfinv` (page 372)** Inverse error function.
- `erfcinv` (page 373)** Inverse Complementary error function.
- `erf2inv` (page 373)** Inverse two-argument error function.

References

[R211] (page 1265)

Examples

```
>>> from diofant import I, oo, erf2
>>> from diofant.abc import x, y
```

Several special values are known:

```
>>> erf2(0, 0)
0
>>> erf2(x, x)
0
>>> erf2(x, oo)
-erf(x) + 1
>>> erf2(x, -oo)
-erf(x) - 1
>>> erf2(oo, y)
erf(y) - 1
>>> erf2(-oo, y)
erf(y) + 1
```

In general one can pull out factors of -1:

```
>>> erf2(-x, -y)
-erf2(x, y)
```

The error function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(erf2(x, y))
erf2(conjugate(x), conjugate(y))
```

Differentiation with respect to x, y is supported:

```
>>> from diofant import diff
>>> diff(erf2(x, y), x)
-2*E**(-x**2)/sqrt(pi)
>>> diff(erf2(x, y), y)
2*E**(-y**2)/sqrt(pi)
```

class `diofant.functions.special.error_functions.erfinv`
Inverse Error Function. The `erfinv` function is defined as:

$$\operatorname{erf}(x) = y \quad \Rightarrow \quad \operatorname{erfinv}(y) = x$$

See also:

`erf` (page 367) Gaussian error function.

`erfc` (page 368) Complementary error function.

`erfi` (page 370) Imaginary error function.

`erf2` (page 371) Two-argument error function.

`erfcinv` (page 373) Inverse Complementary error function.

`erf2inv` (page 373) Inverse two-argument error function.

References

[R212] (page 1265), [R213] (page 1265)

Examples

```
>>> from diofant import I, oo, erfinv
>>> from diofant.abc import x
```

Several special values are known:

```
>>> erfinv(0)
0
>>> erfinv(1)
oo
```

Differentiation with respect to x is supported:


```
>>> from diofant import diff
>>> 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).evalf(30)
0.179143454621291692285822705344
```

class `diofant.functions.special.error_functions.erfcinv`

Inverse Complementary Error Function. The `erfcinv` function is defined as:

$$\operatorname{erfc}(x) = y \quad \Rightarrow \quad \operatorname{erfcinv}(y) = x$$

See also:

`erf` (page 367) Gaussian error function.

`erfc` (page 368) Complementary error function.

`erfi` (page 370) Imaginary error function.

`erf2` (page 371) Two-argument error function.

`erfinv` (page 372) Inverse error function.

`erf2inv` (page 373) Inverse two-argument error function.

References

[R214] (page 1265), [R215] (page 1265)

Examples

```
>>> from diofant import I, oo, erfcinv
>>> from diofant.abc import x
```

Several special values are known:

```
>>> erfcinv(1)
0
>>> erfcinv(0)
oo
```

Differentiation with respect to `x` is supported:

```
>>> from diofant import diff
>>> diff(erfcinv(x), x)
-E**(erfcinv(x)**2)*sqrt(pi)/2
```

class `diofant.functions.special.error_functions.erf2inv`

Two-argument Inverse error function. The `erf2inv` function is defined as:

$$\operatorname{erf2}(x, w) = y \quad \Rightarrow \quad \operatorname{erf2inv}(x, y) = w$$

See also:

erf (page 367) Gaussian error function.
erfc (page 368) Complementary error function.
erfi (page 370) Imaginary error function.
erf2 (page 371) Two-argument error function.
erfinv (page 372) Inverse error function.
erfcinv (page 373) Inverse complementary error function.

References

[R216] (page 1265)

Examples

```
>>> from diofant import I, oo, erf2inv, erfinv, erfcinv
>>> from diofant.abc import x, y
```

Several special values are known:

```
>>> erf2inv(0, 0)
0
>>> erf2inv(1, 0)
1
>>> erf2inv(0, 1)
oo
>>> erf2inv(0, y)
erfinv(y)
>>> erf2inv(oo, y)
erfcinv(-y)
```

Differentiation with respect to x and y is supported:

```
>>> from diofant import diff
>>> 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
```

class `diofant.functions.special.error_functions.FresnelIntegral`
Base class for the Fresnel integrals.

class `diofant.functions.special.error_functions.fresnels`
Fresnel integral S.

This function is defined by

$$S(z) = \int_0^z \sin \frac{\pi}{2} t^2 dt.$$

It is an entire function.

See also:

fresnelc (page 376) Fresnel cosine integral.

References

[R217] (page 1265), [R218] (page 1265), [R219] (page 1265), [R220] (page 1265), [R221] (page 1265)

Examples

```
>>> from diofant import I, oo, fresnels
>>> from diofant.abc import z
```

Several special values are known:

```
>>> fresnels(0)
0
>>> fresnels(oo)
1/2
>>> fresnels(-oo)
-1/2
>>> fresnels(I*oo)
-I/2
>>> fresnels(-I*oo)
I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnels(-z)
-fresnels(z)
>>> fresnels(I*z)
-I*fresnels(z)
```

The Fresnel S integral obeys the mirror symmetry $\overline{S(z)} = S(\bar{z})$:

```
>>> from diofant import conjugate
>>> conjugate(fresnels(z))
fresnels(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(fresnels(z), z)
sin(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from diofant import integrate, pi, sin, gamma, expand_func
>>> integrate(sin(pi*z**2/2), z)
3*fresnels(z)*gamma(3/4)/(4*gamma(7/4))
>>> expand_func(integrate(sin(pi*z**2/2), z))
fresnels(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnels(2).evalf(30)
0.343415678363698242195300815958
```

```
>>> fresnels(-2*I).evalf(30)
0.343415678363698242195300815958*I
```

class diofant.functions.special.error_functions.fresnelc
Fresnel integral C.

This function is defined by

$$C(z) = \int_0^z \cos \frac{\pi}{2} t^2 dt.$$

It is an entire function.

See also:

[fresnels](#) (page 374) Fresnel sine integral.

References

[R222] (page 1265), [R223] (page 1265), [R224] (page 1265), [R225] (page 1265), [R226] (page 1265)

Examples

```
>>> from diofant import I, oo, fresnelc
>>> from diofant.abc import z
```

Several special values are known:

```
>>> fresnelc(0)
0
>>> fresnelc(oo)
1/2
>>> fresnelc(-oo)
-1/2
>>> fresnelc(I*oo)
I/2
>>> fresnelc(-I*oo)
-I/2
```

In general one can pull out factors of -1 and i from the argument:

```
>>> fresnelc(-z)
-fresnelc(z)
>>> fresnelc(I*z)
I*fresnelc(z)
```

The Fresnel C integral obeys the mirror symmetry $\overline{C(z)} = C(\bar{z})$:

```
>>> from diofant import conjugate
>>> conjugate(fresnelc(z))
fresnelc(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(fresnelc(z), z)
cos(pi*z**2/2)
```

Defining the Fresnel functions via an integral

```
>>> from diofant import integrate, pi, cos, gamma, expand_func
>>> integrate(cos(pi*z**2/2), z)
fresnelc(z)*gamma(1/4)/(4*gamma(5/4))
>>> expand_func(integrate(cos(pi*z**2/2), z))
fresnelc(z)
```

We can numerically evaluate the Fresnel integral to arbitrary precision on the whole complex plane:

```
>>> fresnelc(2).evalf(30)
0.488253406075340754500223503357
```

```
>>> fresnelc(-2*I).evalf(30)
-0.488253406075340754500223503357*I
```

Exponential, Logarithmic and Trigonometric Integrals

class diofant.functions.special.error_functions.Ei

The classical exponential integral.

For use in Diofant, this function is defined as

$$\operatorname{Ei}(x) = \sum_{n=1}^{\infty} \frac{x^n}{n n!} + \log(x) + \gamma,$$

where γ is the Euler-Mascheroni constant.

If x is a polar number, this defines an analytic function on the Riemann surface of the logarithm. Otherwise this defines an analytic function in the cut plane $\mathbb{C} \setminus (-\infty, 0]$.

Background

The name *exponential integral* comes from the following statement:

$$\operatorname{Ei}(x) = \int_{-\infty}^x \frac{e^t}{t} dt$$

If the integral is interpreted as a Cauchy principal value, this statement holds for $x > 0$ and $\operatorname{Ei}(x)$ as defined above.

Note that we carefully avoided defining $\operatorname{Ei}(x)$ for negative real x . This is because above integral formula does not hold for any polar lift of such x , indeed all branches of $\operatorname{Ei}(x)$ above the negative reals are imaginary.

However, the following statement holds for all $x \in \mathbb{R}^*$:

$$\int_{-\infty}^x \frac{e^t}{t} dt = \frac{\operatorname{Ei}(|x|e^{i \arg(x)}) + \operatorname{Ei}(|x|e^{-i \arg(x)})}{2},$$

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 .

See also:

expint (page 379) Generalised exponential integral.

E1 (page 380) Special case of the generalised exponential integral.

li (page 381) Logarithmic integral.

Li (page 382) Offset logarithmic integral.

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

diophant.functions.special.gamma_functions.uppergamma (page 364) Upper incomplete gamma function.

References

[R227] (page 1266), [R228] (page 1266), [R229] (page 1266)

Examples

```
>>> from diophant import Ei, polar_lift, exp_polar, I, pi
>>> from diophant.abc import x
```

The exponential integral in Diophant is strictly undefined for negative values of the argument. For convenience, exponential integrals with negative arguments are immediately converted into an expression that agrees with the classical integral definition:

```
>>> Ei(-1)
-I*pi + Ei(exp_polar(I*pi))
```

This yields a real value:

```
>>> Ei(-1).n(chop=True)
-0.219383934395520
```

On the other hand the analytic continuation is not real:

```
>>> Ei(polar_lift(-1)).n(chop=True)
-0.21938393439552 + 3.14159265358979*I
```

The exponential integral has a logarithmic branch point at the origin:

```
>>> Ei(x*exp_polar(2*I*pi))
Ei(x) + 2*I*pi
```

Differentiation is supported:

```
>>> Ei(x).diff(x)
E**x/x
```

The exponential integral is related to many other special functions. For example:

```
>>> from diofant import uppergamma, expint, Shi
>>> Ei(x).rewrite(expint)
-expint(1, x*exp_polar(I*pi)) - I*pi
>>> Ei(x).rewrite(Shi)
Chi(x) + Shi(x)
```

class diofant.functions.special.error_functions.expint

Generalized exponential integral.

This function is defined as

$$E_\nu(z) = z^{\nu-1}\Gamma(1-\nu, z),$$

where $\Gamma(1-\nu, z)$ is the upper incomplete gamma function (`uppergamma`).

Hence for z with positive real part we have

$$E_\nu(z) = \int_1^\infty \frac{e^{-zt}}{t^\nu} dt,$$

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.

See also:

Ei (page 377) Another related function called exponential integral.

E1 (page 380) The classical case, returns `expint(1, z)`.

li (page 381) Logarithmic integral.

Li (page 382) Offset logarithmic integral.

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

diofant.functions.special.gamma_functions.uppergamma (page 364)

References

[R230] (page 1266), [R231] (page 1266), [R232] (page 1266)

Examples

```
>>> from diofant import expint, Rational
>>> from diofant.abc import nu, z
```

Differentiation is supported. Differentiation with respect to z explains further the name: for integral orders, the exponential integral is an iterated integral of the exponential function.

```
>>> expint(nu, z).diff(z)
-expint(nu - 1, z)
```

Differentiation with respect to nu has no classical expression:

```
>>> expint(nu, z).diff(nu)
-z**(nu - 1)*meijerg(((), (1, 1)), ((0, 0, -nu + 1), ()), z)
```

At non-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:

```
>>> from diofant import expand_func
>>> expand_func(expint(5, z))
z**4*expint(1, z)/24 + E**(-z)*(-z**3 + z**2 - 2*z + 6)/24
```

The generalised exponential integral is essentially equivalent to the incomplete gamma function:

```
>>> from diofant import uppergamma
>>> expint(nu, z).rewrite(uppergamma)
z**(nu - 1)*uppergamma(-nu + 1, z)
```

As such it is branched at the origin:

```
>>> from diofant import exp_polar, pi, I
>>> expint(4, z*exp_polar(2*pi*I))
I*pi*z**3/3 + expint(4, z)
>>> expint(nu, z*exp_polar(2*pi*I))
z**(nu - 1)*(E**(2*I*pi*nu) - 1)*gamma(-nu + 1) + expint(nu, z)
```

`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 377) Exponential integral.

expint (page 379) Generalised exponential integral.

li (page 381) Logarithmic integral.

Li (page 382) Offset logarithmic integral.

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

class diofant.functions.special.error_functions.**li**

The classical logarithmic integral.

For the use in Diofant, this function is defined as

$$\operatorname{li}(x) = \int_0^x \frac{1}{\log(t)} dt.$$

See also:

Li (page 382) Offset logarithmic integral.

Ei (page 377) Exponential integral.

expint (page 379) Generalised exponential integral.

E1 (page 380) Special case of the generalised exponential integral.

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

References

[R233] (page 1266), [R234] (page 1266), [R235] (page 1266), [R236] (page 1266)

Examples

```
>>> from diofant import I, oo, li
>>> from diofant.abc import z
```

Several special values are known:

```
>>> li(0)
0
>>> li(1)
-oo
>>> li(oo)
oo
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(li(z), z)
1/log(z)
```

Defining the li function via an integral:

The logarithmic integral can also be defined in terms of Ei :

```
>>> from diofant import Ei
>>> li(z).rewrite(Ei)
Ei(log(z))
>>> diff(li(z).rewrite(Ei), z)
1/log(z)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> li(2).evalf(30)
1.04516378011749278484458888919
```

```
>>> li(2*I).evalf(30)
1.0652795784357498247001125598 + 3.08346052231061726610939702133*I
```

We can even compute Soldner's constant by the help of mpmath:

```
>>> from mpmath import findroot
>>> print(findroot(li, 2))
1.45136923488338
```

Further transformations include rewriting li in terms of the trigonometric integrals Si , Ci , Shi and Chi :

```
>>> from diofant import Si, Ci, Shi, Chi
>>> li(z).rewrite(Si)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Ci)
-log(I*log(z)) - log(1/log(z))/2 + log(log(z))/2 + Ci(I*log(z)) + Shi(log(z))
>>> li(z).rewrite(Shi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
>>> li(z).rewrite(Chi)
-log(1/log(z))/2 + log(log(z))/2 + Chi(log(z)) - Shi(log(z))
```

class diofant.functions.special.error_functions.**Li**

The offset logarithmic integral.

For the use in Diofant, this function is defined as

$$\text{Li}(x) = \text{li}(x) - \text{li}(2)$$

See also:

li (page 381) Logarithmic integral.

Ei (page 377) Exponential integral.

expint (page 379) Generalised exponential integral.

E1 (page 380) Special case of the generalised exponential integral.

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

References

[R237] (page 1266), [R238] (page 1266), [R239] (page 1266)

Examples

```
>>> from diofant import I, oo, Li
>>> from diofant.abc import z
```

The following special value is known:

```
>>> Li(2)
0
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(Li(z), z)
1/log(z)
```

The shifted logarithmic integral can be written in terms of $li(z)$:

```
>>> from diofant import li
>>> Li(z).rewrite(li)
li(z) - li(2)
```

We can numerically evaluate the logarithmic integral to arbitrary precision on the whole complex plane (except the singular points):

```
>>> Li(2).evalf(30)
0
```

```
>>> Li(4).evalf(30)
1.92242131492155809316615998938
```

class `diofant.functions.special.error_functions.Si`
Sine integral.

This function is defined by

$$\text{Si}(z) = \int_0^z \frac{\sin t}{t} dt.$$

It is an entire function.

See also:

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Chi (page 386) Hyperbolic cosine integral.

Ei (page 377) Exponential integral.

expint (page 379) Generalised exponential integral.

E1 (page 380) Special case of the generalised exponential integral.

li (page 381) Logarithmic integral.

Li (page 382) Offset logarithmic integral.

References

[R240] (page 1266)

Examples

```
>>> from diofant import Si
>>> from diofant.abc import z
```

The sine integral is an antiderivative of $\sin(z)/z$:

```
>>> Si(z).diff(z)
sin(z)/z
```

It is unbranched:

```
>>> from diofant import exp_polar, I, pi
>>> Si(z*exp_polar(2*I*pi))
Si(z)
```

Sine integral behaves much like ordinary sine under multiplication by I :

```
>>> Si(I*z)
I*Shi(z)
>>> Si(-z)
-Si(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from diofant import expint
>>> Si(z).rewrite(expint)
-I*(-expint(1, z*exp_polar(-I*pi/2))/2 +
  expint(1, z*exp_polar(I*pi/2))/2) + pi/2
```

class diofant.functions.special.error_functions.Ci

Cosine integral.

This function is defined for positive x by

$$\text{Ci}(x) = \gamma + \log x + \int_0^x \frac{\cos t - 1}{t} dt = - \int_x^\infty \frac{\cos t}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Ci}(z) = - \frac{E_1(e^{i\pi/2}z) + E_1(e^{-i\pi/2}z)}{2}$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm.

The formula also holds as stated for $z \in \mathbb{C}$ with $\Re(z) > 0$. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

See also:

- Si (page 383)** Sine integral.
- Shi (page 385)** Hyperbolic sine integral.
- Chi (page 386)** Hyperbolic cosine integral.
- Ei (page 377)** Exponential integral.
- expint (page 379)** Generalised exponential integral.
- E1 (page 380)** Special case of the generalised exponential integral.
- li (page 381)** Logarithmic integral.
- Li (page 382)** Offset logarithmic integral.

References

[R241] (page 1266)

Examples

```
>>> from diofant import Ci
>>> from diofant.abc import z
```

The cosine integral is a primitive of $\cos(z)/z$:

```
>>> Ci(z).diff(z)
cos(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from diofant import exp_polar, I, pi
>>> Ci(z*exp_polar(2*I*pi))
Ci(z) + 2*I*pi
```

The cosine integral behaves somewhat like ordinary \cos under multiplication by i :

```
>>> from diofant import polar_lift
>>> Ci(polar_lift(I)*z)
Chi(z) + I*pi/2
>>> Ci(polar_lift(-1)*z)
Ci(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from diofant import expint
>>> Ci(z).rewrite(expint)
-expint(1, z*exp_polar(-I*pi/2))/2 - expint(1, z*exp_polar(I*pi/2))/2
```

class `diofant.functions.special.error_functions.Shi`
Sinh integral.

This function is defined by

$$\text{Shi}(z) = \int_0^z \frac{\sinh t}{t} dt.$$

It is an entire function.

See also:

- Si (page 383)** Sine integral.
- Ci (page 384)** Cosine integral.
- Chi (page 386)** Hyperbolic cosine integral.
- Ei (page 377)** Exponential integral.
- expint (page 379)** Generalised exponential integral.
- E1 (page 380)** Special case of the generalised exponential integral.
- li (page 381)** Logarithmic integral.
- Li (page 382)** Offset logarithmic integral.

References

[R242] (page 1266)

Examples

```
>>> from diofant import Shi
>>> from diofant.abc import z
```

The Sinh integral is a primitive of $\sinh(z)/z$:

```
>>> Shi(z).diff(z)
sinh(z)/z
```

It is unbranched:

```
>>> from diofant import exp_polar, I, pi
>>> Shi(z*exp_polar(2*I*pi))
Shi(z)
```

The sinh integral behaves much like ordinary sinh under multiplication by i :

```
>>> Shi(I*z)
I*Si(z)
>>> Shi(-z)
-Shi(z)
```

It can also be expressed in terms of exponential integrals, but beware that the latter is branched:

```
>>> from diofant import expint
>>> Shi(z).rewrite(expint)
expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

class `diofant.functions.special.error_functions.Chi`
Cosh integral.

This function is defined for positive x by

$$\text{Chi}(x) = \gamma + \log x + \int_0^x \frac{\cosh t - 1}{t} dt,$$

where γ is the Euler-Mascheroni constant.

We have

$$\text{Chi}(z) = \text{Ci}\left(e^{i\pi/2}z\right) - i\frac{\pi}{2},$$

which holds for all polar z and thus provides an analytic continuation to the Riemann surface of the logarithm. By lifting to the principal branch we obtain an analytic function on the cut complex plane.

See also:

Si (page 383) Sine integral.

Ci (page 384) Cosine integral.

Shi (page 385) Hyperbolic sine integral.

Ei (page 377) Exponential integral.

expint (page 379) Generalised exponential integral.

E1 (page 380) Special case of the generalised exponential integral.

li (page 381) Logarithmic integral.

Li (page 382) Offset logarithmic integral.

References

[R243] (page 1266)

Examples

```
>>> from diofant import Chi
>>> from diofant.abc import z
```

The cosh integral is a primitive of $\cosh(z)/z$:

```
>>> Chi(z).diff(z)
cosh(z)/z
```

It has a logarithmic branch point at the origin:

```
>>> from diofant import exp_polar, I, pi
>>> Chi(z*exp_polar(2*I*pi))
Chi(z) + 2*I*pi
```

The cosh integral behaves somewhat like ordinary cosh under multiplication by i :

```
>>> from diofant import polar_lift
>>> Chi(polar_lift(I)*z)
Ci(z) + I*pi/2
>>> Chi(polar_lift(-1)*z)
Chi(z) + I*pi
```

It can also be expressed in terms of exponential integrals:

```
>>> from diofant import expint
>>> Chi(z).rewrite(expint)
-expint(1, z)/2 - expint(1, z*exp_polar(I*pi))/2 - I*pi/2
```

Bessel Type Functions

class diofant.functions.special.bessel.BesselBase

Abstract base class for bessel-type functions.

This class is meant to reduce code duplication. All Bessel type functions can 1) be differentiated, and the derivatives expressed in terms of similar functions and 2) be rewritten in terms of other bessel-type functions.

Here “bessel-type functions” are assumed to have one complex parameter.

To use this base class, define class attributes `_a` and `_b` such that $2 * F_n' = -_a * F_{n+1} + _b * F_{n-1}$.

argument

The argument of the bessel-type function.

order

The order of the bessel-type function.

class diofant.functions.special.bessel.besselj

Bessel function of the first kind.

The Bessel J function of order ν is defined to be the function satisfying Bessel’s differential equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2) w = 0,$$

with Laurent expansion

$$J_\nu(z) = z^\nu \left(\frac{1}{\Gamma(\nu + 1) 2^\nu} + O(z^2) \right),$$

if ν is not a negative integer. If $\nu = -n \in \mathbb{Z}_{<0}$ is a negative integer, then the definition is

$$J_{-n}(z) = (-1)^n J_n(z).$$

See also:

[bessely](#) (page 389), [besseli](#) (page 389), [besselk](#) (page 390)

References

[R244] (page 1266), [R245] (page 1266), [R246] (page 1266), [R247] (page 1266)

Examples

Create a Bessel function object:

```
>>> from diofant import besselj, jn
>>> from diofant.abc import z, n
>>> b = besselj(n, z)
```

Differentiate it:

```
>>> b.diff(z)
besselj(n - 1, z)/2 - besselj(n + 1, z)/2
```


Rewrite in terms of spherical Bessel functions:

```
>>> b.rewrite(jn)
sqrt(2)*sqrt(z)*jn(n - 1/2, z)/sqrt(pi)
```

Access the parameter and argument:

```
>>> b.order
n
>>> b.argument
z
```

class diofant.functions.special.bessel.bessely

Bessel function of the second kind.

The Bessel Y function of order ν is defined as

$$Y_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{J_\mu(z) \cos(\pi\mu) - J_{-\mu}(z)}{\sin(\pi\mu)},$$

where $J_\mu(z)$ is the Bessel function of the first kind.

It is a solution to Bessel's equation, and linearly independent from J_ν .

See also:

[besselj](#) (page 388), [besseli](#) (page 389), [besselk](#) (page 390)

References

[R248] (page 1266)

Examples

```
>>> from diofant import bessely, yn
>>> from diofant.abc import z, n
>>> b = bessely(n, z)
>>> b.diff(z)
bessely(n - 1, z)/2 - bessely(n + 1, z)/2
>>> b.rewrite(yn)
sqrt(2)*sqrt(z)*yn(n - 1/2, z)/sqrt(pi)
```

class diofant.functions.special.bessel.besseli

Modified Bessel function of the first kind.

The Bessel I function is a solution to the modified Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} + (z^2 + \nu^2) w = 0.$$

It can be defined as

$$I_\nu(z) = i^{-\nu} J_\nu(iz),$$

where $J_\nu(z)$ is the Bessel function of the first kind.

See also:

[besselj](#) (page 388), [bessely](#) (page 389), [besselk](#) (page 390)

References

[R249] (page 1266)

Examples

```
>>> from diofant import besseli
>>> from diofant.abc import z, n
>>> besseli(n, z).diff(z)
besseli(n - 1, z)/2 + besseli(n + 1, z)/2
```

class `diofant.functions.special.bessel.besselk`

Modified Bessel function of the second kind.

The Bessel K function of order ν is defined as

$$K_\nu(z) = \lim_{\mu \rightarrow \nu} \frac{\pi}{2} \frac{I_{-\mu}(z) - I_\mu(z)}{\sin(\pi\mu)},$$

where $I_\mu(z)$ is the modified Bessel function of the first kind.

It is a solution of the modified Bessel equation, and linearly independent from Y_ν .

See also:

[besselj](#) (page 388), [besseli](#) (page 389), [bessely](#) (page 389)

References

[R250] (page 1266)

Examples

```
>>> from diofant import besselk
>>> from diofant.abc import z, n
>>> besselk(n, z).diff(z)
-besselk(n - 1, z)/2 - besselk(n + 1, z)/2
```

class `diofant.functions.special.bessel.hankel1`

Hankel function of the first kind.

This function is defined as

$$H_\nu^{(1)} = J_\nu(z) + iY_\nu(z),$$

where $J_\nu(z)$ is the Bessel function of the first kind, and $Y_\nu(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation.

See also:

[hankel2](#) (page 391), [besselj](#) (page 388), [bessely](#) (page 389)

References

[R251] (page 1266)

Examples

```
>>> from diofant import hankel1
>>> from diofant.abc import z, n
>>> hankel1(n, z).diff(z)
hankel1(n - 1, z)/2 - hankel1(n + 1, z)/2
```

class diofant.functions.special.bessel.hankel2

Hankel function of the second kind.

This function is defined as

$$H_\nu^{(2)} = J_\nu(z) - iY_\nu(z),$$

where $J_\nu(z)$ is the Bessel function of the first kind, and $Y_\nu(z)$ is the Bessel function of the second kind.

It is a solution to Bessel's equation, and linearly independent from $H_\nu^{(1)}$.

See also:

[hankel1](#) (page 390), [besselj](#) (page 388), [bessely](#) (page 389)

References

[R252] (page 1266)

Examples

```
>>> from diofant import hankel2
>>> from diofant.abc import z, n
>>> hankel2(n, z).diff(z)
hankel2(n - 1, z)/2 - hankel2(n + 1, z)/2
```

class diofant.functions.special.bessel.jn

Spherical Bessel function of the first kind.

This function is a solution to the spherical Bessel equation

$$z^2 \frac{d^2 w}{dz^2} + 2z \frac{dw}{dz} + (z^2 - \nu(\nu + 1))w = 0.$$

It can be defined as

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} J_{\nu+\frac{1}{2}}(z),$$

where $J_\nu(z)$ is the Bessel function of the first kind.

See also:

[besselj](#) (page 388), [bessely](#) (page 389), [besselk](#) (page 390), [yn](#) (page 392)

Examples

```
>>> from diofant import Symbol, jn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print(jn(0, z).expand(func=True))
sin(z)/z
>>> jn(1, z).expand(func=True) == sin(z)/z**2 - cos(z)/z
True
>>> expand_func(jn(3, z))
(-6/z**2 + 15/z**4)*sin(z) + (1/z - 15/z**3)*cos(z)
```

The spherical Bessel functions of integral order are calculated using the formula:

$$j_n(z) = f_n(z) \sin z + (-1)^{n+1} f_{-n-1}(z) \cos z,$$

where the coefficients $f_n(z)$ are available as `diofant.polys.orthopolys.spherical_bessel_fn()` (page 736).

class diofant.functions.special.bessel.yn

Spherical Bessel function of the second kind.

This function is another solution to the spherical Bessel equation, and linearly independent from j_n . It can be defined as

$$j_\nu(z) = \sqrt{\frac{\pi}{2z}} Y_{\nu+\frac{1}{2}}(z),$$

where $Y_\nu(z)$ is the Bessel function of the second kind.

See also:

[besselj](#) (page 388), [bessely](#) (page 389), [besselk](#) (page 390), [jn](#) (page 391)

Examples

```
>>> from diofant import Symbol, yn, sin, cos, expand_func
>>> z = Symbol("z")
>>> print(expand_func(yn(0, z)))
-cos(z)/z
>>> expand_func(yn(1, z)) == -cos(z)/z**2-sin(z)/z
True
```

For integral orders n , y_n is calculated using the formula:

$$y_n(z) = (-1)^{n+1} j_{-n-1}(z)$$

diofant.functions.special.bessel.jn_zeros(*n*, *k*, *method*='diofant', *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.

- `method = "diofant"`: uses mpmath's function `besseljzero`
- `method = "scipy"`: uses the SciPy's `sph_jn` and `newton` to find all roots, which is faster than computing the zeros using a general numerical solver, but it requires SciPy and only works with low precision floating point numbers. [The function used with `method="diofant"` is a recent addition to mpmath, before that a general solver was used.]

See also:

jn (page 391), *yn* (page 392), *besselj* (page 388), *besselk* (page 390), *bessely* (page 389)

Examples

```
>>> from diofant import jn_zeros
>>> jn_zeros(2, 4, dps=5)
[5.7635, 9.095, 12.323, 15.515]
```

Airy Functions

class diofant.functions.special.bessel.AiryBase

Abstract base class for Airy functions.

This class is meant to reduce code duplication.

class diofant.functions.special.bessel.airyai

The Airy function Ai of the first kind.

The Airy function Ai(*z*) is defined to be the function satisfying Airy's differential equation

$$\frac{d^2w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real *z*

$$\text{Ai}(z) := \frac{1}{\pi} \int_0^\infty \cos\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

airybi (page 394) Airy function of the second kind.

airyaiprime (page 395) Derivative of the Airy function of the first kind.

airybiprime (page 397) Derivative of the Airy function of the second kind.

References

[R253] (page 1266), [R254] (page 1266), [R255] (page 1266), [R256] (page 1266)

Examples

Create an Airy function object:

```
>>> from diofant import airyai
>>> from diofant.abc import z
```

```
>>> airyai(z)
airyai(z)
```

Several special values are known:

```
>>> airyai(0)
3**(1/3)/(3*gamma(2/3))
>>> from diofant import oo
>>> airyai(oo)
0
>>> airyai(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(airyai(z))
airyai(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(airyai(z), z)
airyaiprime(z)
>>> diff(airyai(z), z, 2)
z*airyai(z)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(airyai(z), z, 0, 3)
3**(5/6)*gamma(1/3)/(6*pi) - 3**(1/6)*z*gamma(2/3)/(2*pi) + O(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airyai(-2).evalf(50)
0.22740742820168557599192443603787379946077222541710
```

Rewrite $Ai(z)$ in terms of hypergeometric functions:

```
>>> from diofant import hyper
>>> airyai(z).rewrite(hyper)
-3**(2/3)*z*hyper(( ), (4/3, ), z**3/9)/(3*gamma(1/3)) + 3**(1/3)*hyper(( ), (2/3, ),
↪ z**3/9)/(3*gamma(2/3))
```

`class diofant.functions.special.bessel.airybi`

The Airy function Bi of the second kind.

The Airy function $Bi(z)$ is defined to be the function satisfying Airy's differential equation

$$\frac{d^2w(z)}{dz^2} - zw(z) = 0.$$

Equivalently, for real z

$$Bi(z) := \frac{1}{\pi} \int_0^\infty \exp\left(-\frac{t^3}{3} + zt\right) + \sin\left(\frac{t^3}{3} + zt\right) dt.$$

See also:

[**`airyai`** \(page 393\)](#) Airy function of the first kind.

[**`airyaiprime`** \(page 395\)](#) Derivative of the Airy function of the first kind.

[**`airybiprime`** \(page 397\)](#) Derivative of the Airy function of the second kind.

References

[R257] (page 1266), [R258] (page 1266), [R259] (page 1266), [R260] (page 1266)

Examples

Create an Airy function object:

```
>>> from diofant import airybi
>>> from diofant.abc import z
```

```
>>> airybi(z)
airybi(z)
```

Several special values are known:

```
>>> airybi(0)
3**(5/6)/(3*gamma(2/3))
>>> from diofant import oo
>>> airybi(oo)
oo
>>> airybi(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(airybi(z))
airybi(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(airybi(z), z)
airybiprime(z)
>>> diff(airybi(z), z, 2)
z*airybi(z)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(airybi(z), z, 0, 3)
3**(1/3)*gamma(1/3)/(2*pi) + 3**(2/3)*z*gamma(2/3)/(2*pi) + 0(z**3)
```

We can numerically evaluate the Airy function to arbitrary precision on the whole complex plane:

```
>>> airybi(-2).evalf(50)
-0.41230258795639848808323405461146104203453483447240
```

Rewrite $\text{Bi}(z)$ in terms of hypergeometric functions:

```
>>> from diofant import hyper
>>> airybi(z).rewrite(hyper)
3**(1/6)*z*hyper((), (4/3,), z**3/9)/gamma(1/3) + 3**(5/6)*hyper((), (2/3,), z**3/
↳ 9)/(3*gamma(2/3))
```

class diofant.functions.special.bessel.airyaiprime

The derivative Ai' of the Airy function of the first kind.

The Airy function $Ai'(z)$ is defined to be the function

$$Ai'(z) := \frac{d Ai(z)}{dz}.$$

See also:

airyai (page 393) Airy function of the first kind.

airybi (page 394) Airy function of the second kind.

airybiprime (page 397) Derivative of the Airy function of the second kind.

References

[R261] (page 1266), [R262] (page 1266), [R263] (page 1266), [R264] (page 1267)

Examples

Create an Airy function object:

```
>>> from diofant import airyaiprime
>>> from diofant.abc import z
```

```
>>> airyaiprime(z)
airyaiprime(z)
```

Several special values are known:

```
>>> airyaiprime(0)
-3**(2/3)/(3*gamma(1/3))
>>> from diofant import oo
>>> airyaiprime(oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(airyaiprime(z))
airyaiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(airyaiprime(z), z)
z*airyai(z)
>>> diff(airyaiprime(z), z, 2)
z*airyaiprime(z) + airyai(z)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(airyaiprime(z), z, 0, 3)
-3**(2/3)/(3*gamma(1/3)) + 3**(1/3)*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:

```
>>> from diofant import hyper
>>> airyaiprime(z).rewrite(hyper)
3**(1/3)*z**2*hyper((), (5/3,), z**3/9)/(6*gamma(2/3)) - 3**(2/3)*hyper((), (1/3,
↪), z**3/9)/(3*gamma(1/3))
```

class `diofant.functions.special.bessel.airybiprime`

The derivative Bi' of the Airy function of the first kind.

The Airy function $Bi'(z)$ is defined to be the function

$$Bi'(z) := \frac{d Bi(z)}{dz}.$$

See also:

airyai (page 393) Airy function of the first kind.

airybi (page 394) Airy function of the second kind.

airyaiprime (page 395) Derivative of the Airy function of the first kind.

References

[R265] (page 1267), [R266] (page 1267), [R267] (page 1267), [R268] (page 1267)

Examples

Create an Airy function object:

```
>>> from diofant import airybiprime
>>> from diofant.abc import z
```

```
>>> airybiprime(z)
airybiprime(z)
```

Several special values are known:

```
>>> airybiprime(0)
3**(1/6)/gamma(1/3)
>>> from diofant import oo
>>> airybiprime(oo)
oo
>>> airybiprime(-oo)
0
```

The Airy function obeys the mirror symmetry:

```
>>> from diofant import conjugate
>>> conjugate(airybiprime(z))
airybiprime(conjugate(z))
```

Differentiation with respect to z is supported:

```
>>> from diofant import diff
>>> diff(airybiprime(z), z)
z*airybi(z)
>>> diff(airybiprime(z), z, 2)
z*airybiprime(z) + airybi(z)
```

Series expansion is also supported:

```
>>> from diofant import series
>>> series(airybiprime(z), z, 0, 3)
3**(1/6)/gamma(1/3) + 3**(5/6)*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 $Bi'(z)$ in terms of hypergeometric functions:

```
>>> from diofant import hyper
>>> airybiprime(z).rewrite(hyper)
3**(5/6)*z**2*hyper((), (5/3,), z**3/9)/(6*gamma(2/3)) + 3**(1/6)*hyper((), (1/3,
↪), z**3/9)/gamma(1/3)
```

B-Splines

`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 [R269] (page 1267). They are defined on a set of knots, which is a sequence of integers or floats.

The 0th degree splines have a value of one on a single interval:

```
>>> from diofant import bspline_basis
>>> from diofant.abc import x
>>> d = 0
>>> knots = range(5)
>>> bspline_basis(d, knots, 0, x)
Piecewise((1, And(x <= 1, x >= 0)), (0, true))
```

For a given $(d, knots)$ there are $\text{len}(knots) - d - 1$ 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, And(x < 1, x >= 0)),
          (-x**3/2 + 2*x**2 - 2*x + 2/3, And(x < 2, x >= 1)),
          (x**3/2 - 4*x**2 + 10*x - 22/3, And(x < 3, x >= 2)),
```

```
(-x**3/6 + 2*x**2 - 8*x + 32/3, And(x <= 4, x >= 3)),
(0, true))
```

By repeating knot points, you can introduce discontinuities in the B-splines and their derivatives:

```
>>> d = 1
>>> knots = [0,0,2,3,4]
>>> bspline_basis(d, knots, 0, x)
Piecewise((-x/2 + 1, And(x <= 2, x >= 0)), (0, true))
```

It is quite time consuming to construct and evaluate B-splines. If you need to evaluate a B-splines many times, it is best to `lambdify` them first:

```
>>> from diofant import lambdify
>>> 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 399)

References

[R269] (page 1267)

`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 .

See also:

[*diofant.functions.special.bsplines.bspline_basis*](#) (page 398)

Examples

```
>>> from diofant import bspline_basis_set
>>> from diofant.abc import x
>>> d = 2
>>> knots = range(5)
>>> splines = bspline_basis_set(d, knots, x)
>>> splines
[Piecewise((x**2/2, And(x < 1, x >= 0)),
          (-x**2 + 3*x - 3/2, And(x < 2, x >= 1)),
          (x**2/2 - 3*x + 9/2, And(x <= 3, x >= 2)),
          (0, true)),
 Piecewise((x**2/2 - x + 1/2, And(x < 2, x >= 1)),
          (-x**2 + 5*x - 11/2, And(x < 3, x >= 2)),
          (x**2/2 - 4*x + 8, And(x <= 4, x >= 3)),
          (0, true))]
```

Riemann Zeta and Related Functions

class `diofant.functions.special.zeta_functions.zeta`
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 403) 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 401), [lerchphi](#) (page 403), [polylog](#) (page 402)

References

[R270] (page 1267), [R271] (page 1267)

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 import zeta
>>> 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:

```
>>> from diofant import dirichlet_eta
>>> zeta(s).rewrite(dirichlet_eta)
dirichlet_eta(s)/(-2**(-s + 1) + 1)
```

The Riemann zeta function at positive even integer and negative odd integer values is related to the Bernoulli numbers:

```
>>> zeta(2)
pi**2/6
>>> zeta(4)
pi**4/90
>>> zeta(-1)
-1/12
```

The specific formulae are:

$$\zeta(2n) = (-1)^{n+1} \frac{B_{2n} (2\pi)^{2n}}{2(2n)!}$$

$$\zeta(-n) = -\frac{B_{n+1}}{n+1}$$

At negative even integers the Riemann zeta function is zero:

```
>>> zeta(-4)
0
```

No closed-form expressions are known at positive odd integers, but numerical evaluation is possible:

```
>>> zeta(3).n()
1.20205690315959
```

The derivative of $\zeta(s, a)$ with respect to a is easily computed:

```
>>> from diofant.abc import a
>>> zeta(s, a).diff(a)
-s*zeta(s + 1, a)
```

However the derivative with respect to s has no useful closed form expression:

```
>>> zeta(s, a).diff(s)
Derivative(zeta(s, a), s)
```

The Hurwitz zeta function can be expressed in terms of the Lerch transcendent, *diofant.functions.special.zeta_functions.lerchphi* (page 403):

```
>>> from diofant import lerchphi
>>> zeta(s, a).rewrite(lerchphi)
lerchphi(1, s, a)
```

class diofant.functions.special.zeta_functions.dirichlet_eta

Dirichlet eta function.

For $\text{Re}(s) > 0$, 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 400)

References

[R272] (page 1267), [R273] (page 1267)

Examples

The Dirichlet eta function is closely related to the Riemann zeta function:

```
>>> from diofant import dirichlet_eta, zeta
>>> from diofant.abc import s
>>> dirichlet_eta(s).rewrite(zeta)
(-2**(-s + 1) + 1)*zeta(s)
```

class `diofant.functions.special.zeta_functions.polylog`

Polylogarithm function.

For $|z| < 1$ and $s \in \mathbb{C}$, the polylogarithm is defined by

$$\operatorname{Li}_s(z) = \sum_{n=1}^{\infty} \frac{z^n}{n^s},$$

where the standard branch of the argument is used for n . It admits an analytic continuation which is branched at $z = 1$ (notably not on the sheet of initial definition), $z = 0$ and $z = \infty$.

The name polylogarithm comes from the fact that for $s = 1$, the polylogarithm is related to the ordinary logarithm (see examples), and that

$$\operatorname{Li}_{s+1}(z) = \int_0^z \frac{\operatorname{Li}_s(t)}{t} dt.$$

The polylogarithm is a special case of the Lerch transcendent:

$$\operatorname{Li}_s(z) = z\Phi(z, s, 1)$$

See also:

[zeta](#) (page 400), [lerchphi](#) (page 403)

References

[R274] (page 1267), [R275] (page 1267)

Examples

For $z \in \{0, 1, -1\}$, the polylogarithm is automatically expressed using other functions:

```
>>> from diofant import polylog
>>> 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()`:

```
>>> from diofant import expand_func
>>> from diofant.abc import z
>>> expand_func(polylog(1, z))
-log(z*exp_polar(-I*pi) + 1)
>>> expand_func(polylog(0, z))
z/(-z + 1)
```

The derivative with respect to z can be computed in closed form:

```
>>> polylog(s, z).diff(z)
polylog(s - 1, z)/z
```

The polylogarithm can be expressed in terms of the lerch transcendent:

```
>>> from diofant import lerchphi
>>> polylog(s, z).rewrite(lerchphi)
z*lerchphi(z, s, 1)
```

class `diofant.functions.special.zeta_functions.lerchphi`

Lerch transcendent (Lerch phi function).

For $\operatorname{Re}(a) > 0$, $|z| < 1$ and $s \in \mathbb{C}$, the Lerch transcendent is defined as

$$\Phi(z, s, a) = \sum_{n=0}^{\infty} \frac{z^n}{(n+a)^s},$$

where the standard branch of the argument is used for $n+a$, and by analytic continuation for other values of the parameters.

A commonly used related function is the Lerch zeta function, defined by

$$L(q, s, a) = \Phi(e^{2\pi i q}, s, a).$$

Analytic Continuation and Branching Behavior

It can be shown that

$$\Phi(z, s, a) = z\Phi(z, s, a + 1) + a^{-s}.$$

This provides the analytic continuation to $\operatorname{Re}(a) \leq 0$.

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 402), [zeta](#) (page 400)

References

[R276] (page 1267), [R277] (page 1267), [R278] (page 1267)

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 import lerchphi, expand_func
>>> from diofant.abc import z, s, a
>>> expand_func(lerchphi(1, s, a))
zeta(s, a)
```

More generally, if z is a root of unity, the Lerch transcendent reduces to a sum of Hurwitz zeta functions:

```
>>> expand_func(lerchphi(-1, s, a))
2**(-s)*zeta(s, a/2) - 2**(-s)*zeta(s, a/2 + 1/2)
```

If $a = 1$, the Lerch transcendent reduces to the polylogarithm:

```
>>> expand_func(lerchphi(z, s, 1))
polylog(s, z)/z
```

More generally, if a is rational, the Lerch transcendent reduces to a sum of polylogarithms:

```
>>> from diofant import Rational
>>> 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`

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

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n z^n}{(b_1)_n \dots (b_q)_n n!},$$

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.

See also:

[diofant.simplify.hyperexpand](#) (page 943), [diofant.functions.special.gamma_functions.gamma](#) (page 358), [diofant.functions.special.hyper.meijerg](#) (page 406)

References

[R279] (page 1267), [R280] (page 1267)

Examples

The parameters a_p and b_q can be passed as arbitrary iterables, for example:

```
>>> from diofant.functions import hyper
>>> from diofant.abc import x, n, a
>>> hyper((1, 2, 3), [3, 4], x)
hyper((1, 2, 3), (3, 4), x)
```

There is also pretty printing (it looks better using unicode):

```
>>> from diofant import pprint
>>> pprint(hyper((1, 2, 3), [3, 4], x), use_unicode=False)

|_ /1, 2, 3 | \
| | | | x|
3 2 \ 3, 4 | /
```

The parameters must always be iterables, even if they are vectors of length one or zero:

```
>>> hyper((1, ), [1], x)
hyper((1,), (), x)
```

But of course they may be variables (but if they depend on x then you should not expect much implemented functionality):

```
>>> hyper((n, a), (n**2,), x)
hyper((n, a), (n**2,), x)
```

The hypergeometric function generalizes many named special functions. The function `hyperexpand()` tries to express a hypergeometric function using named special functions. For example:

```
>>> from diofant import hyperexpand
>>> hyperexpand(hyper([], [], x))
E**x
```

You can also use `expand_func`:

```
>>> from diofant import expand_func
>>> expand_func(x*hyper([1, 1], [2], -x))
log(x + 1)
```

More examples:

```
>>> from diofant import Rational
>>> 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:

```
>>> from diofant.abc import a
>>> hyperexpand(hyper([-a], [], x))
(-x + 1)**a
```

ap

Numerator parameters of the hypergeometric function.

argument

Argument of the hypergeometric function.

bq

Denominator parameters of the hypergeometric function.

convergence_statement

Return a condition on z under which the series converges.

eta

A quantity related to the convergence of the series.

radius_of_convergence

Compute the radius of convergence of the defining series.

Note that even if this is not ∞ , the function may still be evaluated outside of the radius of convergence by analytic continuation. But if this is zero, then the function is not actually defined anywhere else.

```
>>> from diofant.functions import hyper
>>> from diofant.abc import z
>>> hyper((1, 2), [3], z).radius_of_convergence
1
>>> hyper((1, 2, 3), [4], z).radius_of_convergence
0
>>> hyper((1, 2), (3, 4), z).radius_of_convergence
oo
```

class `diofant.functions.special.hyper.meijerg`

The Meijer G-function is defined by a Mellin-Barnes type integral that resembles an inverse Mellin transform. It generalizes the hypergeometric functions.

The Meijer G-function depends on four sets of parameters. There are “*numerator parameters*” a_1, \dots, a_n and a_{n+1}, \dots, a_p , and there are “*denominator parameters*” b_1, \dots, b_m and b_{m+1}, \dots, b_q . Confusingly, it is traditionally denoted as follows (note the position of m, n, p, q , and how they relate to the lengths of the four parameter vectors):

$$G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_n & a_{n+1}, \dots, a_p \\ b_1, \dots, b_m & b_{m+1}, \dots, b_q \end{matrix} \middle| z \right).$$

However, in `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.

See also:

[`diofant.functions.special.hyper.hyperfant.simplify.hyperexpand`](#) (page 943) (page 404), [`diofant.functions.special.hyper.hyperfant.simplify.hyperexpand`](#) (page 943)

References

[R281] (page 1267), [R282] (page 1267)

Examples

You can pass the parameters either as four separate vectors:

```
>>> from diofant.functions import meijerg
>>> from diofant.abc import x, a
>>> from diofant.core.containers import Tuple
>>> from diofant import pprint
>>> pprint(meijerg((1, 2), (a, 4), (5,), [], x), use_unicode=False)
  _ 1, 2 /1, 2 a, 4 | \
 /_  | | | x|
 \_ |4, 1 \ 5 | /
```

or as two nested vectors:

```
>>> pprint(meijerg([(1, 2), (3, 4)], ([5], Tuple()), x), use_unicode=False)
  _1, 2 /1, 2 3, 4 | \
 /_ | | | x|
 \_|4, 1 \ 5 | /
```

As with the hypergeometric function, the parameters may be passed as arbitrary iterables. Vectors of length zero and one also have to be passed as iterables. The parameters need not be constants, but if they depend on the argument then not much implemented functionality should be expected.

All the subvectors of parameters are available:

```
>>> from diofant import pprint
>>> g = meijerg([1], [2], [3], [4], x)
>>> pprint(g, use_unicode=False)
  _1, 1 /1 2 | \
 /_ | | | x|
 \_|2, 2 \3 4 | /
>>> g.an
(1,)
>>> g.ap
(1, 2)
>>> g.aother
(2,)
>>> g.bm
(3,)
>>> g.bq
(3, 4)
>>> g.bothor
(4,)
```

The Meijer G-function generalizes the hypergeometric functions. In some cases it can be expressed in terms of hypergeometric functions, using Slater's theorem. For example:

```
>>> from diofant import hyperexpand
>>> from diofant.abc import a, b, c
>>> hyperexpand(meijerg([a], [], [c], [b], x), allow_hyper=True)
x**c*gamma(-a + c + 1)*hyper((-a + c + 1,),
                             (-b + c + 1,), -x)/gamma(-b + c + 1)
```

Thus the Meijer G-function also subsumes many named functions as special cases. You can use `expand_func` or `hyperexpand` to (try to) rewrite a Meijer G-function in terms of named special functions. For example:

```
>>> from diofant import expand_func, Rational
>>> expand_func(meijerg([], []), [[0], []], -x)
E**x
>>> hyperexpand(meijerg([], []), [[Rational(1, 2)], [0]], (x/2)**2)
sin(x)/sqrt(pi)
```

an First set of numerator parameters.

aother Second set of numerator parameters.

ap Combined numerator parameters.

argument

Argument of the Meijer G-function.

bm

First set of denominator parameters.

bother

Second set of denominator parameters.

bq

Combined denominator parameters.

delta

A quantity related to the convergence region of the integral, c.f. references.

get_period()

Return a number P such that $G(x*\exp(I*P)) == G(x)$.

```
>>> from diofant.functions.special.hyper import meijerg
>>> from diofant.abc import z
>>> from diofant import pi, S
```

```
>>> meijerg([1], [], [], [], z).get_period()
2*pi
>>> meijerg([pi], [], [], [], z).get_period()
oo
>>> meijerg([1, 2], [], [], [], z).get_period()
oo
>>> meijerg([1,1], [2], [1, Rational(1, 2), Rational(1, 3)], [1], z).get_
period()
12*pi
```

integrand(s)

Get the defining integrand $D(s)$.

nu

A quantity related to the convergence region of the integral, c.f. references.

Elliptic integrals**class diofant.functions.special.elliptic_integrals.elliptic_k**

The complete elliptic integral of the first kind, defined by

$$K(z) = F\left(\frac{\pi}{2} \middle| z\right)$$

where $F(z|m)$ is the Legendre incomplete elliptic integral of the first kind.

The function $K(z)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

See also:

[*elliptic_f*](#) (page 410)

References

[R283] (page 1267), [R284] (page 1267)

Examples

```
>>> from diofant import elliptic_k, I, pi
>>> from diofant.abc import z
>>> elliptic_k(0)
pi/2
>>> elliptic_k(1.0 + I)
1.50923695405127 + 0.625146415202697*I
>>> elliptic_k(z).series(z, n=3)
pi/2 + pi*z/8 + 9*pi*z**2/128 + O(z**3)
```

class diofant.functions.special.elliptic_integrals.elliptic_f

The Legendre incomplete elliptic integral of the first kind, defined by

$$F(z|m) = \int_0^z \frac{dt}{\sqrt{1 - m \sin^2 t}}$$

This function reduces to a complete elliptic integral of the first kind, $K(m)$, when $z = \pi/2$.

See also:

[elliptic_k](#) (page 409)

References

[R285] (page 1267), [R286] (page 1267)

Examples

```
>>> from diofant import elliptic_f, I, 0
>>> from diofant.abc import z, m
>>> elliptic_f(z, m).series(z)
z + z**5*(3*m**2/40 - m/30) + m*z**3/6 + O(z**6)
>>> elliptic_f(3.0 + I/2, 1.0 + I)
2.909449841483 + 1.74720545502474*I
```

class diofant.functions.special.elliptic_integrals.elliptic_e

Called with two arguments z and m , evaluates the incomplete elliptic integral of the second kind, defined by

$$E(z|m) = \int_0^z \sqrt{1 - m \sin^2 t} dt$$

Called with a single argument z , evaluates the Legendre complete elliptic integral of the second kind

$$E(z) = E\left(\frac{\pi}{2} | z\right)$$

The function $E(z)$ is a single-valued function on the complex plane with branch cut along the interval $(1, \infty)$.

References

[R287] (page 1267), [R288] (page 1267), [R289] (page 1267)

Examples

```
>>> from diofant import elliptic_e, I, pi, 0
>>> from diofant.abc import z, m
>>> elliptic_e(z, m).series(z)
z + z**5*(-m**2/40 + m/30) - m*z**3/6 + 0(z**6)
>>> elliptic_e(z).series(z, n=4)
pi/2 - pi*z/8 - 3*pi*z**2/128 - 5*pi*z**3/512 + 0(z**4)
>>> elliptic_e(1 + I, 2 - I/2).n()
1.55203744279187 + 0.290764986058437*I
>>> elliptic_e(0)
pi/2
>>> elliptic_e(2.0 - I)
0.991052601328069 + 0.81879421395609*I
```

class diofant.functions.special.elliptic_integrals.elliptic_pi

Called with three arguments n , z and m , evaluates the Legendre incomplete elliptic integral of the third kind, defined by

$$\Pi(n; z|m) = \int_0^z \frac{dt}{(1 - n \sin^2 t) \sqrt{1 - m \sin^2 t}}$$

Called with two arguments n and m , evaluates the complete elliptic integral of the third kind:

$$\Pi(n|m) = \Pi\left(n; \frac{\pi}{2} | m\right)$$

References

[R290] (page 1267), [R291] (page 1267), [R292] (page 1267)

Examples

```
>>> from diofant import elliptic_pi, I, pi, 0, S
>>> from diofant.abc import z, n, m
>>> elliptic_pi(n, z, m).series(z, n=4)
z + z**3*(m/6 + n/3) + 0(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
```

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`

Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$

`jacobi(n, alpha, beta, x)` gives the n th Jacobi polynomial in x , $P_n^{(\alpha,\beta)}(x)$.

The Jacobi polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1-x)^\alpha(1+x)^\beta$.

See also:

[gegenbauer](#) (page 414), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R293] (page 1267), [R294] (page 1267), [R295] (page 1267)

Examples

```
>>> from diofant import jacobi, conjugate, diff, Rational
>>> from diofant.abc import n,a,b,x
```

```
>>> jacobi(0, a, b, x)
1
>>> jacobi(1, a, b, x)
a/2 - b/2 + x*(a/2 + b/2 + 1)
```

```
>>> jacobi(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/2 + b/2 + n/2 + 1/2)*jacobi(n - 1, a + 1, b + 1, x)
```

`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:

$$\int_{-1}^1 P_m^{(\alpha,\beta)}(x)P_n^{(\alpha,\beta)}(x)(1-x)^\alpha(1+x)^\beta dx = \delta_{m,n}$$

See also:

[gegenbauer](#) (page 414), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [as-soc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R296] (page 1267), [R297] (page 1267), [R298] (page 1267)

Examples

```
>>> from diofant import jacobi_normalized
>>> from diofant.abc import n,a,b,x
```

```
>>> jacobi_normalized(n, a, b, x)
jacobi(n, a, b, x)/sqrt(2**(a + b + 1)*gamma(a + n + 1)*gamma(b + n + 1)/((a + b
↳+ 2*n + 1)*factorial(n)*gamma(a + b + n + 1)))
```

Gegenbauer Polynomials

class `diofant.functions.special.polynomials.gegenbauer`

Gegenbauer polynomial $C_n^{(\alpha)}(x)$

`gegenbauer(n, alpha, x)` gives the nth Gegenbauer polynomial in x , $C_n^{(\alpha)}(x)$.

The Gegenbauer polynomials are orthogonal on $[-1,1]$ with respect to the weight $(1 - x^2)^{\alpha - \frac{1}{2}}$.

See also:

[jacobi](#) (page 412), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R299] (page 1267), [R300] (page 1267), [R301] (page 1268)

Examples

```
>>> from diofant import gegenbauer, conjugate, diff
>>> from diofant.abc import n,a,x
>>> gegenbauer(0, a, x)
1
>>> gegenbauer(1, a, x)
2*a*x
>>> gegenbauer(2, a, x)
-a + x**2*(2*a**2 + 2*a)
>>> gegenbauer(3, a, x)
x**3*(4*a**3/3 + 4*a**2 + 8*a/3) + x*(-2*a**2 - 2*a)
```

```
>>> gegenbauer(n, a, x)
gegenbauer(n, a, x)
>>> gegenbauer(n, a, -x)
(-1)**n*gegenbauer(n, a, x)
```

```
>>> gegenbauer(n, a, 0)
2**n*sqrt(pi)*gamma(a + n/2)/(gamma(a)*gamma(-n/2 + 1/2)*gamma(n + 1))
>>> gegenbauer(n, a, 1)
gamma(2*a + n)/(gamma(2*a)*gamma(n + 1))
```

```
>>> conjugate(gegenbauer(n, a, x))
gegenbauer(n, conjugate(a), conjugate(x))
```

```
>>> diff(gegenbauer(n, a, x), x)
2*a*gegenbauer(n - 1, a + 1, x)
```

Chebyshev Polynomials

class `diofant.functions.special.polynomials.chebyshevt`

Chebyshev polynomial of the first kind, $T_n(x)$

`chebyshevt(n, x)` gives the n th Chebyshev polynomial (of the first kind) in x , $T_n(x)$.

The Chebyshev polynomials of the first kind are orthogonal on $[-1, 1]$ with respect to the weight $\frac{1}{\sqrt{1-x^2}}$.

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R302] (page 1268), [R303] (page 1268), [R304] (page 1268), [R305] (page 1268), [R306] (page 1268)

Examples

```
>>> from diofant import chebyshevt, chebyshevu, diff
>>> from diofant.abc import n,x
>>> chebyshevt(0, x)
1
>>> chebyshevt(1, x)
x
>>> chebyshevt(2, x)
2*x**2 - 1
```

```
>>> chebyshevt(n, x)
chebyshevt(n, x)
>>> chebyshevt(n, -x)
(-1)**n*chebyshevt(n, x)
>>> chebyshevt(-n, x)
chebyshevt(n, x)
```

```
>>> chebyshevt(n, 0)
cos(pi*n/2)
>>> chebyshevt(n, -1)
(-1)**n
```

```
>>> diff(chebyshevt(n, x), x)
n*chebyshevu(n - 1, x)
```

class `diofant.functions.special.polynomials.chebyshevu`

Chebyshev polynomial of the second kind, $U_n(x)$

`chebyshevu(n, x)` gives the n th Chebyshev polynomial of the second kind in x , $U_n(x)$.

The Chebyshev polynomials of the second kind are orthogonal on $[-1, 1]$ with respect to the weight $\sqrt{1-x^2}$.

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[\[R307\]](#) (page 1268), [\[R308\]](#) (page 1268), [\[R309\]](#) (page 1268), [\[R310\]](#) (page 1268), [\[R311\]](#) (page 1268)

Examples

```
>>> from diofant import chebyshevt, chebyshevu, diff
>>> from diofant.abc import n,x
>>> chebyshevu(0, x)
1
>>> chebyshevu(1, x)
2*x
>>> chebyshevu(2, x)
4*x**2 - 1
```

```
>>> chebyshevu(n, x)
chebyshevu(n, x)
>>> chebyshevu(n, -x)
(-1)**n*chebyshevu(n, x)
>>> chebyshevu(-n, x)
-chebyshevu(n - 2, x)
```

```
>>> chebyshevu(n, 0)
cos(pi*n/2)
>>> chebyshevu(n, 1)
n + 1
```

```
>>> diff(chebyshev(n, x), x)
(-x*chebyshev(n, x) + (n + 1)*chebyshevt(n + 1, x))/(x**2 - 1)
```

class diofant.functions.special.polynomials.chebyshevt_root

`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`.

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshev](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshev_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

Examples

```
>>> from diofant import chebyshevt, chebyshevt_root
>>> chebyshevt_root(3, 2)
-sqrt(3)/2
>>> chebyshevt(3, chebyshevt_root(3, 2))
0
```

class diofant.functions.special.polynomials.chebyshevu_root

`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`.

See also:

[chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshev](#) (page 415), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.chebyshev_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

Examples

```
>>> from diofant import chebyshevu, chebyshevu_root
>>> chebyshevu_root(3, 2)
-sqrt(2)/2
>>> chebyshevu(3, chebyshevu_root(3, 2))
0
```

Legendre Polynomials

class `diofant.functions.special.polynomials.Legendre`

`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 .

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R312] (page 1268), [R313] (page 1268), [R314] (page 1268), [R315] (page 1268)

Examples

```
>>> from diofant import legendre, diff
>>> from diofant.abc import x, n
>>> legendre(0, x)
1
>>> legendre(1, x)
x
>>> legendre(2, x)
3*x**2/2 - 1/2
>>> legendre(n, x)
legendre(n, x)
>>> diff(legendre(n,x), x)
n*(x*legendre(n, x) - legendre(n - 1, x))/(x**2 - 1)
```

class `diofant.functions.special.polynomials.assoc_Legendre`

`assoc_legendre(n,m, x)` gives $P_n^m(x)$, where n and m are the degree and order or an expression which is related to the 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 .

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#)

(page 417), *legendre* (page 417), *hermite* (page 419), *laguerre* (page 420), *assoc_laguerre* (page 420), *diofant.polys.orthopolys.jacobi_poly* (page 736), *diofant.polys.orthopolys.gegenbauer_poly* (page 736), *diofant.polys.orthopolys.chebyshevt_poly* (page 736), *diofant.polys.orthopolys.chebyshevu_poly* (page 736), *diofant.polys.orthopolys.hermite_poly* (page 736), *diofant.polys.orthopolys.legendre_poly* (page 736), *diofant.polys.orthopolys.laguerre_poly* (page 736)

References

[R316] (page 1268), [R317] (page 1268), [R318] (page 1268), [R319] (page 1268)

Examples

```
>>> from diofant import assoc_legendre
>>> from diofant.abc import x, m, n
>>> assoc_legendre(0,0, x)
1
>>> assoc_legendre(1,0, x)
x
>>> assoc_legendre(1,1, x)
-sqrt(-x**2 + 1)
>>> assoc_legendre(n,m,x)
assoc_legendre(n, m, x)
```

Hermite Polynomials

class `diofant.functions.special.polynomials.hermite`

`hermite(n, x)` gives the n th Hermite polynomial in x , $H_n(x)$

The Hermite polynomials are orthogonal on $(-\infty, \infty)$ with respect to the weight $\exp\left(-\frac{x^2}{2}\right)$.

See also:

jacobi (page 412), *gegenbauer* (page 414), *chebyshevt* (page 415), *chebyshevt_root* (page 416), *chebyshevu* (page 415), *chebyshevu_root* (page 417), *legendre* (page 417), *assoc_legendre* (page 418), *laguerre* (page 420), *assoc_laguerre* (page 420), *diofant.polys.orthopolys.jacobi_poly* (page 736), *diofant.polys.orthopolys.gegenbauer_poly* (page 736), *diofant.polys.orthopolys.chebyshevt_poly* (page 736), *diofant.polys.orthopolys.chebyshevu_poly* (page 736), *diofant.polys.orthopolys.hermite_poly* (page 736), *diofant.polys.orthopolys.legendre_poly* (page 736), *diofant.polys.orthopolys.laguerre_poly* (page 736)

References

[R320] (page 1268), [R321] (page 1268), [R322] (page 1268)

Examples

```
>>> from diofant import hermite, diff
>>> from diofant.abc import x, n
>>> hermite(0, x)
1
>>> hermite(1, x)
2*x
>>> hermite(2, x)
4*x**2 - 2
>>> hermite(n, x)
hermite(n, x)
>>> diff(hermite(n,x), x)
2*n*hermite(n - 1, x)
>>> hermite(n, -x)
(-1)**n*hermite(n, x)
```

Laguerre Polynomials

class `diofant.functions.special.polynomials.laguerre`

Returns the n th Laguerre polynomial in x , $L_n(x)$.

Parameters `n` : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [assoc_laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R323] (page 1268), [R324] (page 1268), [R325] (page 1268), [R326] (page 1268)

Examples

```
>>> from diofant import laguerre, diff
>>> from diofant.abc import x, n
>>> laguerre(0, x)
1
>>> laguerre(1, x)
-x + 1
>>> laguerre(2, x)
x**2/2 - 2*x + 1
```



```
>>> laguerre(3, x)
-x**3/6 + 3*x**2/2 - 3*x + 1
```

```
>>> laguerre(n, x)
laguerre(n, x)
```

```
>>> diff(laguerre(n, x), x)
-assoc_laguerre(n - 1, 1, x)
```

class `diofant.functions.special.polynomials.assoc_laguerre`

Returns the n th generalized Laguerre polynomial in x , $L_n(x)$.

Parameters n : int

Degree of Laguerre polynomial. Must be $n \geq 0$.

alpha : Expr

Arbitrary expression. For $\alpha=0$ regular Laguerre polynomials will be generated.

See also:

[jacobi](#) (page 412), [gegenbauer](#) (page 414), [chebyshevt](#) (page 415), [chebyshevt_root](#) (page 416), [chebyshevu](#) (page 415), [chebyshevu_root](#) (page 417), [legendre](#) (page 417), [assoc_legendre](#) (page 418), [hermite](#) (page 419), [laguerre](#) (page 420), [diofant.polys.orthopolys.jacobi_poly](#) (page 736), [diofant.polys.orthopolys.gegenbauer_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevt_poly](#) (page 736), [diofant.polys.orthopolys.chebyshevu_poly](#) (page 736), [diofant.polys.orthopolys.hermite_poly](#) (page 736), [diofant.polys.orthopolys.legendre_poly](#) (page 736), [diofant.polys.orthopolys.laguerre_poly](#) (page 736)

References

[R327] (page 1268), [R328] (page 1268), [R329] (page 1268), [R330] (page 1268)

Examples

```
>>> from diofant import laguerre, assoc_laguerre, diff
>>> from diofant.abc import x, n, a
>>> assoc_laguerre(0, a, x)
1
>>> assoc_laguerre(1, a, x)
a - x + 1
>>> assoc_laguerre(2, a, x)
a**2/2 + 3*a/2 + x**2/2 + x*(-a - 2) + 1
>>> assoc_laguerre(3, a, x)
a**3/6 + a**2 + 11*a/6 - x**3/6 + x**2*(a/2 + 3/2) +
x*(-a**2/2 - 5*a/2 - 3) + 1
```

```
>>> assoc_laguerre(n, a, 0)
binomial(a + n, a)
```

```
>>> assoc_laguerre(n, a, x)
assoc_laguerre(n, a, x)
```

```
>>> assoc_laguerre(n, 0, x)
laguerre(n, x)
```

```
>>> diff(assoc_laguerre(n, a, x), x)
-assoc_laguerre(n - 1, a + 1, x)
```

```
>>> diff(assoc_laguerre(n, a, x), a)
Sum(assoc_laguerre(_k, a, x)/(-a + n), (_k, 0, n - 1))
```

Spherical Harmonics

class `diofant.functions.special.spherical_harmonics.Ynm`
Spherical harmonics defined as

$$Y_n^m(\theta, \varphi) := \sqrt{\frac{(2n+1)(n-m)!}{4\pi(n+m)!}} \exp(im\varphi) P_n^m(\cos(\theta))$$

`Ynm()` gives the spherical harmonic function of order n and m in θ and φ , $Y_n^m(\theta, \varphi)$. The four parameters are as follows: $n \geq 0$ an integer and m an integer such that $-n \leq m \leq n$ holds. The two angles are real-valued with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$.

See also:

[diofant.functions.special.spherical_harmonics.Ynm_c](#) (page 424), *[diofant.functions.special.spherical_harmonics.Znm](#)* (page 424)

References

[\[R331\]](#) (page 1268), [\[R332\]](#) (page 1268), [\[R333\]](#) (page 1268), [\[R334\]](#) (page 1268)

Examples

```
>>> from diofant import Ynm, Symbol
>>> from diofant.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, theta, phi)
Ynm(n, m, theta, phi)
```

Several symmetries are known, for the order

```
>>> from diofant import Ynm, Symbol
>>> from diofant.abc import n,m
>>> 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

```
>>> from diofant import Ynm, Symbol, simplify
>>> from diofant.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> Ynm(n, m, -theta, phi)
Ynm(n, m, theta, phi)
```

```
>>> Ynm(n, m, theta, -phi)
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

```
>>> from diofant import Ynm, Symbol, diff
>>> from diofant.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> diff(Ynm(n, m, theta, phi), theta)
m*cot(theta)*Ynm(n, m, theta, phi) + 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

```
>>> from diofant import Ynm, Symbol, conjugate
>>> from diofant.abc import n,m
>>> theta = Symbol("theta")
>>> phi = Symbol("phi")
```

```
>>> 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

```
>>> from diofant import Ynm, Symbol, expand_func
>>> from diofant.abc import n,m
>>> 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))
```

`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 422), [diofant.functions.special.spherical_harmonics.Znm](#) (page 424)

References

[R335] (page 1268), [R336] (page 1268), [R337] (page 1268)

class `diofant.functions.special.spherical_harmonics.Znm`
 Real spherical harmonics defined as

$$Z_n^m(\theta, \varphi) := \begin{cases} \frac{Y_n^m(\theta, \varphi) + \overline{Y_n^m(\theta, \varphi)}}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - \overline{Y_n^m(\theta, \varphi)}}{i\sqrt{2}} & m < 0 \end{cases}$$

which gives in simplified form

$$Z_n^m(\theta, \varphi) = \begin{cases} \frac{Y_n^m(\theta, \varphi) + (-1)^m Y_n^{-m}(\theta, \varphi)}{\sqrt{2}} & m > 0 \\ Y_n^m(\theta, \varphi) & m = 0 \\ \frac{Y_n^m(\theta, \varphi) - (-1)^m Y_n^{-m}(\theta, \varphi)}{i\sqrt{2}} & m < 0 \end{cases}$$

See also:

[diofant.functions.special.spherical_harmonics.Ynm](#) (page 422), [diofant.functions.special.spherical_harmonics.Ynm_c](#) (page 424)

References

[R338] (page 1268), [R339] (page 1269), [R340] (page 1269)

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 425)

`diofant.functions.special.tensor_functions.eval_levicivita(*args)`
Evaluate Levi-Civita symbol.

class `diofant.functions.special.tensor_functions.LeviCivita`
Represent the Levi-Civita symbol.

For even permutations of indices it returns 1, for odd permutations -1, and for everything else (a repeated index) it returns 0.

Thus it represents an alternating pseudotensor.

See also:

[diofant.functions.special.tensor_functions.Eijk](#) (page 424)

Examples

```
>>> from diofant import LeviCivita
>>> from diofant.abc import i, j, k
>>> LeviCivita(1, 2, 3)
1
>>> LeviCivita(1, 3, 2)
-1
>>> LeviCivita(1, 2, 2)
0
>>> LeviCivita(i, j, k)
LeviCivita(i, j, k)
>>> LeviCivita(i, j, i)
0
```

class `diofant.functions.special.tensor_functions.KroneckerDelta`
The discrete, or Kronecker, delta function.

A function that takes in two integers i and j . It returns 0 if i and j are not equal or it returns 1 if i and j are equal.

Parameters i : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

See also:

[diofant.functions.special.tensor_functions.KroneckerDelta.eval](#) (page 426),
[diofant.functions.special.delta_functions.DiracDelta](#) (page 357)

References

[R341] (page 1269)

Examples

A simple example with integer indices:

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from diofant.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

classmethod eval(*i, j*)

Evaluates the discrete delta function.

Examples

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indices_contain_equal_information

Returns True if indices are either both above or below fermi.

Examples

```

>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False

```

is_above_fermi

True if Delta can be non-zero above fermi

See also:

[diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi](#) (page 427), [diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi](#) (page 428), [diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi](#) (page 427)

Examples

```

>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True

```

is_below_fermi

True if Delta can be non-zero below fermi

See also:

[diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi](#) (page 427), [diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi](#) (page 427), [diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi](#) (page 428)

Examples

```

>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)

```

```
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

is_only_above_fermi

True if Delta is restricted to above fermi

See also:

diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi (page 427), *diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi* (page 427), *diofant.functions.special.tensor_functions.KroneckerDelta.is_only_below_fermi* (page 428)

Examples

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

is_only_below_fermi

True if Delta is restricted to below fermi

See also:

diofant.functions.special.tensor_functions.KroneckerDelta.is_above_fermi (page 427), *diofant.functions.special.tensor_functions.KroneckerDelta.is_below_fermi* (page 427), *diofant.functions.special.tensor_functions.KroneckerDelta.is_only_above_fermi* (page 427)

Examples

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
```



```
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

killable_index

Returns the index which is preferred to substitute in the final expression.

The index to substitute is the index with less information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[diofant.functions.special.tensor_functions.KroneckerDelta.preferred_index](#) (page 429)

Examples

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

preferred_index

Returns the index which is preferred to keep in the final expression.

The preferred index is the index with more information regarding fermi level. If indices contain same information, 'a' is preferred before 'b'.

See also:

[diofant.functions.special.tensor_functions.KroneckerDelta.killable_index](#) (page 428)

Examples

```
>>> from diofant.functions.special.tensor_functions import KroneckerDelta
>>> from diofant import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

3.8 Geometry

3.8.1 Introduction

The geometry module for Diofant allows one to create two-dimensional geometrical entities, such as lines and circles, and query for information about these entities. This could include asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines. The primary use case of the module involves entities with numerical values, but it is possible to also use symbolic representations.

3.8.2 Available Entities

The following entities are currently available in the geometry module:

- Point
- Line, Ray, Segment
- Ellipse, Circle
- Polygon, RegularPolygon, Triangle

Most of the work one will do will be through the properties and methods of these entities, but several global methods exist:

- `intersection(entity1, entity2)`
- `are_similar(entity1, entity2)`
- `convex_hull(points)`

For a full API listing and an explanation of the methods and their return values please see the list of classes at the end of this document.

3.8.3 Example Usage

The following Python session gives one an idea of how to work with some of the geometry module.

```
>>> from diofant import *
>>> from diofant.geometry import *
>>> x = Point(0, 0)
>>> y = Point(1, 1)
>>> z = Point(2, 2)
>>> zp = Point(1, 0)
>>> Point.is_collinear(x, y, z)
True
>>> Point.is_collinear(x, y, zp)
False
>>> t = Triangle(zp, y, x)
>>> t.area
1/2
>>> t.medians[x]
Segment(Point2D(0, 0), Point2D(1, 1/2))
>>> Segment(Point(1, Rational(1, 2)), Point(0, 0))
Segment(Point2D(0, 0), Point2D(1, 1/2))
>>> m = t.medians
>>> intersection(m[x], m[y], m[zp])
```

```
[Point2D(2/3, 1/3)]
>>> c = Circle(x, 5)
>>> l = Line(Point(5, -5), Point(5, 5))
>>> c.is_tangent(l) # is l tangent to c?
True
>>> l = Line(x, y)
>>> c.is_tangent(l) # is l tangent to c?
False
>>> intersection(c, l)
[Point2D(-5*sqrt(2)/2, -5*sqrt(2)/2), Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
```

3.8.4 Intersection of medians

```
>>> from diofant import symbols
>>> from diofant.geometry import Point, Triangle, intersection

>>> a, b = symbols("a,b", positive=True)

>>> x = Point(0, 0)
>>> y = Point(a, 0)
>>> z = Point(2*a, b)
>>> t = Triangle(x, y, z)

>>> t.area
a*b/2

>>> t.medians[x]
Segment(Point2D(0, 0), Point2D(3*a/2, b/2))

>>> intersection(t.medians[x], t.medians[y], t.medians[z])
[Point2D(a, b/3)]
```

3.8.5 An in-depth example: Pappus' Hexagon Theorem

From Wikipedia ([\[WikiPappus\]](#) (page 1269)):

Given one set of collinear points A, B, C , and another set of collinear points a, b, c , then the intersection points X, Y, Z of line pairs Ab and aB , Ac and aC , Bc and bC are collinear.

```
>>> from diofant import *
>>> from diofant.geometry import *
>>>
>>> l1 = Line(Point(0, 0), Point(5, 6))
>>> l2 = Line(Point(0, 0), Point(2, -2))
>>>
>>> def subs_point(l, val):
...     """Take an arbitrary point and make it a fixed point."""
...     t = Symbol('t', extended_real=True)
...     ap = l.arbitrary_point()
...     return Point(ap.x.subs(t, val), ap.y.subs(t, val))
...
>>> p11 = subs_point(l1, 5)
>>> p12 = subs_point(l1, 6)
```

```
>>> p13 = subs_point(l1, 11)
>>>
>>> p21 = subs_point(l2, -1)
>>> p22 = subs_point(l2, 2)
>>> p23 = subs_point(l2, 13)
>>>
>>> ll1 = Line(p11, p22)
>>> ll2 = Line(p11, p23)
>>> ll3 = Line(p12, p21)
>>> ll4 = Line(p12, p23)
>>> ll5 = Line(p13, p21)
>>> ll6 = Line(p13, p22)
>>>
>>> pp1 = intersection(ll1, ll3)[0]
>>> pp2 = intersection(ll2, ll5)[0]
>>> pp3 = intersection(ll4, ll6)[0]
>>>
>>> Point.is_collinear(pp1, pp2, pp3)
True
```

References

3.8.6 Miscellaneous Notes

- The area property of Polygon and Triangle may return a positive or negative value, depending on whether or not the points are oriented counter-clockwise or clockwise, respectively. If you always want a positive value be sure to use the abs function.
- Although Polygon can refer to any type of polygon, the code has been written for simple polygons. Hence, expect potential problems if dealing with complex polygons (overlapping sides).
- Since Diofant is still in its infancy some things may not simplify properly and hence some things that should return True (e.g., Point.is_collinear) may not actually do so. Similarly, attempting to find the intersection of entities that do intersect may result in an empty result.

3.8.7 Future Work

Truth Setting Expressions

When one deals with symbolic entities, it often happens that an assertion cannot be guaranteed. For example, consider the following code:

```
>>> from diofant import *
>>> from diofant.geometry import *
>>> x,y,z = map(Symbol, 'xyz')
>>> p1,p2,p3 = Point(x, y), Point(y, z), Point(2*x*y, y)
>>> Point.is_collinear(p1, p2, p3)
False
```

Even though the result is currently False, this is not *always* true. If the quantity $z - y - 2 * y * z + 2 * y * *2 == 0$ then the points will be collinear. It would be really nice to inform the user of this because such a quantity may be useful to a user for further calculation and, at

the very least, being nice to know. This could be potentially done by returning an object (e.g., `GeometryResult`) that the user could use. This actually would not involve an extensive amount of work.

Three Dimensions and Beyond

Currently there are no plans for extending the module to three dimensions, but it certainly would be a good addition. This would probably involve a fair amount of work since many of the algorithms used are specific to two dimensions.

Submodules

Entities

class `diofant.geometry.entity.GeometryEntity`

The base class for all geometrical entities.

This class doesn't represent any particular geometric entity, it only provides the implementation of some methods common to all subclasses.

ambient_dimension

What is the dimension of the space that the object is contained in?

encloses(*o*)

Return True if *o* is inside (not on or outside) the boundaries of self.

The object will be decomposed into Points and individual Entities need only define an `encloses_point` method for their class.

See also:

[`diofant.geometry.ellipse.Ellipse.encloses_point`](#) (page 487), [`diofant.geometry.polygon.Polygon.encloses_point`](#) (page 502)

Examples

```
>>> from diofant import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t2 = Polygon(*RegularPolygon(Point(0, 0), 2, 3).vertices)
>>> t2.encloses(t)
True
>>> t.encloses(t2)
False
```

intersection(*o*)

Returns a list of all of the intersections of self with *o*.

See also:

[`diofant.geometry.util.intersection`](#) (page 435)

Notes

An entity is not required to implement this method.

is_similar(*other*)

Is this geometrical entity similar to another geometrical entity?

Two entities are similar if a uniform scaling (enlarging or shrinking) of one of the entities will allow one to obtain the other.

See also:

[scale](#) (page 434)

Notes

This method is not intended to be used directly but rather through the *are_similar* function found in *util.py*. An entity is not required to implement this method. If two different types of entities can be similar, it is only required that one of them be able to determine this.

rotate(*angle*, *pt=None*)

Rotate *angle* radians counterclockwise about Point *pt*.

The default *pt* is the origin, Point(0, 0)

See also:

[scale](#) (page 434), [translate](#) (page 434)

Examples

```
>>> from diofant import Point, RegularPolygon, Polygon, pi
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t # vertex on x axis
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.rotate(pi/2) # vertex on y axis now
Triangle(Point2D(0, 1), Point2D(-sqrt(3)/2, -1/2), Point2D(sqrt(3)/2, -1/2))
```

scale(*x=1*, *y=1*, *pt=None*)

Scale the object by multiplying the x,y-coordinates by *x* and *y*.

If *pt* is given, the scaling is done relative to that point; the object is shifted by *-pt*, scaled, and shifted by *pt*.

See also:

[rotate](#) (page 434), [translate](#) (page 434)

Examples

```
>>> from diofant import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.scale(2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)/2), Point2D(-1, -sqrt(3)/2))
>>> t.scale(2,2)
Triangle(Point2D(2, 0), Point2D(-1, sqrt(3)), Point2D(-1, -sqrt(3)))
```

translate($x=0, y=0$)

Shift the object by adding to the x,y-coordinates the values x and y.

See also:

[rotate](#) (page 434), [scale](#) (page 434)

Examples

```
>>> from diofant import RegularPolygon, Point, Polygon
>>> t = Polygon(*RegularPolygon(Point(0, 0), 1, 3).vertices)
>>> t
Triangle(Point2D(1, 0), Point2D(-1/2, sqrt(3)/2), Point2D(-1/2, -sqrt(3)/2))
>>> t.translate(2)
Triangle(Point2D(3, 0), Point2D(3/2, sqrt(3)/2), Point2D(3/2, -sqrt(3)/2))
>>> t.translate(2, 2)
Triangle(Point2D(3, 2), Point2D(3/2, sqrt(3)/2 + 2),
Point2D(3/2, -sqrt(3)/2 + 2))
```

Utils

`diofant.geometry.util.idiff`(*eq, y, x, n=1*)

Return dy/dx assuming that $eq == 0$.

Parameters *y* : the dependent variable or a list of dependent variables (with y first)

x : the variable that the derivative is being taken with respect to

n : the order of the derivative (default is 1)

See also:

[diofant.core.function.Derivative](#) (page 129) represents unevaluated derivatives

[diofant.core.function.diff](#) (page 132) explicitly differentiates wrt symbols

Examples

```
>>> from diofant.abc import x, y, a
>>> from diofant.geometry.util import idiff
```

```
>>> circ = x**2 + y**2 - 4
>>> idiff(circ, y, x)
-x/y
>>> idiff(circ, y, x, 2).simplify()
-(x**2 + y**2)/y**3
```

Here, a is assumed to be independent of x:

```
>>> idiff(x + a + y, y, x)
-1
```

Now the x-dependence of a is made explicit by listing a after y in a list.

```
>>> idiff(x + a + y, [y, a], x)
-Derivative(a, x) - 1
```

`diofant.geometry.util.intersection(*entities)`

The intersection of a collection of GeometryEntity instances.

Parameters `entities` : sequence of GeometryEntity

Returns `intersection` : list of GeometryEntity

Raises `NotImplementedError`

When unable to calculate intersection.

See also:

[diofant.geometry.entity.GeometryEntity.intersection](#) (page 433)

Notes

The intersection of any geometrical entity with itself should return a list with one item: the entity in question. An intersection requires two or more entities. If only a single entity is given then the function will return an empty list. It is possible for *intersection* to miss intersections that one knows exists because the required quantities were not fully simplified internally. Reals should be converted to Rationals, e.g. `Rational(str(real_num))` or else failures due to floating point issues may result.

Examples

```
>>> from diofant.geometry import Point, Line, Circle, intersection
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 5)
>>> l1, l2 = Line(p1, p2), Line(p3, p2)
>>> c = Circle(p2, 1)
>>> intersection(l1, p2)
[Point2D(1, 1)]
>>> intersection(l1, l2)
[Point2D(1, 1)]
>>> intersection(c, p2)
[]
>>> intersection(c, Point(1, 0))
[Point2D(1, 0)]
>>> intersection(c, l2)
[Point2D(-sqrt(5)/5 + 1, 2*sqrt(5)/5 + 1),
 Point2D(sqrt(5)/5 + 1, -2*sqrt(5)/5 + 1)]
```

`diofant.geometry.util.convex_hull(*args)`

The convex hull surrounding the Points contained in the list of entities.

Parameters `args` : a collection of Points, Segments and/or Polygons

Returns `convex_hull` : Polygon

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.polygon.Polygon](#) (page 498)

Notes

This can only be performed on a set of non-symbolic points.

References

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

[2] Andrew's Monotone Chain Algorithm (A.M. Andrew, "Another Efficient Algorithm for Convex Hulls in Two Dimensions", 1979) http://geomalgorithms.com/a10-_hull-1.html

Examples

```

>>> from diofant.geometry import Point, convex_hull
>>> points = [(1,1), (1,2), (3,1), (-5,2), (15,4)]
>>> convex_hull(*points)
Polygon(Point2D(-5, 2), Point2D(1, 1), Point2D(3, 1), Point2D(15, 4))

```

`diofant.geometry.util.are_similar(e1, e2)`

Are two geometrical entities similar.

Can one geometrical entity be uniformly scaled to the other?

Parameters `e1` : GeometryEntity

`e2` : GeometryEntity

Returns `are_similar` : boolean

Raises **GeometryError**

When `e1` and `e2` cannot be compared.

See also:

`diofant.geometry.entity.GeometryEntity.is_similar` (page 433)

Notes

If the two objects are equal then they are similar.

Examples

```

>>> from diofant import Point, Circle, Triangle, are_similar
>>> c1, c2 = Circle(Point(0, 0), 4), Circle(Point(1, 4), 3)
>>> t1 = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))
>>> t2 = Triangle(Point(0, 0), Point(2, 0), Point(0, 2))
>>> t3 = Triangle(Point(0, 0), Point(3, 0), Point(0, 1))
>>> are_similar(t1, t2)
True
>>> are_similar(t1, t3)
False

```

`diofant.geometry.util.centroid(*args)`

Find the centroid (center of mass) of the collection containing only Points, Segments or Polygons. The centroid is the weighted average of the individual centroid where the weights are the lengths (of segments) or areas (of polygons). Overlapping regions will add to the weight of that region.

If there are no objects (or a mixture of objects) then None is returned.

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Segment](#) (page 463), [diofant.geometry.polygon.Polygon](#) (page 498)

Examples

```
>>> from diofant import Point, Segment, Polygon
>>> from diofant.geometry.util import centroid
>>> p = Polygon((0, 0), (10, 0), (10, 10))
>>> q = p.translate(0, 20)
>>> p.centroid, q.centroid
(Point2D(20/3, 10/3), Point2D(20/3, 70/3))
>>> centroid(p, q)
Point2D(20/3, 40/3)
>>> p, q = Segment((0, 0), (2, 0)), Segment((0, 0), (2, 2))
>>> centroid(p, q)
Point2D(1, -sqrt(2) + 2)
>>> centroid(Point(0, 0), Point(2, 0))
Point2D(1, 0)
```

Stacking 3 polygons on top of each other effectively triples the weight of that polygon:

```
>>> p = Polygon((0, 0), (1, 0), (1, 1), (0, 1))
>>> q = Polygon((1, 0), (3, 0), (3, 1), (1, 1))
>>> centroid(p, q)
Point2D(3/2, 1/2)
>>> centroid(p, p, p, q) # centroid x-coord shifts left
Point2D(11/10, 1/2)
```

Stacking the squares vertically above and below p has the same effect:

```
>>> centroid(p, p.translate(0, 1), p.translate(0, -1), q)
Point2D(11/10, 1/2)
```

Points

class `diofant.geometry.point.Point`

A point in a n-dimensional Euclidean space.

Parameters `coords` : sequence of n-coordinate values. In the special case where `n=2` or `3`, a `Point2D` or `Point3D` will be created as appropriate.

Raises `TypeError`

When trying to add or subtract points with different dimensions.
When `intersection` is called with object other than a `Point`.

See also:

[diofant.geometry.line.Segment](#) (page 463) Connects two Points

Examples

```
>>> from diofant.geometry import Point
>>> from diofant.abc import x
>>> Point(1, 2, 3)
Point3D(1, 2, 3)
>>> Point([1, 2])
Point2D(1, 2)
>>> Point(0, x)
Point2D(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point(0.5, 0.25)
Point2D(1/2, 1/4)
>>> print(Point(0.5, 0.25, evaluate=False))
Point2D(0.5, 0.25)
```

Attributes

[length](#) (page 441)

Treating a Point as a Line, this returns 0 for the length of a Point.

origin: A *Point* representing the origin of the appropriately-dimensional space.

ambient_dimension

The dimension of the ambient space the point is in. I.e., if the point is in \mathbb{R}^n , the ambient dimension will be n

distance(*p*)

The Euclidean distance from self to point *p*.

Parameters *p* : Point

Returns **distance** : number or symbolic expression.

See also:

[diofant.geometry.line.Segment.length](#) (page 464)

Examples

```
>>> from diofant.geometry import Point
>>> p1, p2 = Point(1, 1), Point(4, 5)
>>> p1.distance(p2)
5
```

```
>>> from diofant.abc import x, y
>>> p3 = Point(x, y)
```

```
>>> p3.distance(Point(0, 0))
sqrt(x**2 + y**2)
```

dot(*p2*)

Return dot product of self with another Point.

equals(*other*)

Returns whether the coordinates of self and other agree.

evalf(*prec=None, **options*)

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

Returns point : Point

Examples

```
>>> from diofant import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> print(p1.evalf())
Point2D(0.5, 1.5)
```

intersection(*o*)

The intersection between this point and another point.

Parameters other : Point

Returns intersection : list of Points

Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

```
>>> from diofant import Point
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point2D(0, 0)]
```

is_collinear(**args*)

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

Parameters points : sequence of Point

Returns `is_collinear` : boolean

See also:

[diofant.geometry.line.Line](#) (page 457)

Notes

Slope is preserved everywhere on a line, so the slope between any two points on the line should be the same. Take the first two points, `p1` and `p2`, and create a translated point `v1` with `p1` as the origin. Now for every other point we create a translated point, `vi` with `p1` also as the origin. Note that these translations preserve slope since everything is consistently translated to a new origin of `p1`. Since slope is preserved then we have the following equality:

- `v1_slope = vi_slope`
- `v1.y/v1.x = vi.y/vi.x` (due to translation)
- `v1.y*vi.x = vi.y*v1.x`
- `v1.y*vi.x - vi.y*v1.x = 0 (*)`

Hence, if we have a `vi` such that the equality in (*) is False then the points are not collinear. We do this test for every point in the list, and if all pass then they are collinear.

Examples

```
>>> from diofant import Point
>>> from diofant.abc import x
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4, p5 = Point(2, 2), Point(x, x), Point(1, 2)
>>> Point.is_collinear(p1, p2, p3, p4)
True
>>> Point.is_collinear(p1, p2, p3, p5)
False
```

`is_scalar_multiple(other)`

Returns whether `self` and `other` are scalar multiples of each other.

`is_zero`

True if every coordinate is zero, otherwise False.

`length`

Treating a Point as a Line, this returns 0 for the length of a Point.

Examples

```
>>> from diofant import Point
>>> p = Point(0, 1)
>>> p.length
0
```

`midpoint(p)`

The midpoint between self and point `p`.

Parameters `p` : Point

Returns `midpoint` : Point

See also:

[diofant.geometry.line.Segment.midpoint](#) (page 465)

Examples

```
>>> from diofant.geometry import Point
>>> p1, p2 = Point(1, 1), Point(13, 5)
>>> p1.midpoint(p2)
Point2D(7, 3)
```

`n(prec=None, **options)`

Evaluate the coordinates of the point.

This method will, where possible, create and return a new Point where the coordinates are evaluated as floating point numbers to the precision indicated (default=15).

Returns `point` : Point

Examples

```
>>> from diofant import Point, Rational
>>> p1 = Point(Rational(1, 2), Rational(3, 2))
>>> p1
Point2D(1/2, 3/2)
>>> print(p1.evalf())
Point2D(0.5, 1.5)
```

origin

A point of all zeros of the same ambient dimension as the current point

class diofant.geometry.point.Point2D

A point in a 2-dimensional Euclidean space.

Parameters `coords` : sequence of 2 coordinate values.

Raises TypeError

When trying to add or subtract points with different dimensions.

When trying to create a point with more than two dimensions. When *intersection* is called with object other than a Point.

See also:

[diofant.geometry.line.Segment](#) (page 463) Connects two Points

Examples

```
>>> from diofant.geometry import Point2D
>>> from diofant.abc import x
>>> Point2D(1, 2)
Point2D(1, 2)
```

```
>>> Point2D([1, 2])
Point2D(1, 2)
>>> Point2D(0, x)
Point2D(0, x)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point2D(0.5, 0.25)
Point2D(1/2, 1/4)
>>> print(Point2D(0.5, 0.25, evaluate=False))
Point2D(0.5, 0.25)
```

Attributes

<code>x</code> (page 445)	Returns the X coordinate of the Point.
<code>y</code> (page 445)	Returns the Y coordinate of the Point.

`diofant.geometry.Point.length`

bounds

Return a tuple (xmin, ymin, xmax, ymax) representing the bounding rectangle for the geometric figure.

is_concyclic(*points)

Is a sequence of points concyclic?

Test whether or not a sequence of points are concyclic (i.e., they lie on a circle).

Parameters `points` : sequence of Points

Returns `is_concyclic` : boolean

True if points are concyclic, False otherwise.

See also:

[diofant.geometry.ellipse.Circle](#) (page 495)

Notes

No points are not considered to be concyclic. One or two points are definitely concyclic and three points are concyclic iff they are not collinear.

For more than three points, create a circle from the first three points. If the circle cannot be created (i.e., they are collinear) then all of the points cannot be concyclic. If the circle is created successfully then simply check the remaining points for containment in the circle.

Examples

```
>>> from diofant.geometry import Point
>>> p1, p2 = Point(-1, 0), Point(1, 0)
>>> p3, p4 = Point(0, 1), Point(-1, 2)
>>> Point.is_concyclic(p1, p2, p3)
```

```
True
>>> Point.is_concyclic(p1, p2, p3, p4)
False
```

rotate(*angle*, *pt=None*)

Rotate angle radians counterclockwise about Point *pt*.

See also:

[rotate](#) (page 443), [scale](#) (page 444)

Examples

```
>>> from diofant import Point2D, pi
>>> t = Point2D(1, 0)
>>> t.rotate(pi/2)
Point2D(0, 1)
>>> t.rotate(pi/2, (2, 0))
Point2D(2, -1)
```

scale(*x=1*, *y=1*, *pt=None*)

Scale the coordinates of the Point by multiplying by *x* and *y* after subtracting *pt* - default is (0, 0) - and then adding *pt* back again (i.e. *pt* is the point of reference for the scaling).

See also:

[rotate](#) (page 443), [translate](#) (page 444)

Examples

```
>>> from diofant import Point2D
>>> t = Point2D(1, 1)
>>> t.scale(2)
Point2D(2, 1)
>>> t.scale(2, 2)
Point2D(2, 2)
```

transform(*matrix*)

Return the point after applying the transformation described by the 3x3 Matrix, *matrix*.

See also:

[diofant.geometry.entity.GeometryEntity.rotate](#) (page 434), [diofant.geometry.entity.GeometryEntity.scale](#) (page 434), [diofant.geometry.entity.GeometryEntity.translate](#) (page 434)

translate(*x=0*, *y=0*)

Shift the Point by adding *x* and *y* to the coordinates of the Point.

See also:

[rotate](#) (page 443), [scale](#) (page 444)

Examples

```
>>> from diofant import Point2D
>>> t = Point2D(0, 1)
>>> t.translate(2)
Point2D(2, 1)
>>> t.translate(2, 2)
Point2D(2, 3)
>>> t + Point2D(2, 2)
Point2D(2, 3)
```

x

Returns the X coordinate of the Point.

Examples

```
>>> from diofant import Point2D
>>> p = Point2D(0, 1)
>>> p.x
0
```

y

Returns the Y coordinate of the Point.

Examples

```
>>> from diofant import Point2D
>>> p = Point2D(0, 1)
>>> p.y
1
```

class diofant.geometry.point.Point3D

A point in a 3-dimensional Euclidean space.

Parameters `coords` : sequence of 3 coordinate values.

Raises TypeError

When trying to add or subtract points with different dimensions.

When *intersection* is called with object other than a Point.

Notes

Currently only 2-dimensional and 3-dimensional points are supported.

Examples

```
>>> from diofant import Point3D
>>> from diofant.abc import x
>>> Point3D(1, 2, 3)
Point3D(1, 2, 3)
```

```
>>> Point3D([1, 2, 3])
Point3D(1, 2, 3)
>>> Point3D(0, x, 3)
Point3D(0, x, 3)
```

Floats are automatically converted to Rational unless the evaluate flag is False:

```
>>> Point3D(0.5, 0.25, 2)
Point3D(1/2, 1/4, 2)
>>> print(Point3D(0.5, 0.25, 3, evaluate=False))
Point3D(0.5, 0.25, 3)
```

Attributes

x (page 448)	Returns the X coordinate of the Point.
y (page 449)	Returns the Y coordinate of the Point.
z (page 449)	Returns the Z coordinate of the Point.

`diofant.geometry.Point.length`

static `are_collinear(*points)`

Is a sequence of points collinear?

Test whether or not a set of points are collinear. Returns True if the set of points are collinear, or False otherwise.

Parameters `points` : sequence of Point

Returns `are_collinear` : boolean

See also:

[diofant.geometry.line3d.Line3D](#) (page 466)

Examples

```
>>> from diofant import Point3D, Matrix
>>> from diofant.abc import x
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> p3, p4, p5 = Point3D(2, 2, 2), Point3D(x, x, x), Point3D(1, 2, 6)
>>> Point3D.are_collinear(p1, p2, p3, p4)
True
>>> Point3D.are_collinear(p1, p2, p3, p5)
False
```

static `are_coplanar(*points)`

This function tests whether passed points are coplanar or not. It uses the fact that the triple scalar product of three vectors vanishes if the vectors are coplanar. Which means that the volume of the solid described by them will have to be zero for coplanarity.

Parameters A set of points 3D `points`

Returns boolean

Examples

```
>>> from diofant import Point3D
>>> p1 = Point3D(1, 2, 2)
>>> p2 = Point3D(2, 7, 2)
>>> p3 = Point3D(0, 0, 2)
>>> p4 = Point3D(1, 1, 2)
>>> Point3D.are_coplanar(p1, p2, p3, p4)
True
>>> p5 = Point3D(0, 1, 3)
>>> Point3D.are_coplanar(p1, p2, p3, p5)
False
```

`direction_cosine(point)`

Gives the direction cosine between 2 points

Parameters `p` : Point3D

Returns list

Examples

```
>>> from diofant import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_cosine(Point3D(2, 3, 5))
[sqrt(6)/6, sqrt(6)/6, sqrt(6)/3]
```

`direction_ratio(point)`

Gives the direction ratio between 2 points

Parameters `p` : Point3D

Returns list

Examples

```
>>> from diofant import Point3D
>>> p1 = Point3D(1, 2, 3)
>>> p1.direction_ratio(Point3D(2, 3, 5))
[1, 1, 2]
```

`intersection(o)`

The intersection between this point and another point.

Parameters `other` : Point

Returns `intersection` : list of Points

Notes

The return value will either be an empty list if there is no intersection, otherwise it will contain this point.

Examples

```
>>> from diofant import Point3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 0, 0)
>>> p1.intersection(p2)
[]
>>> p1.intersection(p3)
[Point3D(0, 0, 0)]
```

scale($x=1, y=1, z=1, pt=None$)

Scale the coordinates of the Point by multiplying by x and y after subtracting pt - default is $(0, 0)$ - and then adding pt back again (i.e. pt is the point of reference for the scaling).

See also:

[translate](#) (page 448)

Examples

```
>>> from diofant import Point3D
>>> t = Point3D(1, 1, 1)
>>> t.scale(2)
Point3D(2, 1, 1)
>>> t.scale(2, 2)
Point3D(2, 2, 1)
```

transform(*matrix*)

Return the point after applying the transformation described by the 4x4 Matrix, *matrix*.

See also:

[diofant.geometry.entity.GeometryEntity.rotate](#) (page 434), [diofant.geometry.entity.GeometryEntity.scale](#) (page 434), [diofant.geometry.entity.GeometryEntity.translate](#) (page 434)

translate($x=0, y=0, z=0$)

Shift the Point by adding x and y to the coordinates of the Point.

See also:

[diofant.geometry.entity.GeometryEntity.rotate](#) (page 434), [diofant.geometry.entity.GeometryEntity.scale](#) (page 434)

Examples

```
>>> from diofant import Point3D
>>> t = Point3D(0, 1, 1)
>>> t.translate(2)
Point3D(2, 1, 1)
>>> t.translate(2, 2)
Point3D(2, 3, 1)
>>> t + Point3D(2, 2, 2)
Point3D(2, 3, 3)
```

x
Returns the X coordinate of the Point.

Examples

```
>>> from diofant import Point3D
>>> p = Point3D(0, 1, 3)
>>> p.x
0
```

y
Returns the Y coordinate of the Point.

Examples

```
>>> from diofant import Point3D
>>> p = Point3D(0, 1, 2)
>>> p.y
1
```

z
Returns the Z coordinate of the Point.

Examples

```
>>> from diofant import Point3D
>>> p = Point3D(0, 1, 1)
>>> p.z
1
```

Lines

class `diofant.geometry.line.LinearEntity`

A base class for all linear entities (line, ray and segment) in a 2-dimensional Euclidean space.

See also:

[*diofant.geometry.entity.GeometryEntity*](#) (page 433)

Notes

This is an abstract class and is not meant to be instantiated.

Attributes

p1 (page 454)

The first defining point of a linear entity.

Continued on next page

Table 3.4 – continued from previous page

p2 (page 454)	The second defining point of a linear entity.
coefficients (page 451)	The coefficients (a, b, c) for $ax + by + c = 0$.
slope (page 457)	The slope of this linear entity, or infinity if vertical.
points (page 456)	The two points used to define this linear entity.

angle_between(*other*)

The angle formed between the two linear entities.

Parameters **self** : LinearEntity

other : LinearEntity

Returns **angle** : angle in radians

See also:

[is_perpendicular](#) (page 453)

Notes

From the dot product of vectors $v1$ and $v2$ it is known that:

$$\text{dot}(v1, v2) = |v1| * |v2| * \cos(A)$$

where A is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(0, 4), Point(2, 0)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.angle_between(l2)
pi/2
```

arbitrary_point(*parameter='t'*)

A parameterized point on the Line.

Parameters **parameter** : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

Returns **point** : Point

Raises ValueError

When parameter already appears in the Line's definition.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.arbitrary_point()
Point2D(4*t + 1, 3*t)
```

static are_concurrent(*lines)

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

Parameters lines : a sequence of linear entities.

Returns True : if the set of linear entities are concurrent,

False : otherwise.

See also:

[diofant.geometry.util.intersection](#) (page 435)

Notes

Simply take the first two lines and find their intersection. If there is no intersection, then the first two lines were parallel and had no intersection so concurrency is impossible amongst the whole set. Otherwise, check to see if the intersection point of the first two lines is a member on the rest of the lines. If so, the lines are concurrent.

Examples

```
>>> from diofant import Point, Line, Line3D
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> p3, p4 = Point(-2, -2), Point(0, 2)
>>> l1, l2, l3 = Line(p1, p2), Line(p1, p3), Line(p1, p4)
>>> Line.are_concurrent(l1, l2, l3)
True
```

```
>>> l4 = Line(p2, p3)
>>> Line.are_concurrent(l2, l3, l4)
False
```

coefficients

The coefficients (a, b, c) for $ax + by + c = 0$.

See also:

[diofant.geometry.line.Line.equation](#) (page 459)

Examples

```
>>> from diofant import Point, Line
>>> from diofant.abc import x, y
>>> p1, p2 = Point(0, 0), Point(5, 3)
```

```
>>> l = Line(p1, p2)
>>> l.coefficients
(-3, 5, 0)
```

```
>>> p3 = Point(x, y)
>>> l2 = Line(p1, p3)
>>> l2.coefficients
(-y, x, 0)
```

contains(*other*)

Subclasses should implement this method and should return True if *other* is on the boundaries of *self*; False if not on the boundaries of *self*; None if a determination cannot be made.

intersection(*o*)

The intersection with another geometrical entity.

Parameters *o* : Point or LinearEntity

Returns **intersection** : list of geometrical entities

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Line, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(7, 7)
>>> l1 = Line(p1, p2)
>>> l1.intersection(p3)
[Point2D(7, 7)]
```

```
>>> p4, p5 = Point(5, 0), Point(0, 3)
>>> l2 = Line(p4, p5)
>>> l1.intersection(l2)
[Point2D(15/8, 15/8)]
```

```
>>> p6, p7 = Point(0, 5), Point(2, 6)
>>> s1 = Segment(p6, p7)
>>> l1.intersection(s1)
[]
```

is_parallel(*other*)

Are two linear entities parallel?

Parameters **self** : LinearEntity

other : LinearEntity

Returns **True** : if *self* and *other* are parallel,

False : otherwise.

See also:

[coefficients](#) (page 451)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> p3, p4 = Point(3, 4), Point(6, 7)
>>> l1, l2 = Line(p1, p2), Line(p3, p4)
>>> Line.is_parallel(l1, l2)
True
```

```
>>> p5 = Point(6, 6)
>>> l3 = Line(p3, p5)
>>> Line.is_parallel(l1, l3)
False
```

`is_perpendicular(other)`

Are two linear entities perpendicular?

Parameters `self` : LinearEntity

other : LinearEntity

Returns True : if self and other are perpendicular,

False : otherwise.

See also:

[*coefficients*](#) (page 451)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(-1, 1)
>>> l1, l2 = Line(p1, p2), Line(p1, p3)
>>> l1.is_perpendicular(l2)
True
```

```
>>> p4 = Point(5, 3)
>>> l3 = Line(p1, p4)
>>> l1.is_perpendicular(l3)
False
```

`is_similar(other)`

Return True if self and other are contained in the same line.

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 1), Point(3, 4), Point(2, 3)
>>> l1 = Line(p1, p2)
>>> l2 = Line(p1, p3)
>>> l1.is_similar(l2)
True
```

`length`

The length of the line.

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.length
oo
```

p1

The first defining point of a linear entity.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p1
Point2D(0, 0)
```

p2

The second defining point of a linear entity.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l = Line(p1, p2)
>>> l.p2
Point2D(5, 3)
```

parallel_line(*p*)

Create a new Line parallel to this linear entity which passes through the point *p*.

Parameters *p* : Point

Returns *line* : Line

See also:

[is_parallel](#) (page 452)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.parallel_line(p3)
```

```
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

perpendicular_line(*p*)

Create a new Line perpendicular to this linear entity which passes through the point *p*.

Parameters *p* : Point

Returns *line* : Line

See also:

[is_perpendicular](#) (page 453), [perpendicular_segment](#) (page 455)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(2, 3), Point(-2, 2)
>>> l1 = Line(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

perpendicular_segment(*p*)

Create a perpendicular line segment from *p* to this line.

The endpoints of the segment are *p* and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

Parameters *p* : Point

Returns *segment* : Segment

See also:

[perpendicular_line](#) (page 455)

Notes

Returns *p* itself if *p* is on this linear entity.

Examples

```
>>> from diofant import Point, Line
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, 2)
>>> l1 = Line(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
```

```
>>> l1.perpendicular_segment(Point(4, 0))
Segment(Point2D(2, 2), Point2D(4, 0))
```

points

The two points used to define this linear entity.

Returns points : tuple of Points

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 11)
>>> l1 = Line(p1, p2)
>>> l1.points
(Point2D(0, 0), Point2D(5, 11))
```

projection(*o*)

Project a point, line, ray, or segment onto this linear entity.

Parameters other : Point or LinearEntity (Line, Ray, Segment)

Returns projection : Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter other.

Raises GeometryError

When method is unable to perform projection.

See also:

[diofant.geometry.point.Point](#) (page 438), [perpendicular_line](#) (page 455)

Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point P is projected onto a line L by finding the point on L that is closest to P. This point is the intersection of L and the line perpendicular to L that passes through P.

Examples

```
>>> from diofant import Point, Line, Segment, Rational
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(Rational(1, 2), 0)
>>> l1 = Line(p1, p2)
>>> l1.projection(p3)
Point2D(1/4, 1/4)
```

```
>>> p4, p5 = Point(10, 0), Point(12, 1)
>>> s1 = Segment(p4, p5)
```

```
>>> l1.projection(s1)
Segment(Point2D(5, 5), Point2D(13/2, 13/2))
```

random_point()

A random point on a LinearEntity.

Returns point : Point

See also:

diofant.geometry.point.Point (page 438)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> p3 = l1.random_point()
>>> # random point - don't know its coords in advance
>>> p3
Point2D(...)
>>> # point should belong to the line
>>> p3 in l1
True
```

slope

The slope of this linear entity, or infinity if vertical.

Returns slope : number or diofant expression

See also:

coefficients (page 451)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(3, 5)
>>> l1 = Line(p1, p2)
>>> l1.slope
5/3
```

```
>>> p3 = Point(0, 4)
>>> l2 = Line(p1, p3)
>>> l2.slope
oo
```

class diofant.geometry.line.Line

An infinite line in space.

A line is declared with two distinct points or a point and slope as defined using keyword *slope*.

Parameters p1 : Point

pt : Point

slope : diofant expression

See also:

[diofant.geometry.point.Point](#) (page 438)

Notes

At the moment only lines in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import diofant
>>> from diofant import Point
>>> from diofant.abc import L
>>> from diofant.geometry import Line, Segment
>>> L = Line(Point(2,3), Point(3,5))
>>> L
Line(Point2D(2, 3), Point2D(3, 5))
>>> L.points
(Point2D(2, 3), Point2D(3, 5))
>>> L.equation()
-2*x + y + 1
>>> L.coefficients
(-2, 1, 1)
```

Instantiate with keyword slope:

```
>>> Line(Point(0, 0), slope=0)
Line(Point2D(0, 0), Point2D(1, 0))
```

Instantiate with another linear object

```
>>> s = Segment((0, 0), (0, 1))
>>> Line(s).equation()
x
```

contains(*o*)

Return True if *o* is on this Line, or False otherwise.

Examples

```
>>> from diofant import Line, Point
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> l = Line(p1, p2)
>>> l.contains(p1)
True
>>> l.contains((0, 1))
True
>>> l.contains((0, 0))
False
```

distance(*o*)

Finds the shortest distance between a line and a point.

Raises NotImplementedError is raised if o is not a Point**Examples**

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Line(p1, p2)
>>> s.distance(Point(-1, 1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
```

equal(*other*)

Returns True if self and other are the same mathematical entities

equation(*x='x', y='y'*)

The equation of the line: $ax + by + c$.

Parameters *x* : str, optional

The name to use for the x-axis, default value is 'x'.

y : str, optional

The name to use for the y-axis, default value is 'y'.

Returns **equation** : diofant expression

See also:

[LinearEntity.coefficients](#) (page 451)

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(1, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.equation()
-3*x + 4*y + 3
```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of line. Gives values that will produce a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

Parameters *parameter* : str, optional

Default value is 't'.

Returns **plot_interval** : list (plot interval)

[*parameter*, *lower_bound*, *upper_bound*]

Examples

```
>>> from diofant import Point, Line
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> l1 = Line(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

class `diofant.geometry.line.Ray`

A Ray is a semi-line in the space with a source point and a direction.

Parameters `p1` : Point

The source of the Ray

`p2` : Point or radian value

This point determines the direction in which the Ray propagates. If given as an angle it is interpreted in radians with the positive direction being ccw.

See also:

[diofant.geometry.point.Point](#) (page 438), [Line](#) (page 457)

Notes

At the moment only rays in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import diofant
>>> from diofant import Point, pi
>>> from diofant.abc import r
>>> from diofant.geometry import Ray
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r = Ray(Point(2, 3), Point(3, 5))
>>> r
Ray(Point2D(2, 3), Point2D(3, 5))
>>> r.points
(Point2D(2, 3), Point2D(3, 5))
>>> r.source
Point2D(2, 3)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.slope
2
>>> Ray(Point(0, 0), angle=pi/4).slope
1
```

Attributes

`source` (page 462)

The point from which the ray emanates.

Continued on next page

Table 3.5 – continued from previous page

<i>xdirection</i> (page 462)	The x direction of the ray.
<i>ydirection</i> (page 463)	The y direction of the ray.

contains(*o*)

Is other GeometryEntity contained in this Ray?

Examples

```
>>> from diofant import Ray, Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 4)
>>> r = Ray(p1, p2)
>>> r.contains(p1)
True
>>> r.contains((1, 1))
True
>>> r.contains((1, 3))
False
>>> s = Segment((1, 1), (2, 2))
>>> r.contains(s)
True
>>> s = Segment((1, 2), (2, 5))
>>> r.contains(s)
False
>>> r1 = Ray((2, 2), (3, 3))
>>> r.contains(r1)
True
>>> r1 = Ray((2, 2), (3, 5))
>>> r.contains(r1)
False
```

direction

The direction in which the ray emanates.

See also:

[*diofant.geometry.point.Point*](#) (page 438)

Examples

```
>>> from diofant import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.direction
Point2D(4, 1)
```

distance(*o*)

Finds the shortest distance between the ray and a point.

Raises `NotImplementedError` is raised if *o* is not a `Point`

Examples

```
>>> from diofant import Point, Ray
>>> p1, p2 = Point(0, 0), Point(1, 1)
>>> s = Ray(p1, p2)
>>> s.distance(Point(-1, -1))
sqrt(2)
>>> s.distance((-1, 2))
3*sqrt(2)/2
```

equals(*other*)

Returns True if self and other are the same mathematical entities

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from diofant import Point, Ray, pi
>>> r = Ray((0, 0), angle=pi/4)
>>> r.plot_interval()
[t, 0, 10]
```

source

The point from which the ray emanates.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Ray
>>> p1, p2 = Point(0, 0), Point(4, 1)
>>> r1 = Ray(p1, p2)
>>> r1.source
Point2D(0, 0)
```

xdirection

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

See also:

[ydirection](#) (page 463)

Examples

```
>>> from diofant import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(1, 1), Point(0, -1)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

ydirection

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 462)

Examples

```
>>> from diofant import Point, Ray
>>> p1, p2, p3 = Point(0, 0), Point(-1, -1), Point(-1, 0)
>>> r1, r2 = Ray(p1, p2), Ray(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

class diofant.geometry.line.Segment

A undirected line segment in space.

Parameters **p1** : Point

p2 : Point

See also:

[diofant.geometry.point.Point](#) (page 438), [Line](#) (page 457)

Notes

At the moment only segments in a 2D space can be declared, because Points can be defined only for 2D spaces.

Examples

```
>>> import diofant
>>> from diofant import Point
>>> from diofant.abc import s
>>> from diofant.geometry import Segment
>>> Segment((1, 0), (1, 1)) # tuples are interpreted as pts
Segment(Point2D(1, 0), Point2D(1, 1))
>>> s = Segment(Point(4, 3), Point(1, 1))
```

```
>>> s
Segment(Point2D(1, 1), Point2D(4, 3))
>>> s.points
(Point2D(1, 1), Point2D(4, 3))
>>> s.slope
2/3
>>> s.length
sqrt(13)
>>> s.midpoint
Point2D(5/2, 2)
```

Attributes

<i>length</i> (page 464)	The length of the line segment.
<i>midpoint</i> (page 465)	The midpoint of the line segment.

contains(*other*)

Is the other GeometryEntity contained within this Segment?

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s2 = Segment(p2, p1)
>>> s.contains(s2)
True
```

distance(*o*)

Finds the shortest distance between a line segment and a point.

Raises NotImplementedError is raised if *o* is not a Point

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2 = Point(0, 1), Point(3, 4)
>>> s = Segment(p1, p2)
>>> s.distance(Point(10, 15))
sqrt(170)
>>> s.distance((0, 12))
sqrt(73)
```

length

The length of the line segment.

See also:

diofant.geometry.point.Point.distance (page 439)

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.length
5
```

midpoint

The midpoint of the line segment.

See also:

[diofant.geometry.point.Point.midpoint](#) (page 441)

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2 = Point(0, 0), Point(4, 3)
>>> s1 = Segment(p1, p2)
>>> s1.midpoint
Point2D(2, 3/2)
```

perpendicular_bisector(*p=None*)

The perpendicular bisector of this segment.

If no point is specified or the point specified is not on the bisector then the bisector is returned as a Line. Otherwise a Segment is returned that joins the point specified and the intersection of the bisector and the segment.

Parameters *p* : Point

Returns *bisector* : Line or Segment

See also:

[LinearEntity.perpendicular_segment](#) (page 455)

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2, p3 = Point(0, 0), Point(6, 6), Point(5, 1)
>>> s1 = Segment(p1, p2)
>>> s1.perpendicular_bisector()
Line(Point2D(3, 3), Point2D(9, -3))
```

```
>>> s1.perpendicular_bisector(p3)
Segment(Point2D(3, 3), Point2D(5, 1))
```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

Parameters *parameter* : str, optional

Default value is 't'.

Returns *plot_interval* : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from diofant import Point, Segment
>>> p1, p2 = Point(0, 0), Point(5, 3)
>>> s1 = Segment(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

3D Line

Line-like geometrical entities.

Contains

LinearEntity3D Line3D Ray3D Segment3D

class diofant.geometry.line3d.Line3D

An infinite 3D line in space.

A line is declared with two distinct points or a point and `direction_ratio` as defined using keyword *direction_ratio*.

Parameters `p1` : Point3D

`pt` : Point3D

`direction_ratio` : list

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> import diofant
>>> from diofant import Point3D
>>> from diofant.abc import L
>>> from diofant.geometry import Line3D, Segment3D
>>> L = Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L
Line3D(Point3D(2, 3, 4), Point3D(3, 5, 1))
>>> L.points
(Point3D(2, 3, 4), Point3D(3, 5, 1))
```

contains(o)

Return True if o is on this Line, or False otherwise.

Examples

```

>>> from diofant import Line3D
>>> a = (0, 0, 0)
>>> b = (1, 1, 1)
>>> c = (2, 2, 2)
>>> l1 = Line3D(a, b)
>>> l2 = Line3D(b, a)
>>> l1 == l2
False
>>> l1 in l2
True

```

distance(*o*)

Finds the shortest distance between a line and a point.

Raises `NotImplementedError` is raised if *o* is not an instance of `Point3D`

Examples

```

>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 1)
>>> s = Line3D(p1, p2)
>>> s.distance(Point3D(-1, 1, 1))
2*sqrt(6)/3
>>> s.distance((-1, 1, 1))
2*sqrt(6)/3

```

equals(*other*)

Returns True if self and other are the same mathematical entities

equation(*x='x'*, *y='y'*, *z='z'*, *k='k'*)

The equation of the line in 3D

Parameters *x* : str, optional

The name to use for the x-axis, default value is 'x'.

y : str, optional

The name to use for the y-axis, default value is 'y'.

z : str, optional

The name to use for the x-axis, default value is 'z'.

Returns `equation` : tuple

Examples

```

>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 0)
>>> l1 = Line3D(p1, p2)
>>> l1.equation()
(x/4 - 1/4, y/3, zoo*z, k)

```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of line. Gives values that will produce

a line that is +/- 5 units long (where a unit is the distance between the two points that define the line).

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list (plot interval)

[parameter, lower_bound, upper_bound]

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.plot_interval()
[t, -5, 5]
```

class diofant.geometry.line3d.LinearEntity3D

An base class for all linear entities (line, ray and segment) in a 3-dimensional Euclidean space.

Notes

This is a base class and is not meant to be instantiated.

Attributes

p1 (page 473)	The first defining point of a linear entity.
p2 (page 473)	The second defining point of a linear entity.
direction_ratio (page 470)	The direction ratio of a given line in 3D.
direction_cosine (page 470)	The normalized direction ratio of a given line in 3D.
points (page 474)	The two points used to define this linear entity.

angle_between(*other*)

The angle formed between the two linear entities.

Parameters **self** : LinearEntity

other : LinearEntity

Returns **angle** : angle in radians

See also:

[is_perpendicular](#) (page 472)

Notes

From the dot product of vectors v1 and v2 it is known that:

$$\text{dot}(v1, v2) = |v1| * |v2| * \cos(A)$$

where A is the angle formed between the two vectors. We can get the directional vectors of the two lines and readily find the angle between the two using the above formula.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.angle_between(l2)
acos(-sqrt(2)/3)
```

arbitrary_point(*parameter='t'*)

A parameterized point on the Line.

Parameters *parameter* : str, optional

The name of the parameter which will be used for the parametric point. The default value is 't'. When this parameter is 0, the first point used to define the line will be returned, and when it is 1 the second point will be returned.

Returns *point* : Point3D

Raises **ValueError**

When parameter already appears in the Line's definition.

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(1, 0, 0), Point3D(5, 3, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.arbitrary_point()
Point3D(4*t + 1, 3*t, t)
```

static **are_concurrent**(**lines*)

Is a sequence of linear entities concurrent?

Two or more linear entities are concurrent if they all intersect at a single point.

Parameters *lines* : a sequence of linear entities.

Returns **True** : if the set of linear entities are concurrent,

False : otherwise.

See also:

[diofant.geometry.util.intersection](#) (page 435)

Notes

Simply take the first two lines and find their intersection. If there is no intersection, then the first two lines were parallel and had no intersection so concurrency is impossible amongst the whole set. Otherwise, check to see if the intersection point of the first two lines is a member on the rest of the lines. If so, the lines are concurrent.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 2)
>>> p3, p4 = Point3D(-2, -2, -2), Point3D(0, 2, 1)
>>> l1, l2, l3 = Line3D(p1, p2), Line3D(p1, p3), Line3D(p1, p4)
>>> Line3D.are_concurrent(l1, l2, l3)
True
```

```
>>> l4 = Line3D(p2, p3)
>>> Line3D.are_concurrent(l2, l3, l4)
False
```

contains(*other*)

Subclasses should implement this method and should return True if other is on the boundaries of self; False if not on the boundaries of self; None if a determination cannot be made.

direction_cosine

The normalized direction ratio of a given line in 3D.

See also:

[diofant.geometry.line.Line.equation](#) (page 459)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_cosine
[sqrt(35)/7, 3*sqrt(35)/35, sqrt(35)/35]
>>> sum(i**2 for i in _)
1
```

direction_ratio

The direction ratio of a given line in 3D.

See also:

[diofant.geometry.line.Line.equation](#) (page 459)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.direction_ratio
[5, 3, 1]
```

intersection(*o*)

The intersection with another geometrical entity.

Parameters *o* : Point or LinearEntity3D

Returns **intersection** : list of geometrical entities

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Line3D, Segment3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(7, 7, 7)
>>> l1 = Line3D(p1, p2)
>>> l1.intersection(p3)
[Point3D(7, 7, 7)]
```

```
>>> l1 = Line3D(Point3D(4,19,12), Point3D(5,25,17))
>>> l2 = Line3D(Point3D(-3, -15, -19), direction_ratio=[2,8,8])
>>> l1.intersection(l2)
[Point3D(1, 1, -3)]
```

```
>>> p6, p7 = Point3D(0, 5, 2), Point3D(2, 6, 3)
>>> s1 = Segment3D(p6, p7)
>>> l1.intersection(s1)
[]
```

is_parallel(*other*)

Are two linear entities parallel?

Parameters **self** : LinearEntity

other : LinearEntity

Returns **True** : if self and other are parallel,

False : otherwise.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 4, 5)
>>> p3, p4 = Point3D(2, 1, 1), Point3D(8, 9, 11)
>>> l1, l2 = Line3D(p1, p2), Line3D(p3, p4)
>>> Line3D.is_parallel(l1, l2)
True
```

```
>>> p5 = Point3D(6, 6, 6)
>>> l3 = Line3D(p3, p5)
>>> Line3D.is_parallel(l1, l3)
False
```

is_perpendicular(*other*)

Are two linear entities perpendicular?

Parameters **self** : LinearEntity

other : LinearEntity

Returns **True** : if self and other are perpendicular,

False : otherwise.

See also:

[direction_ratio](#) (page 470)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(-1, 2, 0)
>>> l1, l2 = Line3D(p1, p2), Line3D(p2, p3)
>>> l1.is_perpendicular(l2)
False
```

```
>>> p4 = Point3D(5, 3, 7)
>>> l3 = Line3D(p1, p4)
>>> l1.is_perpendicular(l3)
False
```

is_similar(*other*)

Return True if self and other are contained in the same line.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(2, 2, 2)
>>> l1 = Line3D(p1, p2)
>>> l2 = Line3D(p1, p3)
>>> l1.is_similar(l2)
True
```

length

The length of the line.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(3, 5, 1)
>>> l1 = Line3D(p1, p2)
```

```
>>> l1.length
00
```

p1

The first defining point of a linear entity.

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.p1
Point3D(0, 0, 0)
```

p2

The second defining point of a linear entity.

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 1)
>>> l = Line3D(p1, p2)
>>> l.p2
Point3D(5, 3, 1)
```

parallel_line(p)

Create a new Line parallel to this linear entity which passes through the point p .

Parameters p : Point3D

Returns $line$: Line3D

See also:

[is_parallel](#) (page 471)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.parallel_line(p3)
>>> p3 in l2
True
>>> l1.is_parallel(l2)
True
```

perpendicular_line(*p*)

Create a new Line perpendicular to this linear entity which passes through the point *p*.

Parameters *p* : Point3D

Returns *line* : Line3D

See also:

[is_perpendicular](#) (page 472), [perpendicular_segment](#) (page 474)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(2, 3, 4), Point3D(-2, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> l2 = l1.perpendicular_line(p3)
>>> p3 in l2
True
>>> l1.is_perpendicular(l2)
True
```

perpendicular_segment(*p*)

Create a perpendicular line segment from *p* to this line.

The endpoints of the segment are *p* and the closest point in the line containing self. (If self is not a line, the point might not be in self.)

Parameters *p* : Point3D

Returns *segment* : Segment3D

See also:

[perpendicular_line](#) (page 473)

Notes

Returns *p* itself if *p* is on this linear entity.

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, 2, 0)
>>> l1 = Line3D(p1, p2)
>>> s1 = l1.perpendicular_segment(p3)
>>> l1.is_perpendicular(s1)
True
>>> p3 in s1
True
>>> l1.perpendicular_segment(Point3D(4, 0, 0))
Segment3D(Point3D(4/3, 4/3, 4/3), Point3D(4, 0, 0))
```

points

The two points used to define this linear entity.

Returns points : tuple of Points

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Line3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 11, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.points
(Point3D(0, 0, 0), Point3D(5, 11, 1))
```

`projection(o)`

Project a point, line, ray, or segment onto this linear entity.

Parameters other : Point or LinearEntity (Line, Ray, Segment)

Returns projection : Point or LinearEntity (Line, Ray, Segment)

The return type matches the type of the parameter other.

Raises GeometryError

When method is unable to perform projection.

See also:

[diofant.geometry.point.Point3D](#) (page 445), [perpendicular_line](#) (page 473)

Notes

A projection involves taking the two points that define the linear entity and projecting those points onto a Line and then reforming the linear entity using these projections. A point P is projected onto a line L by finding the point on L that is closest to P. This point is the intersection of L and the line perpendicular to L that passes through P.

Examples

```
>>> from diofant import Point3D, Line3D, Segment3D, Rational
>>> p1, p2, p3 = Point3D(0, 0, 1), Point3D(1, 1, 2), Point3D(2, 0, 1)
>>> l1 = Line3D(p1, p2)
>>> l1.projection(p3)
Point3D(2/3, 2/3, 5/3)
```

```
>>> p4, p5 = Point3D(10, 0, 1), Point3D(12, 1, 3)
>>> s1 = Segment3D(p4, p5)
>>> l1.projection(s1)
[Segment3D(Point3D(10/3, 10/3, 13/3), Point3D(5, 5, 6))]
```

`class diofant.geometry.line3d.Ray3D`

A Ray is a semi-line in the space with a source point and a direction.

Parameters p1 : Point3D

The source of the Ray

p2 : Point or a direction vector

direction_ratio: Determines the direction in which the Ray propagates.

See also:

[diofant.geometry.point.Point3D](#) (page 445), [Line3D](#) (page 466)

Examples

```
>>> import diofant
>>> from diofant import Point3D, pi
>>> from diofant.abc import r
>>> from diofant.geometry import Ray3D
>>> r = Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r
Ray3D(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.points
(Point3D(2, 3, 4), Point3D(3, 5, 0))
>>> r.source
Point3D(2, 3, 4)
>>> r.xdirection
00
>>> r.ydirection
00
>>> r.direction_ratio
[1, 2, -4]
```

Attributes

source (page 477)	The point from which the ray emanates.
xdirection (page 477)	The x direction of the ray.
ydirection (page 478)	The y direction of the ray.
zdirection (page 478)	The z direction of the ray.

contains(*o*)

Is other GeometryEntity contained in this Ray?

distance(*o*)

Finds the shortest distance between the ray and a point.

Raises NotImplementedError is raised if o is not a Point

Examples

```
>>> from diofant import Point3D, Ray3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(1, 1, 2)
>>> s = Ray3D(p1, p2)
>>> s.distance(Point3D(-1, -1, 2))
sqrt(6)
```



```
>>> s.distance((-1, -1, 2))
sqrt(6)
```

equals(*other*)

Returns True if self and other are the same mathematical entities

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the Ray. Gives values that will produce a ray that is 10 units long (where a unit is the distance between the two points that define the ray).

Parameters **parameter** : str, optional

Default value is 't'.

Returns **plot_interval** : list

[parameter, lower_bound, upper_bound]

Examples

```
>>> from diofant import Point3D, Ray3D, pi
>>> r = Ray3D(Point3D(0, 0, 0), Point3D(1, 1, 1))
>>> r.plot_interval()
[t, 0, 10]
```

source

The point from which the ray emanates.

See also:

[diofant.geometry.point.Point3D](#) (page 445)

Examples

```
>>> from diofant import Point3D, Ray3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 1, 5)
>>> r1 = Ray3D(p1, p2)
>>> r1.source
Point3D(0, 0, 0)
```

xdirection

The x direction of the ray.

Positive infinity if the ray points in the positive x direction, negative infinity if the ray points in the negative x direction, or 0 if the ray is vertical.

See also:

[ydirection](#) (page 478)

Examples

```
>>> from diofant import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(1, 1, 1), Point3D(0, -1, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
```

```
>>> r1.xdirection
oo
>>> r2.xdirection
0
```

ydirection

The y direction of the ray.

Positive infinity if the ray points in the positive y direction, negative infinity if the ray points in the negative y direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 477)

Examples

```
>>> from diofant import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
```

zdirection

The z direction of the ray.

Positive infinity if the ray points in the positive z direction, negative infinity if the ray points in the negative z direction, or 0 if the ray is horizontal.

See also:

[xdirection](#) (page 477)

Examples

```
>>> from diofant import Point3D, Ray3D
>>> p1, p2, p3 = Point3D(0, 0, 0), Point3D(-1, -1, -1), Point3D(-1, 0, 0)
>>> r1, r2 = Ray3D(p1, p2), Ray3D(p1, p3)
>>> r1.ydirection
-oo
>>> r2.ydirection
0
>>> r2.zdirection
0
```

class `diofant.geometry.line3d.Segment3D`

A undirected line segment in a 3D space.

Parameters `p1` : Point3D

`p2` : Point3D

See also:

[diofant.geometry.point.Point3D](#) (page 445), [Line3D](#) (page 466)

Examples

```
>>> import diofant
>>> from diofant import Point3D
>>> from diofant.abc import s
>>> from diofant.geometry import Segment3D
>>> Segment3D((1, 0, 0), (1, 1, 1)) # tuples are interpreted as pts
Segment3D(Point3D(1, 0, 0), Point3D(1, 1, 1))
>>> s = Segment3D(Point3D(4, 3, 9), Point3D(1, 1, 7))
>>> s
Segment3D(Point3D(1, 1, 7), Point3D(4, 3, 9))
>>> s.points
(Point3D(1, 1, 7), Point3D(4, 3, 9))
>>> s.length
sqrt(17)
>>> s.midpoint
Point3D(5/2, 2, 8)
```

Attributes

<i>length</i> (page 479)	The length of the line segment.
<i>midpoint</i> (page 480)	The midpoint of the line segment.

contains(*other*)

Is the other GeometryEntity contained within this Segment?

Examples

```
>>> from diofant import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 1, 1), Point3D(3, 4, 5)
>>> s = Segment3D(p1, p2)
>>> s2 = Segment3D(p2, p1)
>>> s.contains(s2)
True
```

distance(*o*)

Finds the shortest distance between a line segment and a point.

Raises NotImplementedError is raised if *o* is not a Point3D

Examples

```
>>> from diofant import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 3), Point3D(1, 1, 4)
>>> s = Segment3D(p1, p2)
>>> s.distance(Point3D(10, 15, 12))
sqrt(341)
>>> s.distance((10, 15, 12))
sqrt(341)
```

length

The length of the line segment.

See also:

[diofant.geometry.point.Point.distance](#) (page 439)

Examples

```
>>> from diofant import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.length
sqrt(34)
```

midpoint

The midpoint of the line segment.

See also:

[diofant.geometry.point.Point.midpoint](#) (page 441)

Examples

```
>>> from diofant import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(4, 3, 3)
>>> s1 = Segment3D(p1, p2)
>>> s1.midpoint
Point3D(2, 3/2, 3/2)
```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the Segment gives values that will produce the full segment in a plot.

Parameters *parameter* : str, optional

Default value is 't'.

Returns *plot_interval* : list

[*parameter*, *lower_bound*, *upper_bound*]

Examples

```
>>> from diofant import Point3D, Segment3D
>>> p1, p2 = Point3D(0, 0, 0), Point3D(5, 3, 0)
>>> s1 = Segment3D(p1, p2)
>>> s1.plot_interval()
[t, 0, 1]
```

Curves

class `diofant.geometry.curve.Curve`

A curve in space.

A curve is defined by parametric functions for the coordinates, a parameter and the lower and upper bounds for the parameter value.

Parameters function : list of functions

limits : 3-tuple

Function parameter and lower and upper bounds.

Raises ValueError

When *functions* are specified incorrectly. When *limits* are specified incorrectly.

See also:

[diofant.core.function.Function](#) (page 134), [diofant.polys.polyfuncs.interpolate](#) (page 730)

Examples

```
>>> from diofant import sin, cos, Symbol, interpolate
>>> from diofant.abc import t, a
>>> from diofant.geometry import Curve
>>> C = Curve((sin(t), cos(t)), (t, 0, 2))
>>> C.functions
(sin(t), cos(t))
>>> C.limits
(t, 0, 2)
>>> C.parameter
t
>>> C = Curve((t, interpolate([1, 4, 9, 16], t)), (t, 0, 1)); C
Curve((t, t**2), (t, 0, 1))
>>> C.subs(t, 4)
Point2D(4, 16)
>>> C.arbitrary_point(a)
Point2D(a, a**2)
```

Attributes

<i>functions</i> (page 482)	The functions specifying the curve.
<i>parameter</i> (page 483)	The curve function variable.
<i>limits</i> (page 482)	The limits for the curve.

arbitrary_point(*parameter*='t')

A parameterized point on the curve.

Parameters parameter : str or Symbol, optional

Default value is 't'; the Curve's parameter is selected with None or self.parameter otherwise the provided symbol is used.

Returns arbitrary_point : Point

Raises ValueError

When *parameter* already appears in the functions.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Symbol
>>> from diofant.abc import s
>>> from diofant.geometry import Curve
>>> C = Curve([2*s, s**2], (s, 0, 2))
>>> C.arbitrary_point()
Point2D(2*t, t**2)
>>> C.arbitrary_point(C.parameter)
Point2D(2*s, s**2)
>>> C.arbitrary_point(None)
Point2D(2*s, s**2)
>>> C.arbitrary_point(Symbol('a'))
Point2D(2*a, a**2)
```

free_symbols

Return a set of symbols other than the bound symbols used to parametrically define the Curve.

Examples

```
>>> from diofant.abc import t, a
>>> from diofant.geometry import Curve
>>> Curve((t, t**2), (t, 0, 2)).free_symbols == set()
True
>>> Curve((t, t**2), (t, a, 2)).free_symbols == {a}
True
```

functions

The functions specifying the curve.

Returns functions : list of parameterized coordinate functions.

See also:

[parameter](#) (page 483)

Examples

```
>>> from diofant.abc import t
>>> from diofant.geometry import Curve
>>> C = Curve((t, t**2), (t, 0, 2))
>>> C.functions
(t, t**2)
```

limits

The limits for the curve.

Returns limits : tuple

Contains parameter and lower and upper limits.

See also:[plot_interval](#) (page 483)**Examples**

```
>>> from diofant.abc import t
>>> from diofant.geometry import Curve
>>> C = Curve([t, t**3], (t, -2, 2))
>>> C.limits
(t, -2, 2)
```

parameter

The curve function variable.

Returns parameter : Diofant symbol**See also:**[functions](#) (page 482)**Examples**

```
>>> from diofant.abc import t
>>> from diofant.geometry import Curve
>>> C = Curve([t, t**2], (t, 0, 2))
>>> C.parameter
t
```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the curve.

Parameters parameter : str or Symbol, optional

Default value is 't'; otherwise the provided symbol is used.

Returns plot_interval : list (plot interval)

[parameter, lower_bound, upper_bound]

See also:[limits](#) (page 482) Returns limits of the parameter interval**Examples**

```
>>> from diofant import Curve, sin
>>> from diofant.abc import x, t, s
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval()
[t, 1, 2]
>>> Curve((x, sin(x)), (x, 1, 2)).plot_interval(s)
[s, 1, 2]
```

rotate(*angle=0, pt=None*)

Rotate angle radians counterclockwise about Point pt.

The default pt is the origin, Point(0, 0).

Examples

```
>>> from diofant.geometry.curve import Curve
>>> from diofant.abc import x
>>> from diofant import pi
>>> Curve((x, x), (x, 0, 1)).rotate(pi/2)
Curve((-x, x), (x, 0, 1))
```

scale($x=1, y=1, pt=None$)

Override GeometryEntity.scale since Curve is not made up of Points.

Examples

```
>>> from diofant.geometry.curve import Curve
>>> from diofant import pi
>>> from diofant.abc import x
>>> Curve((x, x), (x, 0, 1)).scale(2)
Curve((2*x, x), (x, 0, 1))
```

translate($x=0, y=0$)

Translate the Curve by (x, y).

Examples

```
>>> from diofant.geometry.curve import Curve
>>> from diofant import pi
>>> from diofant.abc import x
>>> Curve((x, x), (x, 0, 1)).translate(1, 2)
Curve((x + 1, x + 2), (x, 0, 1))
```

Ellipses

class diofant.geometry.ellipse.Ellipse

An elliptical GeometryEntity.

Parameters **center** : Point, optional

Default value is Point(0, 0)

hradius : number or Diofant expression, optional

vradius : number or Diofant expression, optional

eccentricity : number or Diofant expression, optional

Two of *hradius*, *vradius* and *eccentricity* must be supplied to create an Ellipse. The third is derived from the two supplied.

Raises **GeometryError**

When *hradius*, *vradius* and *eccentricity* are incorrectly supplied as parameters.

TypeError

When *center* is not a Point.

See also:[Circle](#) (page 495)**Notes**

Constructed from a center and two radii, the first being the horizontal radius (along the x-axis) and the second being the vertical radius (along the y-axis).

When symbolic value for `hradius` and `vradius` are used, any calculation that refers to the foci or the major or minor axis will assume that the ellipse has its major radius on the x-axis. If this is not true then a manual rotation is necessary.

Examples

```
>>> from diofant import Ellipse, Point, Rational
>>> e1 = Ellipse(Point(0, 0), 5, 1)
>>> e1.hradius, e1.vradius
(5, 1)
>>> e2 = Ellipse(Point(3, 1), hradius=3, eccentricity=Rational(4, 5))
>>> e2
Ellipse(Point2D(3, 1), 3, 9/5)
```

Attributes

center (page 486)	The center of the ellipse.
hradius (page 489)	The horizontal radius of the ellipse.
vradius (page 495)	The vertical radius of the ellipse.
area (page 486)	The area of the ellipse.
circumference (page 487)	The circumference of the ellipse.
eccentricity (page 487)	The eccentricity of the ellipse.
periapsis (page 492)	The periapsis of the ellipse.
apoapsis (page 485)	The apoapsis of the ellipse.
focus_distance (page 489)	The focale distance of the ellipse.
foci (page 488)	The foci of the ellipse.

apoapsis

The apoapsis of the ellipse.

The greatest distance between the focus and the contour.

Returns apoapsis : number

See also:

[periapsis](#) (page 492) Returns shortest distance between foci and contour

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.apoapsis
2*sqrt(2) + 3
```

arbitrary_point(*parameter='t'*)

A parameterized point on the ellipse.

Parameters *parameter* : str, optional

Default value is 't'.

Returns *arbitrary_point* : Point

Raises **ValueError**

When *parameter* already appears in the functions.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.arbitrary_point()
Point2D(3*cos(t), 2*sin(t))
```

area

The area of the ellipse.

Returns *area* : number

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.area
3*pi
```

center

The center of the ellipse.

Returns *center* : number

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.center
Point2D(0, 0)
```

circumference

The circumference of the ellipse.

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.circumference
12*Integral(sqrt((-8*_x**2/9 + 1)/(-_x**2 + 1)), (_x, 0, 1))
```

eccentricity

The eccentricity of the ellipse.

Returns eccentricity : number

Examples

```
>>> from diofant import Point, Ellipse, sqrt
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, sqrt(2))
>>> e1.eccentricity
sqrt(7)/3
```

encloses_point(*p*)

Return True if *p* is enclosed by (is inside of) self.

Parameters *p* : Point

Returns encloses_point : True, False or None

See also:

[diofant.geometry.point.Point](#) (page 438)

Notes

Being on the border of self is considered False.

Examples

```
>>> from diofant import Ellipse, S
>>> from diofant.abc import t
>>> e = Ellipse((0, 0), 3, 2)
```

```
>>> e.encloses_point((0, 0))
True
>>> e.encloses_point(e.arbitrary_point(t).subs(t, S.Half))
False
>>> e.encloses_point((4, 0))
False
```

equation($x='x'$, $y='y'$)

The equation of the ellipse.

Parameters x : str, optional

Label for the x-axis. Default value is 'x'.

y : str, optional

Label for the y-axis. Default value is 'y'.

Returns **equation** : diofant expression

See also:

[arbitrary_point](#) (page 486) Returns parameterized point on ellipse

Examples

```
>>> from diofant import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.equation()
y**2/4 + (x/3 - 1/3)**2 - 1
```

evolute($x='x'$, $y='y'$)

The equation of evolute of the ellipse.

Parameters x : str, optional

Label for the x-axis. Default value is 'x'.

y : str, optional

Label for the y-axis. Default value is 'y'.

Returns **equation** : diofant expression

Examples

```
>>> from diofant import Point, Ellipse
>>> e1 = Ellipse(Point(1, 0), 3, 2)
>>> e1.evolute()
2**(2/3)*y**(2/3) + (3*x - 3)**(2/3) - 5**(2/3)
```

foci

The foci of the ellipse.

Raises ValueError

When the major and minor axis cannot be determined.

See also:

[diofant.geometry.point.Point](#) (page 438)

focus_distance (page 489) Returns the distance between focus and center

Notes

The foci can only be calculated if the major/minor axes are known.

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.foci
(Point2D(-2*sqrt(2), 0), Point2D(2*sqrt(2), 0))
```

focus_distance

The focale distance of the ellipse.

The distance between the center and one focus.

Returns focus_distance : number

See also:

foci (page 488)

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.focus_distance
2*sqrt(2)
```

hradius

The horizontal radius of the ellipse.

Returns hradius : number

See also:

vradius (page 495), *major* (page 491), *minor* (page 491)

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.hradius
3
```

intersection(*o*)

The intersection of this ellipse and another geometrical entity *o*.

Parameters o : GeometryEntity

Returns intersection : list of GeometryEntity objects

See also:

[diofant.geometry.entity.GeometryEntity](#) (page 433)

Notes

Currently supports intersections with Point, Line, Segment, Ray, Circle and Ellipse types.

Examples

```
>>> from diofant import Ellipse, Point, Line, sqrt
>>> e = Ellipse(Point(0, 0), 5, 7)
>>> e.intersection(Point(0, 0))
[]
>>> e.intersection(Point(5, 0))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(0,0), Point(0, 1)))
[Point2D(0, -7), Point2D(0, 7)]
>>> e.intersection(Line(Point(5,0), Point(5, 1)))
[Point2D(5, 0)]
>>> e.intersection(Line(Point(6,0), Point(6, 1)))
[]
>>> e = Ellipse(Point(-1, 0), 4, 3)
>>> e.intersection(Ellipse(Point(1, 0), 4, 3))
[Point2D(0, -3*sqrt(15)/4), Point2D(0, 3*sqrt(15)/4)]
>>> e.intersection(Ellipse(Point(5, 0), 4, 3))
[Point2D(2, -3*sqrt(7)/4), Point2D(2, 3*sqrt(7)/4)]
>>> e.intersection(Ellipse(Point(100500, 0), 4, 3))
[]
>>> e.intersection(Ellipse(Point(0, 0), 3, 4))
[Point2D(-363/175, -48*sqrt(111)/175), Point2D(-363/175, 48*sqrt(111)/175),
↳Point2D(3, 0)]
```

```
>>> e.intersection(Ellipse(Point(-1, 0), 3, 4))
[Point2D(-17/5, -12/5), Point2D(-17/5, 12/5), Point2D(7/5, -12/5), Point2D(7/
↳5, 12/5)]
```

is_tangent(o)

Is *o* tangent to the ellipse?

Parameters *o* : GeometryEntity

An Ellipse, LinearEntity or Polygon

Returns is_tangent: boolean

True if *o* is tangent to the ellipse, False otherwise.

Raises NotImplementedError

When the wrong type of argument is supplied.

See also:

[tangent_lines](#) (page 495)

Examples

```
>>> from diofant import Point, Ellipse, Line
>>> p0, p1, p2 = Point(0, 0), Point(3, 0), Point(3, 3)
>>> e1 = Ellipse(p0, 3, 2)
>>> l1 = Line(p1, p2)
>>> e1.is_tangent(l1)
True
```

major

Longer axis of the ellipse (if it can be determined) else *hradius*.

Returns major : number or expression

See also:

hradius (page 489), *vradius* (page 495), *minor* (page 491)

Examples

```
>>> from diofant import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.major
3
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).major
a
>>> Ellipse(p1, b, a).major
b
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).major
m + 1
```

minor

Shorter axis of the ellipse (if it can be determined) else *vradius*.

Returns minor : number or expression

See also:

hradius (page 489), *vradius* (page 495), *major* (page 491)

Examples

```
>>> from diofant import Point, Ellipse, Symbol
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.minor
1
```

```
>>> a = Symbol('a')
>>> b = Symbol('b')
>>> Ellipse(p1, a, b).minor
b
>>> Ellipse(p1, b, a).minor
a
```

```
>>> m = Symbol('m')
>>> M = m + 1
>>> Ellipse(p1, m, M).minor
m
```

normal_lines(*p*, *prec=None*)

Normal lines between *p* and the ellipse.

Parameters *p* : Point

Returns **normal_lines** : list with 1, 2 or 4 Lines

Examples

```
>>> from diofant import Line, Point, Ellipse
>>> e = Ellipse((0, 0), 2, 3)
>>> c = e.center
>>> e.normal_lines(c + Point(1, 0))
[Line(Point2D(0, 0), Point2D(1, 0))]
>>> e.normal_lines(c)
[Line(Point2D(0, 0), Point2D(0, 1)), Line(Point2D(0, 0), Point2D(1, 0))]
```

Off-axis points require the solution of a quartic equation. This often leads to very large expressions that may be of little practical use. An approximate solution of *prec* digits can be obtained by passing in the desired value:

```
>>> e.normal_lines((3, 3), prec=2)
[Line(Point2D(-38/47, -85/31), Point2D(9/47, -21/17)),
Line(Point2D(19/13, -43/21), Point2D(32/13, -8/3))]
```

Whereas the above solution has an operation count of 12, the exact solution has an operation count of 2020.

periapsis

The periapsis of the ellipse.

The shortest distance between the focus and the contour.

Returns **periapsis** : number

See also:

[apoapsis](#) (page 485) Returns greatest distance between focus and contour

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
```



```
>>> e1.periapsis
-2*sqrt(2) + 3
```

plot_interval(*parameter='t'*)

The plot interval for the default geometric plot of the Ellipse.

Parameters *parameter* : str, optional

Default value is 't'.

Returns *plot_interval* : list

[*parameter*, *lower_bound*, *upper_bound*]

Examples

```
>>> from diofant import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.plot_interval()
[t, -pi, pi]
```

random_point(*seed=None*)

A random point on the ellipse.

Returns *point* : Point

See also:

[diofant.geometry.point.Point](#) (page 438)

[arbitrary_point](#) (page 486) Returns parameterized point on ellipse

Notes

An *arbitrary_point* with a random value of *t* substituted into it may not test as being on the ellipse because the expression tested that a point is on the ellipse doesn't simplify to zero and doesn't evaluate exactly to zero:

```
>>> from diofant.abc import t
>>> e1.arbitrary_point(t)
Point2D(3*cos(t), 2*sin(t))
>>> p2 = _.subs(t, 0.1)
>>> p2 in e1
False
```

Note that *arbitrary_point* routine does not take this approach. A value for $\cos(t)$ and $\sin(t)$ (not *t*) is substituted into the arbitrary point. There is a small chance that this will give a point that will not test as being in the ellipse, so the process is repeated (up to 10 times) until a valid point is obtained.

Examples

```
>>> from diofant import Point, Ellipse, Segment
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.random_point() # gives some random point
Point2D(...)
```

```
>>> p1 = e1.random_point(seed=0); p1.n(2)
Point2D(2.1, 1.4)
```

The `random_point` method assures that the point will test as being in the ellipse:

```
>>> p1 in e1
True
```

reflect(*line*)

Override `GeometryEntity.reflect` since the radius is not a `GeometryEntity`.

Notes

Until the general ellipse (with no axis parallel to the x-axis) is supported a `NotImplementedError` is raised and the equation whose zeros define the rotated ellipse is given.

Examples

```
>>> from diofant import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
>>> from diofant import Ellipse, Line, Point
>>> Ellipse(Point(3, 4), 1, 3).reflect(Line(Point(0, -4), Point(5, 0)))
Traceback (most recent call last):
...
NotImplementedError:
General Ellipse is not supported but the equation of the reflected
Ellipse is given by the zeros of:  $f(x, y) = (9*x/41 + 40*y/41 + 37/41)**2 + (40*x/123 - 3*y/41 - 364/123)**2 - 1$ 
```

rotate(*angle=0*, *pt=None*)

Rotate *angle* radians counterclockwise about Point *pt*.

Note: since the general ellipse is not supported, only rotations that are integer multiples of $\pi/2$ are allowed.

Examples

```
>>> from diofant import Ellipse, pi
>>> Ellipse((1, 0), 2, 1).rotate(pi/2)
Ellipse(Point2D(0, 1), 1, 2)
>>> Ellipse((1, 0), 2, 1).rotate(pi)
Ellipse(Point2D(-1, 0), 2, 1)
```

scale(*x=1*, *y=1*, *pt=None*)

Override `GeometryEntity.scale` since it is the major and minor axes which must be scaled and they are not `GeometryEntities`.

Examples

```
>>> from diofant import Ellipse
>>> Ellipse((0, 0), 2, 1).scale(2, 4)
Circle(Point2D(0, 0), 4)
>>> Ellipse((0, 0), 2, 1).scale(2)
Ellipse(Point2D(0, 0), 4, 1)
```

tangent_lines(*p*)

Tangent lines between *p* and the ellipse.

If *p* is on the ellipse, returns the tangent line through point *p*. Otherwise, returns the tangent line(s) from *p* to the ellipse, or None if no tangent line is possible (e.g., *p* inside ellipse).

Parameters *p* : Point

Returns **tangent_lines** : list with 1 or 2 Lines

Raises **NotImplementedError**

Can only find tangent lines for a point, *p*, on the ellipse.

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Line](#) (page 457)

Examples

```
>>> from diofant import Point, Ellipse
>>> e1 = Ellipse(Point(0, 0), 3, 2)
>>> e1.tangent_lines(Point(3, 0))
[Line(Point2D(3, 0), Point2D(3, -12))]
```

vradius

The vertical radius of the ellipse.

Returns **vradius** : number

See also:

[hradius](#) (page 489), [major](#) (page 491), [minor](#) (page 491)

Examples

```
>>> from diofant import Point, Ellipse
>>> p1 = Point(0, 0)
>>> e1 = Ellipse(p1, 3, 1)
>>> e1.vradius
1
```

class diofant.geometry.ellipse.Circle

A circle in space.

Constructed simply from a center and a radius, or from three non-collinear points.

Parameters `center` : Point

`radius` : number or diofant expression

`points` : sequence of three Points

Raises `GeometryError`

When trying to construct circle from three collinear points. When trying to construct circle from incorrect parameters.

See also:

Ellipse (page 484), *diofant.geometry.point.Point* (page 438)

Examples

```
>>> from diofant.geometry import Point, Circle
>>> # a circle constructed from a center and radius
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.hradius, c1.vradius, c1.radius
(5, 5, 5)
```

```
>>> # a circle costructed from three points
>>> c2 = Circle(Point(0, 0), Point(1, 1), Point(1, 0))
>>> c2.hradius, c2.vradius, c2.radius, c2.center
(sqrt(2)/2, sqrt(2)/2, sqrt(2)/2, Point2D(1/2, 1/2))
```

Attributes

<i>circumference</i> (page 496)	The circumference of the circle.
<i>equation</i> (page 496)([x, y])	The equation of the circle.

radius (synonymous with hradius, vradius, major and minor)
--

circumference

The circumference of the circle.

Returns `circumference` : number or Diofant expression

Examples

```
>>> from diofant import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.circumference
12*pi
```

equation(x='x', y='y')

The equation of the circle.

Parameters `x` : str or Symbol, optional

Default value is 'x'.

`y` : str or Symbol, optional

Default value is 'y'.

Returns equation : Diofant expression

Examples

```
>>> from diofant import Point, Circle
>>> c1 = Circle(Point(0, 0), 5)
>>> c1.equation()
x**2 + y**2 - 25
```

intersection(*o*)

The intersection of this circle with another geometrical entity.

Parameters *o* : GeometryEntity

Returns intersection : list of GeometryEntities

Examples

```
>>> from diofant import Point, Circle, Line, Ray
>>> p1, p2, p3 = Point(0, 0), Point(5, 5), Point(6, 0)
>>> p4 = Point(5, 0)
>>> c1 = Circle(p1, 5)
>>> c1.intersection(p2)
[]
>>> c1.intersection(p4)
[Point2D(5, 0)]
>>> c1.intersection(Ray(p1, p2))
[Point2D(5*sqrt(2)/2, 5*sqrt(2)/2)]
>>> c1.intersection(Line(p2, p3))
[]
```

radius

The radius of the circle.

Returns radius : number or diofant expression

See also:

[Ellipse.major](#) (page 491), [Ellipse.minor](#) (page 491), [Ellipse.hradius](#) (page 489), [Ellipse.vradius](#) (page 495)

Examples

```
>>> from diofant import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.radius
6
```

reflect(*line*)

Override GeometryEntity.reflect since the radius is not a GeometryEntity.

Examples

```
>>> from diofant import Circle, Line
>>> Circle((0, 1), 1).reflect(Line((0, 0), (1, 1)))
Circle(Point2D(1, 0), -1)
```

scale(*x=1, y=1, pt=None*)

Override `GeometryEntity.scale` since the radius is not a `GeometryEntity`.

Examples

```
>>> from diofant import Circle
>>> Circle((0, 0), 1).scale(2, 2)
Circle(Point2D(0, 0), 2)
>>> Circle((0, 0), 1).scale(2, 4)
Ellipse(Point2D(0, 0), 2, 4)
```

vradius

This `Ellipse` property is an alias for the `Circle`'s radius.

Whereas `hradius`, `major` and `minor` can use `Ellipse`'s conventions, the `vradius` does not exist for a circle. It is always a positive value in order that the `Circle`, like `Polygons`, will have an area that can be positive or negative as determined by the sign of the `hradius`.

Examples

```
>>> from diofant import Point, Circle
>>> c1 = Circle(Point(3, 4), 6)
>>> c1.vradius
6
```

Polygons

class `diofant.geometry.polygon.Polygon`

A two-dimensional polygon.

A simple polygon in space. Can be constructed from a sequence of points or from a center, radius, number of sides and rotation angle.

Parameters `vertices` : sequence of `Points`

Raises `GeometryError`

If all parameters are not `Points`.

If the `Polygon` has intersecting sides.

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Segment](#) (page 463), [Triangle](#) (page 512)

Notes

Polygons are treated as closed paths rather than 2D areas so some calculations can be negative or positive (e.g., area) based on the orientation of the points.

Any consecutive identical points are reduced to a single point and any points collinear and between two points will be removed unless they are needed to define an explicit intersection (see examples).

A Triangle, Segment or Point will be returned when there are 3 or fewer points provided.

Examples

```
>>> from diofant import Point, Polygon, pi
>>> p1, p2, p3, p4, p5 = [(0, 0), (1, 0), (5, 1), (0, 1), (3, 0)]
>>> Polygon(p1, p2, p3, p4)
Polygon(Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1))
>>> Polygon(p1, p2)
Segment(Point2D(0, 0), Point2D(1, 0))
>>> Polygon(p1, p2, p5)
Segment(Point2D(0, 0), Point2D(3, 0))
```

While the sides of a polygon are not allowed to cross implicitly, they can do so explicitly. For example, a polygon shaped like a Z with the top left connecting to the bottom right of the Z must have the point in the middle of the Z explicitly given:

```
>>> mid = Point(1, 1)
>>> Polygon((0, 2), (2, 2), mid, (0, 0), (2, 0), mid).area
0
>>> Polygon((0, 2), (2, 2), mid, (2, 0), (0, 0), mid).area
-2
```

When the the keyword n is used to define the number of sides of the Polygon then a RegularPolygon is created and the other arguments are interpreted as center, radius and rotation. The unrotated RegularPolygon will always have a vertex at $\text{Point}(r, 0)$ where r is the radius of the circle that circumscribes the RegularPolygon. Its method `spin` can be used to increment that angle.

```
>>> p = Polygon((0,0), 1, n=3)
>>> p
RegularPolygon(Point2D(0, 0), 1, 3, 0)
>>> p.vertices[0]
Point2D(1, 0)
>>> p.args[0]
Point2D(0, 0)
>>> p.spin(pi/2)
>>> p.vertices[0]
Point2D(0, 1)
```

Attributes

`area` (page 501)

The area of the polygon.

Continued on next page

Table 3.13 – continued from previous page

angles (page 500)	The internal angle at each vertex.
perimeter (page 503)	The perimeter of the polygon.
vertices (page 504)	The vertices of the polygon.
centroid (page 501)	The centroid of the polygon.
sides (page 504)	The line segments that form the sides of the polygon.

angles

The internal angle at each vertex.

Returns `angles` : dict

A dictionary where each key is a vertex and each value is the internal angle at that vertex. The vertices are represented as Points.

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.LinearEntity.angle_between](#) (page 450)

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.angles[p1]
pi/2
>>> poly.angles[p2]
acos(-4*sqrt(17)/17)
```

`arbitrary_point(parameter='t')`

A parameterized point on the polygon.

The parameter, varying from 0 to 1, assigns points to the position on the perimeter that is that fraction of the total perimeter. So the point evaluated at $t=1/2$ would return the point from the first vertex that is 1/2 way around the polygon.

Parameters `parameter` : str, optional

Default value is 't'.

Returns `arbitrary_point` : Point

Raises `ValueError`

When `parameter` already appears in the Polygon's definition.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant import Polygon, S, Symbol
>>> t = Symbol('t', extended_real=True)
>>> tri = Polygon((0, 0), (1, 0), (1, 1))
>>> p = tri.arbitrary_point('t')
```



```
>>> perimeter = tri.perimeter
>>> s1, s2 = [s.length for s in tri.sides[:2]]
>>> p.subs(t, (s1 + s2/2)/perimeter)
Point2D(1, 1/2)
```

area

The area of the polygon.

See also:

[diofant.geometry.ellipse.Ellipse.area](#) (page 486)

Notes

The area calculation can be positive or negative based on the orientation of the points.

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.area
3
```

centroid

The centroid of the polygon.

Returns centroid : Point

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.util.centroid](#) (page 437)

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.centroid
Point2D(31/18, 11/18)
```

distance(o)

Returns the shortest distance between self and o.

If o is a point, then self does not need to be convex. If o is another polygon self and o must be complex.

Examples

```
>>> from diofant import Point, Polygon, RegularPolygon
>>> p1, p2 = map(Point, [(0, 0), (7, 5)])
>>> poly = Polygon(*RegularPolygon(p1, 1, 3).vertices)
>>> poly.distance(p2)
sqrt(61)
```

encloses_point(*p*)

Return True if *p* is enclosed by (is inside of) self.

Parameters *p* : Point

Returns **encloses_point** : True, False or None

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.ellipse.Ellipse.encloses_point](#) (page 487)

Notes

Being on the border of self is considered False.

References

[R342] (page 1269)

Examples

```
>>> from diofant import Polygon, Point
>>> from diofant.abc import t
>>> p = Polygon((0, 0), (4, 0), (4, 4))
>>> p.encloses_point(Point(2, 1))
True
>>> p.encloses_point(Point(2, 2))
False
>>> p.encloses_point(Point(5, 5))
False
```

intersection(*o*)

The intersection of two polygons.

The intersection may be empty and can contain individual Points and complete Line Segments.

Parameters **other**: Polygon

Returns **intersection** : list

The list of Segments and Points

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Segment](#) (page 463)

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly1 = Polygon(p1, p2, p3, p4)
>>> p5, p6, p7 = map(Point, [(3, 2), (1, -1), (0, 2)])
>>> poly2 = Polygon(p5, p6, p7)
>>> poly1.intersection(poly2)
[Point2D(2/3, 0), Point2D(9/5, 1/5), Point2D(7/3, 1), Point2D(1/3, 1)]
```

`is_convex()`

Is the polygon convex?

A polygon is convex if all its interior angles are less than 180 degrees.

Returns `is_convex` : boolean

True if this polygon is convex, False otherwise.

See also:

[diofant.geometry.util.convex_hull](#) (page 436)

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.is_convex()
True
```

`perimeter`

The perimeter of the polygon.

Returns `perimeter` : number or Basic instance

See also:

[diofant.geometry.line.Segment.length](#) (page 464)

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.perimeter
sqrt(17) + 7
```

`plot_interval(parameter='t')`

The plot interval for the default geometric plot of the polygon.

Parameters `parameter` : str, optional

Default value is 't'.

Returns `plot_interval` : list (plot interval)

[parameter, lower_bound, upper_bound]

Examples

```
>>> from diofant import Polygon
>>> p = Polygon((0, 0), (1, 0), (1, 1))
>>> p.plot_interval()
[t, 0, 1]
```

sides

The line segments that form the sides of the polygon.

Returns sides : list of sides

Each side is a Segment.

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Segment](#) (page 463)

Notes

The Segments that represent the sides are an undirected line segment so cannot be used to tell the orientation of the polygon.

Examples

```
>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.sides
[Segment(Point2D(0, 0), Point2D(1, 0)),
Segment(Point2D(1, 0), Point2D(5, 1)),
Segment(Point2D(0, 1), Point2D(5, 1)), Segment(Point2D(0, 0), Point2D(0, 1))]
```

vertices

The vertices of the polygon.

Returns vertices : tuple of Points

See also:

[diofant.geometry.point.Point](#) (page 438)

Notes

When iterating over the vertices, it is more efficient to index self rather than to request the vertices and index them. Only use the vertices when you want to process all of them at once. This is even more important with RegularPolygons that calculate each vertex.

Examples

```

>>> from diofant import Point, Polygon
>>> p1, p2, p3, p4 = map(Point, [(0, 0), (1, 0), (5, 1), (0, 1)])
>>> poly = Polygon(p1, p2, p3, p4)
>>> poly.vertices
(Point2D(0, 0), Point2D(1, 0), Point2D(5, 1), Point2D(0, 1))
>>> poly.args[0]
Point2D(0, 0)

```

class diofant.geometry.polygon.RegularPolygon

A regular polygon.

Such a polygon has all internal angles equal and all sides the same length.

Parameters *center* : Point

radius : number or Basic instance

The distance from the center to a vertex

n : int

The number of sides

Raises GeometryError

If the *center* is not a Point, or the *radius* is not a number or Basic instance, or the number of sides, *n*, is less than three.

See also:

[diofant.geometry.point.Point](#) (page 438), [Polygon](#) (page 498)

Notes

A RegularPolygon can be instantiated with Polygon with the kwarg *n*.

Regular polygons are instantiated with a center, radius, number of sides and a rotation angle. Whereas the arguments of a Polygon are vertices, the vertices of the RegularPolygon must be obtained with the vertices method.

Examples

```

>>> from diofant.geometry import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r
RegularPolygon(Point2D(0, 0), 5, 3, 0)
>>> r.vertices[0]
Point2D(5, 0)

```

Attributes

vertices (page 512)	The vertices of the RegularPolygon.
center (page 507)	The center of the RegularPolygon
radius (page 510)	Radius of the RegularPolygon
Continued on next page	

Table 3.14 - continued from previous page

rotation (page 511)	CCW angle by which the RegularPolygon is rotated
apothem (page 506)	The inradius of the RegularPolygon.
interior_angle (page 510)	Measure of the interior angles.
exterior_angle (page 509)	Measure of the exterior angles.
circumcircle (page 508)	The circumcircle of the RegularPolygon.
incircle (page 509)	The incircle of the RegularPolygon.
angles (page 506)	Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

angles

Returns a dictionary with keys, the vertices of the Polygon, and values, the interior angle at each vertex.

Examples

```
>>> from diofant import RegularPolygon, Point2D, sqrt, pi
>>> r = RegularPolygon(Point2D(0, 0), 5, 3)
>>> r.angles == {Point2D(-5/2, -5*sqrt(3)/2): pi/3,
...             Point2D(-5/2, 5*sqrt(3)/2): pi/3,
...             Point2D(5, 0): pi/3}
True
```

apothem

The inradius of the RegularPolygon.

The apothem/inradius is the radius of the inscribed circle.

Returns apothem : number or instance of Basic

See also:

[diofant.geometry.line.Segment.length](#) (page 464), [diofant.geometry.ellipse.Circle.radius](#) (page 497)

Examples

```
>>> from diofant import Symbol
>>> from diofant.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.apothem
sqrt(2)*r/2
```

area

Returns the area.

Examples

```
>>> from diofant.geometry import RegularPolygon
>>> square = RegularPolygon((0, 0), 1, 4)
>>> square.area
2
>>> _ == square.length**2
True
```

args

Returns the center point, the radius, the number of sides, and the orientation angle.

Examples

```
>>> from diofant import RegularPolygon, Point
>>> r = RegularPolygon(Point(0, 0), 5, 3)
>>> r.args
(Point2D(0, 0), 5, 3, 0)
```

center

The center of the RegularPolygon

This is also the center of the circumscribing circle.

Returns center : Point

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.ellipse.Ellipse.center](#) (page 486)

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

centroid

The center of the RegularPolygon

This is also the center of the circumscribing circle.

Returns center : Point

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.ellipse.Ellipse.center](#) (page 486)

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.center
Point2D(0, 0)
```

circumcenter

Alias for center.

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.circumcenter
Point2D(0, 0)
```

circumcircle

The circumcircle of the RegularPolygon.

Returns circumcircle : Circle

See also:

[circumcenter](#) (page 507), [diofant.geometry.ellipse.Circle](#) (page 495)

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.circumcircle
Circle(Point2D(0, 0), 4)
```

circumradius

Alias for radius.

Examples

```
>>> from diofant import Symbol
>>> from diofant.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.circumradius
r
```

encloses_point(p)

Return True if p is enclosed by (is inside of) self.

Parameters p : Point

Returns encloses_point : True, False or None

See also:

[diofant.geometry.ellipse.Ellipse.encloses_point](#) (page 487)

Notes

Being on the border of self is considered False.

The general Polygon.encloses_point method is called only if a point is not within or beyond the incircle or circumcircle, respectively.

Examples

```

>>> from diofant import RegularPolygon, S, Point, Symbol
>>> p = RegularPolygon((0, 0), 3, 4)
>>> p.encloses_point(Point(0, 0))
True
>>> r, R = p.inradius, p.circumradius
>>> p.encloses_point(Point((r + R)/2, 0))
True
>>> p.encloses_point(Point(R/2, R/2 + (R - r)/10))
False
>>> t = Symbol('t', extended_real=True)
>>> p.encloses_point(p.arbitrary_point().subs(t, S.Half))
False
>>> p.encloses_point(Point(5, 5))
False

```

exterior_angle

Measure of the exterior angles.

Returns exterior_angle : number

See also:

[diofant.geometry.line.LinearEntity.angle_between](#) (page 450)

Examples

```

>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.exterior_angle
pi/4

```

incircle

The incircle of the RegularPolygon.

Returns incircle : Circle

See also:

[inradius](#) (page 509), [diofant.geometry.ellipse.Circle](#) (page 495)

Examples

```

>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 7)
>>> rp.incicle
Circle(Point2D(0, 0), 4*cos(pi/7))

```

inradius

Alias for apothem.

Examples

```
>>> from diofant import Symbol
>>> from diofant.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.inradius
sqrt(2)*r/2
```

interior_angle

Measure of the interior angles.

Returns interior_angle : number

See also:

[diofant.geometry.line.LinearEntity.angle_between](#) (page 450)

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 4, 8)
>>> rp.interior_angle
3*pi/4
```

length

Returns the length of the sides.

The half-length of the side and the apothem form two legs of a right triangle whose hypotenuse is the radius of the regular polygon.

Examples

```
>>> from diofant.geometry import RegularPolygon
>>> from diofant import sqrt
>>> s = square_in_unit_circle = RegularPolygon((0, 0), 1, 4)
>>> s.length
sqrt(2)
>>> sqrt((_/2)**2 + s.apothem**2) == s.radius
True
```

radius

Radius of the RegularPolygon

This is also the radius of the circumscribing circle.

Returns radius : number or instance of Basic

See also:

[diofant.geometry.line.Segment.length](#) (page 464), [diofant.geometry.ellipse.Circle.radius](#) (page 497)

Examples

```
>>> from diofant import Symbol
>>> from diofant.geometry import RegularPolygon, Point
>>> radius = Symbol('r')
>>> rp = RegularPolygon(Point(0, 0), radius, 4)
>>> rp.radius
r
```

reflect(*line*)

Override GeometryEntity.reflect since this is not made of only points.

```
>>> from diofant import RegularPolygon, Line
```

```
>>> RegularPolygon((0, 0), 1, 4).reflect(Line((0, 1), slope=-2))
RegularPolygon(Point2D(4/5, 2/5), -1, 4, acos(3/5))
```

rotate(*angle*, *pt=None*)

Override GeometryEntity.rotate to first rotate the RegularPolygon about its center.

```
>>> from diofant import Point, RegularPolygon, Polygon, pi
>>> t = RegularPolygon(Point(1, 0), 1, 3)
>>> t.vertices[0] # vertex on x-axis
Point2D(2, 0)
>>> t.rotate(pi/2).vertices[0] # vertex on y axis now
Point2D(0, 2)
```

See also:

[rotation](#) (page 511)

[spin](#) (page 512) Rotates a RegularPolygon in place

rotation

CCW angle by which the RegularPolygon is rotated

Returns rotation : number or instance of Basic

Examples

```
>>> from diofant import pi
>>> from diofant.geometry import RegularPolygon, Point
>>> RegularPolygon(Point(0, 0), 3, 4, pi).rotation
pi
```

scale(*x=1*, *y=1*, *pt=None*)

Override GeometryEntity.scale since it is the radius that must be scaled (if $x == y$) or else a new Polygon must be returned.

```
>>> from diofant import RegularPolygon
```

Symmetric scaling returns a RegularPolygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 2)
RegularPolygon(Point2D(0, 0), 2, 4, 0)
```

Asymmetric scaling returns a kite as a Polygon:

```
>>> RegularPolygon((0, 0), 1, 4).scale(2, 1)
Polygon(Point2D(2, 0), Point2D(0, 1), Point2D(-2, 0), Point2D(0, -1))
```

spin(*angle*)

Increment *in place* the virtual Polygon's rotation by ccw angle.

See also: rotate method which moves the center.

```
>>> from diofant import Polygon, Point, pi
>>> r = Polygon(Point(0,0), 1, n=3)
>>> r.vertices[0]
Point2D(1, 0)
>>> r.spin(pi/6)
>>> r.vertices[0]
Point2D(sqrt(3)/2, 1/2)
```

See also:

[rotation](#) (page 511)

rotate (**page 511**) Creates a copy of the RegularPolygon rotated about a Point

vertices

The vertices of the RegularPolygon.

Returns vertices : list

Each vertex is a Point.

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant.geometry import RegularPolygon, Point
>>> rp = RegularPolygon(Point(0, 0), 5, 4)
>>> rp.vertices
[Point2D(5, 0), Point2D(0, 5), Point2D(-5, 0), Point2D(0, -5)]
```

class diofant.geometry.polygon.**Triangle**

A polygon with three vertices and three sides.

Parameters points : sequence of Points

keyword: asa, sas, or sss to specify sides/angles of the triangle

Raises GeometryError

If the number of vertices is not equal to three, or one of the vertices is not a Point, or a valid keyword is not given.

See also:

[diofant.geometry.point.Point](#) (page 438), [Polygon](#) (page 498)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
Triangle(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Keywords `sss`, `sas`, or `asa` can be used to give the desired side lengths (in order) and interior angles (in degrees) that define the triangle:

```
>>> Triangle(sss=(3, 4, 5))
Triangle(Point2D(0, 0), Point2D(3, 0), Point2D(3, 4))
>>> Triangle(asa=(30, 1, 30))
Triangle(Point2D(0, 0), Point2D(1, 0), Point2D(1/2, sqrt(3)/6))
>>> Triangle(sas=(1, 45, 2))
Triangle(Point2D(0, 0), Point2D(2, 0), Point2D(sqrt(2)/2, sqrt(2)/2))
```

Attributes

<i>vertices</i> (page 519)	The triangle's vertices
<i>altitudes</i> (page 513)	The altitudes of the triangle.
<i>orthocenter</i> (page 518)	The orthocenter of the triangle.
<i>circumcenter</i> (page 514)	The circumcenter of the triangle
<i>circumradius</i> (page 515)	The radius of the circumcircle of the triangle.
<i>circumcircle</i> (page 514)	The circle which passes through the three vertices of the triangle.
<i>inradius</i> (page 516)	The radius of the incircle.
<i>incircle</i> (page 515)	The incircle of the triangle.
<i>medians</i> (page 518)	The medians of the triangle.
<i>medial</i> (page 518)	The medial triangle of the triangle.

altitudes

The altitudes of the triangle.

An altitude of a triangle is a segment through a vertex, perpendicular to the opposite side, with length being the height of the vertex measured from the line containing the side.

Returns altitudes : dict

The dictionary consists of keys which are vertices and values which are Segments.

See also:

[*diofant.geometry.point.Point*](#) (page 438), [*diofant.geometry.line.Segment.length*](#) (page 464)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.altitudes[p1]
Segment(Point2D(0, 0), Point2D(1/2, 1/2))
```

bisectors()

The angle bisectors of the triangle.

An angle bisector of a triangle is a straight line through a vertex which cuts the corresponding angle in half.

Returns bisectors : dict

Each key is a vertex (Point) and each value is the corresponding bisector (Segment).

See also:

[diofant.geometry.point.Point](#) (page 438), [diofant.geometry.line.Segment](#) (page 463)

Examples

```
>>> from diofant.geometry import Point, Triangle, Segment
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> from diofant import sqrt
>>> t.bisectors()[p2] == Segment(Point(0, sqrt(2) - 1), Point(1, 0))
True
```

circumcenter

The circumcenter of the triangle

The circumcenter is the center of the circumcircle.

Returns circumcenter : Point

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcenter
Point2D(1/2, 1/2)
```

circumcircle

The circle which passes through the three vertices of the triangle.

Returns circumcircle : Circle

See also:

[diofant.geometry.ellipse.Circle](#) (page 495)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.circumcircle
Circle(Point2D(1/2, 1/2), sqrt(2)/2)
```

circumradius

The radius of the circumcircle of the triangle.

Returns circumradius : number of Basic instance

See also:

[diofant.geometry.ellipse.Circle.radius](#) (page 497)

Examples

```
>>> from diofant import Symbol
>>> from diofant.geometry import Point, Triangle
>>> a = Symbol('a')
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, a)
>>> t = Triangle(p1, p2, p3)
>>> t.circumradius
sqrt(a**2/4 + 1/4)
```

incenter

The center of the incircle.

The incircle is the circle which lies inside the triangle and touches all three sides.

Returns incenter : Point

See also:

[incircle](#) (page 515), [diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.incenter
Point2D(-sqrt(2)/2 + 1, -sqrt(2)/2 + 1)
```

incircle

The incircle of the triangle.

The incircle is the circle which lies inside the triangle and touches all three sides.

Returns incircle : Circle

See also:

[diofant.geometry.ellipse.Circle](#) (page 495)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(2, 0), Point(0, 2)
>>> t = Triangle(p1, p2, p3)
>>> t.incircle
Circle(Point2D(-sqrt(2) + 2, -sqrt(2) + 2), -sqrt(2) + 2)
```

inradius

The radius of the incircle.

Returns inradius : number of Basic instance

See also:

incircle (page 515), *diofant.geometry.ellipse.Circle.radius* (page 497)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(4, 0), Point(0, 3)
>>> t = Triangle(p1, p2, p3)
>>> t.inradius
1
```

is_equilateral()

Are all the sides the same length?

Returns is_equilateral : boolean

See also:

diofant.geometry.entity.GeometryEntity.is_similar (page 433), *Regular-Polygon* (page 505), *is_isosceles* (page 516), *is_right* (page 517), *is_scalene* (page 517)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_equilateral()
False
```

```
>>> from diofant import sqrt
>>> t2 = Triangle(Point(0, 0), Point(10, 0), Point(5, 5*sqrt(3)))
>>> t2.is_equilateral()
True
```

is_isosceles()

Are two or more of the sides the same length?

Returns is_isosceles : boolean

See also:

is_equilateral (page 516), *is_right* (page 517), *is_scalene* (page 517)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(2, 4))
>>> t1.is_isosceles()
True
```

is_right()

Is the triangle right-angled.

Returns is_right : boolean

See also:

diofant.geometry.line.LinearEntity.is_perpendicular (page 453),
is_equilateral (page 516), *is_isosceles* (page 516), *is_scalene* (page 517)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t1.is_right()
True
```

is_scalene()

Are all the sides of the triangle of different lengths?

Returns is_scalene : boolean

See also:

is_equilateral (page 516), *is_isosceles* (page 516), *is_right* (page 517)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(1, 4))
>>> t1.is_scalene()
True
```

is_similar(*other*)

Is another triangle similar to this one.

Two triangles are similar if one can be uniformly scaled to the other.

Parameters other: **Triangle**

Returns is_similar : boolean

See also:

diofant.geometry.entity.GeometryEntity.is_similar (page 433)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t1 = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -3))
>>> t1.is_similar(t2)
True
```

```
>>> t2 = Triangle(Point(0, 0), Point(-4, 0), Point(-4, -4))
>>> t1.is_similar(t2)
False
```

medial

The medial triangle of the triangle.

The triangle which is formed from the midpoints of the three sides.

Returns medial : Triangle

See also:

[diofant.geometry.line.Segment.midpoint](#) (page 465)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medial
Triangle(Point2D(1/2, 0), Point2D(1/2, 1/2), Point2D(0, 1/2))
```

medians

The medians of the triangle.

A median of a triangle is a straight line through a vertex and the midpoint of the opposite side, and divides the triangle into two equal areas.

Returns medians : dict

Each key is a vertex (Point) and each value is the median (Segment) at that point.

See also:

[diofant.geometry.point.Point.midpoint](#) (page 441), [diofant.geometry.line.Segment.midpoint](#) (page 465)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.medians[p1]
Segment(Point2D(0, 0), Point2D(1/2, 1/2))
```

orthocenter

The orthocenter of the triangle.

The orthocenter is the intersection of the altitudes of a triangle. It may lie inside, outside or on the triangle.

Returns orthocenter : Point

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant.geometry import Point, Triangle
>>> p1, p2, p3 = Point(0, 0), Point(1, 0), Point(0, 1)
>>> t = Triangle(p1, p2, p3)
>>> t.orthocenter
Point2D(0, 0)
```

vertices

The triangle's vertices

Returns vertices : tuple

Each element in the tuple is a Point

See also:

[diofant.geometry.point.Point](#) (page 438)

Examples

```
>>> from diofant.geometry import Triangle, Point
>>> t = Triangle(Point(0, 0), Point(4, 0), Point(4, 3))
>>> t.vertices
(Point2D(0, 0), Point2D(4, 0), Point2D(4, 3))
```

Plane

Geometrical Planes.

Contains

Plane

class diofant.geometry.plane.Plane

A plane is a flat, two-dimensional surface. A plane is the two-dimensional analogue of a point (zero-dimensions), a line (one-dimension) and a solid (three-dimensions). A plane can generally be constructed by two types of inputs. They are three non-collinear points and a point and the plane's normal vector.

Examples

```
>>> from diofant import Plane, Point3D
>>> from diofant.abc import x
>>> Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
```

```
>>> Plane((1, 1, 1), (2, 3, 4), (2, 2, 2))
Plane(Point3D(1, 1, 1), (-1, 2, -1))
>>> Plane(Point3D(1, 1, 1), normal_vector=(1,4,7))
Plane(Point3D(1, 1, 1), (1, 4, 7))
```

Attributes

<code>p1</code> (page 523)	The only defining point of the plane.
<code>normal_vector</code> (page 523)	Normal vector of the given plane.

`angle_between(o)`

Angle between the plane and other geometric entity.

Parameters `LinearEntity3D, Plane`.

Returns `angle` : angle in radians

Notes

This method accepts only 3D entities as it's parameter, but if you want to calculate the angle between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the angle.

Examples

```
>>> from diofant import Point3D, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 2), normal_vector=(1, 2, 3))
>>> b = Line3D(Point3D(1, 3, 4), Point3D(2, 2, 2))
>>> a.angle_between(b)
-asin(sqrt(21)/6)
```

`arbitrary_point(t=None)`

Returns an arbitrary point on the Plane; varying t from 0 to 2π will move the point in a circle of radius 1 about `p1` of the Plane.

Returns `Point3D`

Examples

```
>>> from diofant.geometry.plane import Plane
>>> from diofant.abc import t
>>> p = Plane((0, 0, 0), (0, 0, 1), (0, 1, 0))
>>> p.arbitrary_point(t)
Point3D(0, cos(t), sin(t))
>>> _.distance(p.p1).simplify()
1
```

`static are_concurrent(*planes)`

Is a sequence of Planes concurrent?

Two or more Planes are concurrent if their intersections are a common line.

Parameters planes: list**Returns** Boolean**Examples**

```

>>> from diofant import Plane, Point3D
>>> a = Plane(Point3D(5, 0, 0), normal_vector=(1, -1, 1))
>>> b = Plane(Point3D(0, -2, 0), normal_vector=(3, 1, 1))
>>> c = Plane(Point3D(0, -1, 0), normal_vector=(5, -1, 9))
>>> Plane.are_concurrent(a, b)
True
>>> Plane.are_concurrent(a, b, c)
False

```

distance(*o*)

Distance between the plane and another geometric entity.

Parameters **Point3D, LinearEntity3D, Plane.****Returns** distance**Notes**

This method accepts only 3D entities as it's parameter, but if you want to calculate the distance between a 2D entity and a plane you should first convert to a 3D entity by projecting onto a desired plane and then proceed to calculate the distance.

Examples

```

>>> from diofant import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.distance(b)
sqrt(3)
>>> c = Line3D(Point3D(2, 3, 1), Point3D(1, 2, 2))
>>> a.distance(c)
0

```

equation(*x=None, y=None, z=None*)

The equation of the Plane.

Examples

```

>>> from diofant import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 2), Point3D(2, 4, 7), Point3D(3, 5, 1))
>>> a.equation()
-23*x + 11*y - 2*z + 16
>>> a = Plane(Point3D(1, 4, 2), normal_vector=(6, 6, 6))
>>> a.equation()
6*x + 6*y + 6*z - 42

```

intersection(*o*)

The intersection with other geometrical entity.

Parameters *Point*, *Point3D*, *LinearEntity*, *LinearEntity3D*, *Plane*

Returns List

Examples

```
>>> from diofant import Point, Point3D, Line, Line3D, Plane
>>> a = Plane(Point3D(1, 2, 3), normal_vector=(1, 1, 1))
>>> b = Point3D(1, 2, 3)
>>> a.intersection(b)
[Point3D(1, 2, 3)]
>>> c = Line3D(Point3D(1, 4, 7), Point3D(2, 2, 2))
>>> a.intersection(c)
[Point3D(2, 2, 2)]
>>> d = Plane(Point3D(6, 0, 0), normal_vector=(2, -5, 3))
>>> e = Plane(Point3D(2, 0, 0), normal_vector=(3, 4, -3))
>>> d.intersection(e)
[Line3D(Point3D(78/23, -24/23, 0), Point3D(147/23, 321/23, 23))]
```

is_coplanar(*o*)

Returns True if *o* is coplanar with self, else False.

Examples

```
>>> from diofant import Plane, Point3D
>>> o = (0, 0, 0)
>>> p = Plane(o, (1, 1, 1))
>>> p2 = Plane(o, (2, 2, 2))
>>> p == p2
False
>>> p.is_coplanar(p2)
True
```

is_parallel(*l*)

Is the given geometric entity parallel to the plane?

Parameters *LinearEntity3D* or *Plane*

Returns Boolean

Examples

```
>>> from diofant import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(3,1,3), normal_vector=(4, 8, 12))
>>> a.is_parallel(b)
True
```

is_perpendicular(*l*)

is the given geometric entity perpendicular to the given plane?

Parameters *LinearEntity3D* or *Plane*

Returns Boolean

Examples

```
>>> from diofant import Plane, Point3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> b = Plane(Point3D(2, 2, 2), normal_vector=(-1, 2, -1))
>>> a.is_perpendicular(b)
True
```

normal_vector

Normal vector of the given plane.

Examples

```
>>> from diofant import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.normal_vector
(-1, 2, -1)
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 4, 7))
>>> a.normal_vector
(1, 4, 7)
```

p1

The only defining point of the plane. Others can be obtained from the `arbitrary_point` method.

See also:

[*diofant.geometry.point.Point3D*](#) (page 445)

Examples

```
>>> from diofant import Point3D, Plane
>>> a = Plane(Point3D(1, 1, 1), Point3D(2, 3, 4), Point3D(2, 2, 2))
>>> a.p1
Point3D(1, 1, 1)
```

parallel_plane(*pt*)

Plane parallel to the given plane and passing through the point *pt*.

Parameters *pt*: **Point3D**

Returns Plane

Examples

```
>>> from diofant import Plane, Point3D
>>> a = Plane(Point3D(1, 4, 6), normal_vector=(2, 4, 6))
>>> a.parallel_plane(Point3D(2, 3, 5))
Plane(Point3D(2, 3, 5), (2, 4, 6))
```

perpendicular_line(*pt*)

A line perpendicular to the given plane.

Parameters *pt*: **Point3D**

Returns Line3D

Examples

```
>>> from diofant import Plane, Point3D, Line3D
>>> a = Plane(Point3D(1,4,6), normal_vector=(2, 4, 6))
>>> a.perpendicular_line(Point3D(9, 8, 7))
Line3D(Point3D(9, 8, 7), Point3D(11, 12, 13))
```

perpendicular_plane(pts*)**

Return a perpendicular passing through the given points. If the direction ratio between the points is the same as the Plane's normal vector then, to select from the infinite number of possible planes, a third point will be chosen on the z-axis (or the y-axis if the normal vector is already parallel to the z-axis). If less than two points are given they will be supplied as follows: if no point is given then *pt1* will be self.p1; if a second point is not given it will be a point through *pt1* on a line parallel to the z-axis (if the normal is not already the z-axis, otherwise on the line parallel to the y-axis).

Parameters *pts*: **0, 1 or 2 Point3D**

Returns Plane

Examples

```
>>> from diofant import Plane, Point3D, Line3D
>>> a, b = Point3D(0, 0, 0), Point3D(0, 1, 0)
>>> Z = (0, 0, 1)
>>> p = Plane(a, normal_vector=Z)
>>> p.perpendicular_plane(a, b)
Plane(Point3D(0, 0, 0), (1, 0, 0))
```

projection(*pt*)

Project the given point onto the plane along the plane normal.

Parameters **Point or Point3D**

Returns Point3D

Examples

```
>>> from diofant import Plane, Point, Point3D
>>> A = Plane(Point3D(1, 1, 2), normal_vector=(1, 1, 1))
```

The projection is along the normal vector direction, not the z axis, so (1, 1) does not project to (1, 1, 2) on the plane A:


```
>>> b = Point(1, 1)
>>> A.projection(b)
Point3D(5/3, 5/3, 2/3)
>>> _ in A
True
```

But the point (1, 1, 2) projects to (1, 1) on the XY-plane:

```
>>> XY = Plane((0, 0, 0), (0, 0, 1))
>>> XY.projection((1, 1, 2))
Point3D(1, 1, 0)
```

projection_line(*line*)

Project the given line onto the plane through the normal plane containing the line.

Parameters **LinearEntity** or **LinearEntity3D**

Returns Point3D, Line3D, Ray3D or Segment3D

Notes

For the interaction between 2D and 3D lines(segments, rays), you should convert the line to 3D by using this method. For example for finding the intersection between a 2D and a 3D line, convert the 2D line to a 3D line by projecting it on a required plane and then proceed to find the intersection between those lines.

Examples

```
>>> from diofant import Plane, Line, Line3D, Point, Point3D
>>> a = Plane(Point3D(1, 1, 1), normal_vector=(1, 1, 1))
>>> b = Line(Point(1, 1), Point(2, 2))
>>> a.projection_line(b)
Line3D(Point3D(4/3, 4/3, 1/3), Point3D(5/3, 5/3, -1/3))
>>> c = Line3D(Point3D(1, 1, 1), Point3D(2, 2, 2))
>>> a.projection_line(c)
Point3D(1, 1, 1)
```

random_point(*seed=None*)

Returns a random point on the Plane.

Returns Point3D

3.9 Symbolic Integrals

The integrals module in Diofant implements methods to calculate definite and indefinite integrals of expressions.

Principal method in this module is *integrate()* (page 546)

- `integrate(f, x)` returns the indefinite integral $\int f dx$
- `integrate(f, (x, a, b))` returns the definite integral $\int_a^b f dx$

3.9.1 Examples

Diofant can integrate a vast array of functions. It can integrate polynomial functions:

```
>>> from diofant import *
>>> init_printing(pretty_print=True, use_unicode=False, wrap_line=False, no_
    _global=True)
>>> x = Symbol('x')
>>> integrate(x**2 + x + 1, x)
      3      2
     x      x
    -- + -- + x
     3      2
```

Rational functions:

```
>>> integrate(x/(x**2+2*x+1), x)
      1
     log(x + 1) + ----
                x + 1
```

Exponential-polynomial functions. These multiplicative combinations of polynomials and the functions `exp`, `cos` and `sin` can be integrated by hand using repeated integration by parts, which is an extremely tedious process. Happily, Diofant will deal with these integrals.

```
>>> integrate(x**2 * exp(x) * cos(x), x)
      x  2      x  2      x      x
     E *x  *sin(x)  E *x  *cos(x)  - E *x*sin(x)  + E *x*cos(x)
     ----- + ----- - ----- + -----
              2          2          2          2
```

even a few nonelementary integrals (in particular, some integrals involving the error function) can be evaluated:

```
>>> integrate(exp(-x**2)*erf(x), x)
      2
     \sqrt{pi} *erf (x)
     -----
              4
```

3.9.2 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 $a < \text{Re}(s) < b$.

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 557) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). If `noconds=False`, then only F will be returned (i.e. not cond, and also not the strip (a,b)).

```

>>> from diofant.integrals.transforms import mellin_transform
>>> from diofant import exp
>>> from diofant.abc import x, s
>>> mellin_transform(exp(-x), x, s)
(gamma(s), (0, oo), True)

```

See also:

inverse_mellin_transform (page 527), *laplace_transform* (page 528), *fourier_transform* (page 528), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532)

`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 557) 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 557).

```

>>> from diofant.integrals.transforms import inverse_mellin_transform
>>> from diofant import oo, gamma
>>> from diofant.abc import x, 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 526), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532)

`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 558) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). If `noconds=True`, only F will be returned (i.e. not cond , and also not the plane a).

```
>>> from diofant.integrals import laplace_transform
>>> from diofant.abc import t, s, a
>>> laplace_transform(t**a, t, s)
(s**(-a)*gamma(a + 1)/s, 0, -re(a) < 1)
```

See also:

inverse_laplace_transform (page 528), *mellin_transform* (page 526), *fourier_transform* (page 528), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532)

`diofant.integrals.transforms.inverse_laplace_transform(F, s, t, plane=None, **hints)`

Compute the inverse Laplace transform of $F(s)$, defined as

$$f(t) = \int_{c-i\infty}^{c+i\infty} e^{st} F(s) ds,$$

for c so large that $F(s)$ has no singularities in the half-plane $\operatorname{Re}(s) > c - \epsilon$.

The plane can be specified by argument `plane`, but will be inferred if passed as `None`.

Under certain regularity conditions, this recovers $f(t)$ from its Laplace Transform $F(s)$, for non-negative t , and vice versa.

If the integral cannot be computed in closed form, this function returns an unevaluated *InverseLaplaceTransform* (page 558) 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 557).

```
>>> from diofant.integrals.transforms import inverse_laplace_transform
>>> from diofant import exp, Symbol
>>> from diofant.abc import s, t
>>> a = Symbol('a', positive=True)
>>> inverse_laplace_transform(exp(-a*s)/s, s, t)
Heaviside(-a + t)
```

See also:

laplace_transform (page 528), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532)

`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 558) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import fourier_transform, exp
>>> from diofant.abc import x, k
>>> 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 529), *sine_transform* (page 529), *inverse_sine_transform* (page 530), *cosine_transform* (page 530), *inverse_cosine_transform* (page 531), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532), *mellin_transform* (page 526), *laplace_transform* (page 528)

`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 558) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import inverse_fourier_transform, exp, sqrt, pi
>>> from diofant.abc import x, k
>>> 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 528), *sine_transform* (page 529), *inverse_sine_transform* (page 530), *cosine_transform* (page 530), *inverse_cosine_transform* (page 531), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532), *mellin_transform* (page 526), *laplace_transform* (page 528)

`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 558) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import sine_transform, exp
>>> from diofant.abc import x, k, a
>>> sine_transform(x*exp(-a*x**2), x, k)
sqrt(2)*E**(-k**2/(4*a))*k/(4*a**(3/2))
>>> sine_transform(x**(-a), x, k)
2**(-a + 1/2)*k**(a - 1)*gamma(-a/2 + 1)/gamma(a/2 + 1/2)
```

See also:

fourier_transform (page 528), *inverse_fourier_transform* (page 529), *inverse_sine_transform* (page 530), *cosine_transform* (page 530), *inverse_cosine_transform* (page 531), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532), *mellin_transform* (page 526), *laplace_transform* (page 528)

`diofant.integrals.transforms.inverse_sine_transform(F, k, x, **hints)`
 Compute the unitary, ordinary-frequency inverse sine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^\infty F(k) \sin(2\pi xk) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *InverseSineTransform* (page 558) object.

For a description of possible hints, refer to the docstring of *diofant.integrals.transforms.IntegralTransform.doit()* (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import inverse_sine_transform, exp, sqrt, gamma, pi
>>> from diofant.abc import x, k, a
>>> inverse_sine_transform(2**((1-2*a)/2)*k**(a - 1)*
...     gamma(-a/2 + 1)/gamma((a+1)/2), k, x)
x**(-a)
>>> inverse_sine_transform(sqrt(2)*k*exp(-k**2/(4*a))/(4*sqrt(a)**3), k, x)
E**(-a*x**2)*x
```

See also:

fourier_transform (page 528), *inverse_fourier_transform* (page 529), *sine_transform* (page 529), *cosine_transform* (page 530), *inverse_cosine_transform* (page 531), *hankel_transform* (page 531), *inverse_hankel_transform* (page 532), *mellin_transform* (page 526), *laplace_transform* (page 528)

`diofant.integrals.transforms.cosine_transform(f, x, k, **hints)`
 Compute the unitary, ordinary-frequency cosine transform of f , defined as

$$F(k) = \sqrt{\frac{2}{\pi}} \int_0^\infty f(x) \cos(2\pi xk) dx.$$

If the transform cannot be computed in closed form, this function returns an unevaluated *CosineTransform* (page 558) object.

For a description of possible hints, refer to the docstring of `diofant.integrals.transforms.IntegralTransform.doit()` (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import cosine_transform, exp, sqrt, cos
>>> from diofant.abc import x, k, a
>>> cosine_transform(exp(-a*x), x, k)
sqrt(2)*a/(sqrt(pi)*(a**2 + k**2))
>>> cosine_transform(exp(-a*sqrt(x))*cos(a*sqrt(x)), x, k)
E**(-a**2/(2*k))*a/(2*k**(3/2))
```

See also:

[fourier_transform](#) (page 528), [inverse_fourier_transform](#) (page 529), [sine_transform](#) (page 529), [inverse_sine_transform](#) (page 530), [inverse_cosine_transform](#) (page 531), [hankel_transform](#) (page 531), [inverse_hankel_transform](#) (page 532), [mellin_transform](#) (page 526), [laplace_transform](#) (page 528)

`diofant.integrals.transforms.inverse_cosine_transform(F, k, x, **hints)`

Compute the unitary, ordinary-frequency inverse cosine transform of F , defined as

$$f(x) = \sqrt{\frac{2}{\pi}} \int_0^{\infty} F(k) \cos(2\pi xk) dk.$$

If the transform cannot be computed in closed form, this function returns an unevaluated `InverseCosineTransform` (page 558) object.

For a description of possible hints, refer to the docstring of `diofant.integrals.transforms.IntegralTransform.doit()` (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import inverse_cosine_transform, exp, sqrt, pi
>>> from diofant.abc import x, k, a
>>> 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 528), [inverse_fourier_transform](#) (page 529), [sine_transform](#) (page 529), [inverse_sine_transform](#) (page 530), [cosine_transform](#) (page 530), [hankel_transform](#) (page 531), [inverse_hankel_transform](#) (page 532), [mellin_transform](#) (page 526), [laplace_transform](#) (page 528)

`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 558) object.

For a description of possible hints, refer to the docstring of `diofant.integrals.transforms.IntegralTransform.doit()` (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import hankel_transform, inverse_hankel_transform
>>> from diofant import gamma, exp, sinh, cosh
>>> from diofant.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
E**(-a*r)
```

See also:

[fourier_transform](#) (page 528), [inverse_fourier_transform](#) (page 529), [sine_transform](#) (page 529), [inverse_sine_transform](#) (page 530), [cosine_transform](#) (page 530), [inverse_cosine_transform](#) (page 531), [inverse_hankel_transform](#) (page 532), [mellin_transform](#) (page 526), [laplace_transform](#) (page 528)

`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 559) object.

For a description of possible hints, refer to the docstring of `diofant.integrals.transforms.IntegralTransform.doit()` (page 557). Note that for this transform, by default `noconds=True`.

```
>>> from diofant import hankel_transform, inverse_hankel_transform, gamma
>>> from diofant import gamma, exp, sinh, cosh
>>> from diofant.abc import r, k, m, nu, a
```

```
>>> ht = hankel_transform(1/r**m, r, k, nu)
>>> ht
2*2**(-m)*k**(m - 2)*gamma(-m/2 + nu/2 + 1)/gamma(m/2 + nu/2)
```

```
>>> inverse_hankel_transform(ht, k, r, nu)
r**(-m)
```

```
>>> ht = hankel_transform(exp(-a*r), r, k, 0)
>>> ht
a/(k**3*(a**2/k**2 + 1)**(3/2))
```

```
>>> inverse_hankel_transform(ht, k, r, 0)
E**(-a*r)
```


See also:

`fourier_transform` (page 528), `inverse_fourier_transform` (page 529), `sine_transform` (page 529), `inverse_sine_transform` (page 530), `cosine_transform` (page 530), `inverse_cosine_transform` (page 531), `hankel_transform` (page 531), `mellin_transform` (page 526), `laplace_transform` (page 528)

3.9.3 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:

Computing Integrals using Meijer G-Functions

This text aims to describe in some detail the steps (and subtleties) involved in using Meijer G-functions for computing definite and indefinite integrals. We shall ignore proofs completely.

Overview

The algorithm to compute $\int f(x)dx$ or $\int_0^\infty f(x)dx$ generally consists of three steps:

1. Rewrite the integrand using Meijer G-functions (one or sometimes two).
2. Apply an integration theorem, to get the answer (usually expressed as another G-function).
3. Expand the result in named special functions.

Step (3) is implemented in the function `hyperexpand` (q.v.). Steps (1) and (2) are described below. Moreover, G-functions are usually branched. Thus our treatment of branched functions is described first.

Some other integrals (e.g. $\int_{-\infty}^\infty$) can also be computed by first recasting them into one of the above forms. There is a lot of choice involved here, and the algorithm is heuristic at best.

Polar Numbers and Branched Functions

Both Meijer G-Functions and Hypergeometric functions are typically branched (possible branchpoints being $0, \pm 1, \infty$). This is not very important when e.g. expanding a single hypergeometric function into named special functions, since sorting out the branches can be left to the human user. However this algorithm manipulates and transforms G-functions, and to do this correctly it needs at least some crude understanding of the branchings involved.

To begin, we consider the set $\mathcal{S} = \{(r, \theta) : r > 0, \theta \in \mathbb{R}\}$. We have a map $p : \mathcal{S} \rightarrow \mathbb{C} - \{0\}, (r, \theta) \mapsto re^{i\theta}$. Decreeing this to be a local biholomorphism gives \mathcal{S} both a topology and a complex

structure. This Riemann Surface is usually referred to as the Riemann Surface of the logarithm, for the following reason: We can define maps $\text{Exp} : \mathbb{C} \rightarrow \mathcal{S}, (x + iy) \mapsto (\exp(x), y)$ and $\text{Log} : \mathcal{S} \rightarrow \mathbb{C}, (e^x, y) \mapsto x + iy$. These can both be shown to be holomorphic, and are indeed mutual inverses.

We also sometimes formally attach a point “zero” (0) to \mathcal{S} and denote the resulting object \mathcal{S}_0 . Notably there is no complex structure defined near 0. A fundamental system of neighbourhoods is given by $\{\text{Exp}(z) : \Re(z) < k\}$, which at least defines a topology. Elements of \mathcal{S}_0 shall be called polar numbers. We further define functions $\text{Arg} : \mathcal{S} \rightarrow \mathbb{R}, (r, \theta) \mapsto \theta$ and $|\cdot| : \mathcal{S}_0 \rightarrow \mathbb{R}_{>0}, (r, \theta) \mapsto r$. These have evident meaning and are both continuous everywhere.

Using these maps many operations can be extended from \mathbb{C} to \mathcal{S} . We define $\text{Exp}(a)\text{Exp}(b) = \text{Exp}(a + b)$ for $a, b \in \mathbb{C}$, also for $a \in \mathcal{S}$ and $b \in \mathbb{C}$ we define $a^b = \text{Exp}(b \text{Log}(a))$. It can be checked easily that using these definitions, many algebraic properties holding for positive reals (e.g. $(ab)^c = a^c b^c$) which hold in \mathbb{C} only for some numbers (because of branch cuts) hold indeed for all polar numbers.

As one peculiarity it should be mentioned that addition of polar numbers is not usually defined. However, formal sums of polar numbers can be used to express branching behaviour. For example, consider the functions $F(z) = \sqrt{1+z}$ and $G(a, b) = \sqrt{a+b}$, where a, b, z are polar numbers. The general rule is that functions of a single polar variable are defined in such a way that they are continuous on circles, and agree with the usual definition for positive reals. Thus if $S(z)$ denotes the standard branch of the square root function on \mathbb{C} , we are forced to define

$$F(z) = \begin{cases} S(p(z)) & : |z| < 1 \\ S(p(z)) & : -\pi < \text{Arg}(z) + 4\pi n \leq \pi \text{ for some } n \in \mathbb{Z} . \\ -S(p(z)) & : \text{else} \end{cases}$$

(We are omitting $|z| = 1$ here, this does not matter for integration.) Finally we define $G(a, b) = \sqrt{a}F(b/a)$.

Representing Branched Functions on the Argand Plane

Suppose $f : \mathcal{S} \rightarrow \mathbb{C}$ 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 called “lifting to the principal branch” throughout the Diofant documentation.

Table Lookups and Inverse Mellin Transforms

Suppose we are given an integrand $f(x)$ and are trying to rewrite it as a single G-function. To do this, we first split $f(x)$ into the form $x^s g(x)$ (where $g(x)$ is supposed to be simpler than $f(x)$). This is because multiplicative powers can be absorbed into the G-function later. This splitting is done by `_split_mul(f, x)`. Then we assemble a tuple of functions that occur in f (e.g. if $f(x) = e^x \cos x$, we would assemble the tuple `(cos, exp)`). This is done by the function `_mytype(f, x)`. Next we index a lookup table (created using `_create_lookup_table()`) with this tuple. This (hopefully) yields a list of Meijer G-function formulae involving these functions, we then pattern-match all of them. If one fits, we were successful, otherwise not and we have to try something else.

Suppose now we want to rewrite as a product of two G-functions. To do this, we (try to) find all inequivalent ways of splitting $f(x)$ into a product $f_1(x)f_2(x)$. We could try these splittings in any order, but it is often a good idea to minimise (a) the number of powers 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 `_rewritel` and `_rewrite2` respectively use above-mentioned helpers and `_rewrite_single` to rewrite their argument as respectively one or two G-functions.

Applying the Integral Theorems

If the integrand has been recast into G-functions, evaluating the integral is relatively easy. We first do some substitutions to reduce e.g. the exponent of the argument of the G-function to unity (see `_rewrite_saxena_1` and `_rewrite_saxena`, respectively, for one or two G-functions). Next we go through a list of conditions under which the integral theorem applies. It can fail for basically two reasons: either the integral does not exist, or the manipulations in deriving the theorem may not be allowed (for more details, see this [\[BlogPost\]](#) (page 1269)).

Sometimes this can be remedied by reducing the argument of the G-functions involved. For example it is clear that the G-function representing e^z satisfies $G(\text{Exp}(2\pi i)z) = G(z)$ for all $z \in \mathcal{S}$. The function `meijerg.get_period()` can be used to discover this, and the function `principal_branch(z,period)` in `functions/elementary/complexes.py` can be used to exploit the information. This is done transparently by the integration code.

The G-Function Integration Theorems

This section intends to display in detail the definite integration theorems used in the code. The following two formulae go back to Meijer (In fact he proved more general formulae; indeed in the literature formulae are usually stated in more general form. However it is very easy to deduce the general formulae from the ones we give here. It seemed best to keep the theorems as simple as possible, since they are very complicated anyway.):

1.

$$\int_0^\infty G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \eta x \right) dx = \frac{\prod_{j=1}^m \Gamma(b_j + 1) \prod_{j=1}^n \Gamma(-a_j)}{\eta \prod_{j=m+1}^q \Gamma(-b_j) \prod_{j=n+1}^p \Gamma(a_j + 1)}$$

2.

$$\int_0^\infty G_{u,v}^{s,t} \left(\begin{matrix} c_1, \dots, c_u \\ d_1, \dots, d_v \end{matrix} \middle| \sigma x \right) G_{p,q}^{m,n} \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| \omega x \right) dx = G_{v+p,u+q}^{m+t,n+s} \left(\begin{matrix} a_1, \dots, a_n, -d_1, \dots, -d_v, a_{n+1}, \dots, a_p \\ b_1, \dots, b_m, -c_1, \dots, -c_u, b_{m+1}, \dots, b_q \end{matrix} \middle| \frac{\omega}{\sigma} \right)$$

The more interesting question is under what conditions these formulae are valid. Below we detail the conditions implemented in Diofant. They are an amalgamation of conditions found in [Prudnikov1990] (page 1274) and [Luke1969] (page 1274); please let us know if you find any errors.

Conditions of Convergence for Integral (1)

We can without loss of generality assume $p \leq q$, since the G-functions of indices m, n, p, q and of indices n, m, q, p can be related easily (see e.g. [Luke1969] (page 1274), section 5.3). We introduce the following notation:

$$\begin{aligned}\xi &= m + n - p \\ \delta &= m + n - \frac{p + q}{2} \\ C_3 &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, m \\ &0 < -\Re(a_j) \text{ for } j = 1, \dots, n \\ C_3^* &: -\Re(b_j) < 1 \text{ for } j = 1, \dots, q \\ &0 < -\Re(a_j) \text{ for } j = 1, \dots, p \\ C_4 &: -\Re(\delta) + \frac{q + 1 - p}{2} > q - p\end{aligned}$$

The convergence conditions will be detailed in several “cases”, numbered one to five. For later use it will be helpful to separate conditions “at infinity” from conditions “at zero”. By conditions “at infinity” we mean conditions that only depend on the behaviour of the integrand for large, positive values of x , whereas by conditions “at zero” we mean conditions that only depend on the behaviour of the integrand on $(0, \epsilon)$ for any $\epsilon > 0$. Since all our conditions are specified in terms of parameters of the G-functions, this distinction is not immediately visible. They are, however, of very distinct character mathematically; the conditions at infinity being in particular much harder to control.

In order for the integral theorem to be valid, conditions n “at zero” and “at infinity” both have to be fulfilled, for some n .

These are the conditions “at infinity”:

1.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi \wedge (A \vee B \vee C),$$

where

$$A = 1 \leq n \wedge p < q \wedge 1 \leq m$$

$$B = 1 \leq p \wedge 1 \leq m \wedge q = p + 1 \wedge \neg(n = 0 \wedge m = p + 1)$$

$$C = 1 \leq n \wedge q = p \wedge |\arg(\eta)| \neq (\delta - 2k)\pi \text{ for } k = 0, 1, \dots, \left\lceil \frac{\delta}{2} \right\rceil.$$

2.

$$n = 0 \wedge p + 1 \leq m \wedge |\arg(\eta)| < \delta\pi$$

3.

$$(p < q \wedge 1 \leq m \wedge \delta > 0 \wedge |\arg(\eta)| = \delta\pi) \vee (p \leq q - 2 \wedge \delta = 0 \wedge \arg(\eta) = 0)$$

4.

$$p = q \wedge \delta = 0 \wedge \arg(\eta) = 0 \wedge \eta \neq 0 \wedge \Re \left(\sum_{j=1}^p b_j - a_j \right) < 0$$

5.

$$\delta > 0 \wedge |\arg(\eta)| < \delta\pi$$

And these are the conditions "at zero":

1.

$$\eta \neq 0 \wedge C_3$$

2.

$$C_3$$

3.

$$C_3 \wedge C_4$$

4.

$$C_3$$

5.

$$C_3$$

Conditions of Convergence for Integral (2)

We introduce the following notation:

$$b^* = s + t - \frac{u + v}{2}$$

$$c^* = m + n - \frac{p + q}{2}$$

$$\rho = \sum_{j=1}^v d_j - \sum_{j=1}^u c_j + \frac{u - v}{2} + 1$$

$$\mu = \sum_{j=1}^q b_j - \sum_{j=1}^p a_j + \frac{p - q}{2} + 1$$

$$\phi = q - p - \frac{u - v}{2} + 1$$

$$\eta = 1 - (v - u) - \mu - \rho$$

$$\psi = \frac{\pi(q - m - n) + |\arg(\omega)|}{q - p}$$

$$\theta = \frac{\pi(v - s - t) + |\arg(\sigma)|}{v - u}$$

$$\lambda_c = (q - p)|\omega|^{1/(q-p)} \cos \psi + (v - u)|\sigma|^{1/(v-u)} \cos \theta$$

$$\lambda_{s0}(c_1, c_2) = c_1(q - p)|\omega|^{1/(q-p)} \sin \psi + c_2(v - u)|\sigma|^{1/(v-u)} \sin \theta$$

$$\lambda_s = \begin{cases} \lambda_{s0}(-1, -1) \lambda_{s0}(1, 1) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), -1) \lambda_{s0}(\text{sign}(\arg(\omega)), 1) & \text{for } \arg(\omega) \neq 0 \wedge \arg(\sigma) = 0 \\ \lambda_{s0}(-1, \text{sign}(\arg(\sigma))) \lambda_{s0}(1, \text{sign}(\arg(\sigma))) & \text{for } \arg(\omega) = 0 \wedge \arg(\sigma) \neq 0 \\ \lambda_{s0}(\text{sign}(\arg(\omega)), \text{sign}(\arg(\sigma))) & \text{otherwise} \end{cases}$$

$$z_0 = \frac{\omega}{\sigma} e^{-i\pi(b^* + c^*)}$$

$$z_1 = \frac{\sigma}{\omega} e^{-i\pi(b^* + c^*)}$$

The following conditions will be helpful:

$$C_1 : (a_i - b_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, n, j = 1, \dots, m) \\ \wedge (c_i - d_j \notin \mathbb{Z}_{>0} \text{ for } i = 1, \dots, t, j = 1, \dots, s)$$

$$C_2 : \Re(1 + b_i + d_j) > 0 \text{ for } i = 1, \dots, m, j = 1, \dots, s$$

$$C_3 : \Re(a_i + c_j) < 1 \text{ for } i = 1, \dots, n, j = 1, \dots, t$$

$$C_4 : (p - q)\Re(c_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, t$$

$$C_5 : (p - q)\Re(1 + d_i) - \Re(\mu) > -\frac{3}{2} \text{ for } i = 1, \dots, s$$

$$C_6 : (u - v)\Re(a_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, n$$

$$C_7 : (u - v)\Re(1 + b_i) - \Re(\rho) > -\frac{3}{2} \text{ for } i = 1, \dots, m$$

$$C_8 : 0 < |\phi| + 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_9 : 0 < |\phi| - 2\Re((\mu - 1)(-u + v) + (-p + q)(\rho - 1) + (-p + q)(-u + v))$$

$$C_{10} : |\arg(\sigma)| < \pi b^*$$

$$C_{11} : |\arg(\sigma)| = \pi b^*$$

$$C_{12} : |\arg(\omega)| < c^* \pi$$

$$C_{13} : |\arg(\omega)| = c^* \pi$$

$$C_{14}^1 : (z_0 \neq 1 \wedge |\arg(1 - z_0)| < \pi) \vee (z_0 = 1 \wedge \Re(\mu + \rho - u + v) < 1)$$

$$C_{14}^2 : (z_1 \neq 1 \wedge |\arg(1 - z_1)| < \pi) \vee (z_1 = 1 \wedge \Re(\mu + \rho - p + q) < 1)$$

$$C_{14} : \phi = 0 \wedge b^* + c^* \leq 1 \wedge (C_{14}^1 \vee C_{14}^2)$$

$$C_{15} : \lambda_c > 0 \vee (\lambda_c = 0 \wedge \lambda_s \neq 0 \wedge \Re(\eta) > -1) \vee (\lambda_c = 0 \wedge \lambda_s = 0 \wedge \Re(\eta) > 0)$$

$$C_{16} : \int_0^\infty G_{u,v}^{s,t}(\sigma x) dx \text{ converges at infinity}$$

$$C_{17} : \int_0^\infty G_{p,q}^{m,n}(\omega x) dx \text{ converges at infinity}$$

Note that C_{16} and C_{17} are the reason we split the convergence conditions for integral (1).

With this notation established, the implemented convergence conditions can be enumerated as follows:

1.

$$mnst \neq 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

2.

$$u = v \wedge b^* = 0 \wedge 0 < c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{12}$$

3.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re \mu < 1 \wedge \Re \rho < 1 \wedge \sigma \neq \omega \wedge C_1 \wedge C_2 \wedge C_3$$

4.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

5.

$$p = q \wedge u = v \wedge b^* = 0 \wedge c^* = 0 \wedge 0 < \sigma \wedge 0 < \omega \wedge \Re(\mu + \rho) < 1 \wedge \omega \neq \sigma \wedge C_1 \wedge C_2 \wedge C_3$$

6.

$$q < p \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{10} \wedge C_{13}$$

7.

$$p < q \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{10} \wedge C_{13}$$

8.

$$v < u \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11} \wedge C_{12}$$

9.

$$u < v \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11} \wedge C_{12}$$

10.

$$q < p \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_{13}$$

11.

$$p < q \wedge u = v \wedge b^* = 0 \wedge 0 \leq c^* \wedge 0 < \sigma \wedge \Re \rho < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_{13}$$

12.

$$p = q \wedge v < u \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_7 \wedge C_{11}$$

13.

$$p = q \wedge u < v \wedge 0 \leq b^* \wedge c^* = 0 \wedge 0 < \omega \wedge \Re \mu < 1 \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_6 \wedge C_{11}$$

14.

$$p < q \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_7 \wedge C_{11} \wedge C_{13}$$

15.

$$q < p \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_6 \wedge C_{11} \wedge C_{13}$$

16.

$$q < p \wedge v < u \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_5 \wedge C_7 \wedge C_8 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

17.

$$p < q \wedge u < v \wedge 0 \leq b^* \wedge 0 \leq c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_6 \wedge C_9 \wedge C_{11} \wedge C_{13} \wedge C_{14}$$

18.

$$t = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 < \phi \wedge C_1 \wedge C_2 \wedge C_{10}$$

19.

$$s = 0 \wedge 0 < t \wedge 0 < b^* \wedge \phi < 0 \wedge C_1 \wedge C_3 \wedge C_{10}$$

20.

$$n = 0 \wedge 0 < m \wedge 0 < c^* \wedge \phi < 0 \wedge C_1 \wedge C_2 \wedge C_{12}$$

21.

$$m = 0 \wedge 0 < n \wedge 0 < c^* \wedge 0 < \phi \wedge C_1 \wedge C_3 \wedge C_{12}$$

22.

$$st = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

23.

$$mn = 0 \wedge 0 < b^* \wedge 0 < c^* \wedge C_1 \wedge C_2 \wedge C_3 \wedge C_{10} \wedge C_{12}$$

24.

$$p < m + n \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

25.

$$q < m + n \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge c^* < 0 \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

26.

$$p = q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

27.

$$p = q + 1 \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

28.

$$p < q - 1 \wedge t = 0 \wedge \phi = 0 \wedge 0 < s \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - p + 1) \wedge C_1 \wedge C_2 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

29.

$$q + 1 < p \wedge s = 0 \wedge \phi = 0 \wedge 0 < t \wedge 0 < b^* \wedge 0 \leq c^* \wedge \pi c^* < |\arg(\omega)| \wedge |\arg(\omega)| < \pi(m + n - q + 1) \wedge C_1 \wedge C_3 \wedge C_{10} \wedge C_{14} \wedge C_{15}$$

30.

$$n = 0 \wedge \phi = 0 \wedge 0 < s + t \wedge 0 < m \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

31.

$$m = 0 \wedge \phi = 0 \wedge v < s + t \wedge 0 < n \wedge 0 < c^* \wedge b^* < 0 \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

32.

$$n = 0 \wedge \phi = 0 \wedge u = v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

33.

$$m = 0 \wedge \phi = 0 \wedge u = v + 1 \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(b^* + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

34.

$$n = 0 \wedge \phi = 0 \wedge u < v - 1 \wedge 0 < m \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - u + 1) \wedge C_1 \wedge C_2 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

35.

$$m = 0 \wedge \phi = 0 \wedge v + 1 < u \wedge 0 < n \wedge 0 < c^* \wedge 0 \leq b^* \wedge \pi b^* < |\arg(\sigma)| \wedge |\arg(\sigma)| < \pi(s + t - v + 1) \wedge C_1 \wedge C_3 \wedge C_{12} \wedge C_{14} \wedge C_{15}$$

36.

$$C_{17} \wedge t = 0 \wedge u < s \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

37.

$$C_{17} \wedge s = 0 \wedge v < t \wedge 0 < b^* \wedge C_{10} \wedge C_1 \wedge C_2 \wedge C_3$$

38.

$$C_{16} \wedge n = 0 \wedge p < m \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

39.

$$C_{16} \wedge m = 0 \wedge q < n \wedge 0 < c^* \wedge C_{12} \wedge C_1 \wedge C_2 \wedge C_3$$

The Inverse Laplace Transform of a G-function

The inverse laplace transform of a Meijer G-function can be expressed as another G-function. This is a fairly versatile method for computing this transform. However, I could not find the details in the literature, so I work them out here. In [Luke1969] (page 1274), section 5.6.3, there is a formula for the inverse Laplace transform of a G-function of argument bz , and convergence conditions are also given. However, we need a formula for argument bz^a for rational a .

We are asked to compute

$$f(t) = \frac{1}{2\pi i} \int_{c-i\infty}^{c+i\infty} e^{zt} G(bz^a) dz,$$

for positive real t . Three questions arise:

1. When does this integral converge?
2. How can we compute the integral?
3. When is our computation valid?

How to compute the integral

We shall work formally for now. Denote by $\Delta(s)$ the product of gamma functions appearing in the definition of G , so that

$$G(z) = \frac{1}{2\pi i} \int_L \Delta(s) z^s ds.$$

Thus

$$f(t) = \frac{1}{(2\pi i)^2} \int_{c-i\infty}^{c+i\infty} \int_L e^{zt} \Delta(s) b^s z^{as} ds dz.$$

We interchange the order of integration to get

$$f(t) = \frac{1}{2\pi i} \int_L b^s \Delta(s) \int_{c-i\infty}^{c+i\infty} e^{zt} z^{as} \frac{dz}{2\pi i} ds.$$

The inner integral is easily seen to be $\frac{1}{\Gamma(-as)} \frac{1}{t^{1+as}}$. (Using Cauchy's theorem and Jordan's lemma deform the contour to run from $-\infty$ to $-\infty$, encircling 0 once in the negative sense. For as real and greater than one, this contour can be pushed onto the negative real axis and the integral is recognised as a product of a sine and a gamma function. The formula is then proved using the functional equation of the gamma function, and extended to the entire domain of convergence of the original integral by appealing to analytic continuation.) Hence we find

$$f(t) = \frac{1}{t} \frac{1}{2\pi i} \int_L \Delta(s) \frac{1}{\Gamma(-as)} \left(\frac{b}{t^a}\right)^s ds,$$

which is a so-called Fox H function (of argument $\frac{b}{t^a}$). For rational a , this can be expressed as a Meijer G-function using the gamma function multiplication theorem.

When this computation is valid

There are a number of obstacles in this computation. Interchange of integrals is only valid if all integrals involved are absolutely convergent. In particular the inner integral has to converge. Also, for our identification of the final integral as a Fox H / Meijer G-function to be correct, the poles of the newly obtained gamma function must be separated properly.

It is easy to check that the inner integral converges absolutely for $\Re(as) < -1$. Thus the contour L has to run left of the line $\Re(as) = -1$. Under this condition, the poles of the newly-introduced gamma function are separated properly.

It remains to observe that the Meijer G-function is an analytic, unbranched function of its parameters, and of the coefficient b . Hence so is $f(t)$. Thus the final computation remains valid as long as the initial integral converges, and if there exists a changed set of parameters where the computation is valid. If we assume w.l.o.g. that $a > 0$, then the latter condition is fulfilled if G converges along contours (2) or (3) of [Luke1969] (page 1274), section 5.2, i.e. either $\delta \geq \frac{a}{2}$ or $p \geq 1, p \geq q$.

When the integral exists

Using [Luke1969] (page 1274), 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 $\omega_{\pm} = b e^{\pm i\Re(c) \frac{\pi}{2}}$. We have three cases:

1. $b = 0$ or $\Re(c) \leq 0$. In this case the integral converges if $\Re(a) \leq -1$.
2. $b \neq 0, \Im(c) = 0, \Re(c) > 0$. In this case the integral converges if $\Re(\omega_{\pm}) < 0$.
3. $b \neq 0, \Im(c) = 0, \Re(c) > 0, \Re(\omega_{\pm}) \leq 0$, and at least one of $\Re(\omega_{\pm}) = 0$. Here the same condition as in (1) applies.

Implemented G-Function Formulae

An important part of the algorithm is a table expressing various functions as Meijer G-functions. This is essentially a table of Mellin Transforms in disguise. The following automatically generated table shows the formulae currently implemented in 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:

$$a = aG_{1,1}^{1,0} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right) + aG_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right)$$

$$(b + pz^q)^{-a} = \frac{b^{-a}}{\Gamma(a)} G_{1,1}^{1,1} \left(\begin{matrix} -a + 1 \\ 0 \end{matrix} \middle| \frac{pz^q}{b} \right)$$

$$\begin{aligned} \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) \\ (\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q})^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) \\ (-\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q})^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{(\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q})^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{(-\sqrt{pz^{\frac{q}{2}}} + \sqrt{a + pz^q})^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(\begin{matrix} \\ 0 \end{matrix} \middle| pz^q \exp_{\text{polar}}(-i\pi) \right) \end{aligned}$$

Functions involving $|b - pz^q|$:

$$|b - pz^q|^{-a} = \frac{\pi |b|^{-a}}{\cos\left(\frac{\pi a}{2}\right)\Gamma(a)} G_{2,2}^{1,1} \left(\begin{matrix} -a + 1, -\frac{a}{2} + \frac{1}{2} \\ 0, -\frac{a}{2} + \frac{1}{2} \end{matrix} \middle| \frac{pz^q}{b} \right), \text{ if } \Re a < 1$$

Functions involving $\text{Chi}(pz^q)$:

$$\text{Chi}(pz^q) = -\frac{\pi^{\frac{3}{2}}}{2} G_{2,4}^{2,0} \left(\begin{matrix} \frac{1}{2}, 1 \\ \frac{1}{2}, \frac{1}{2} \end{matrix} \middle| \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\text{Ci}(pz^q)$:

$$\text{Ci}(pz^q) = -\frac{\sqrt{\pi}}{2} G_{1,3}^{2,0} \left(\begin{matrix} \\ 0, 0 \end{matrix} \middle| \frac{1}{2} \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\text{Ei}(pz^q)$:

$$\text{Ei}(pz^q) = -i\pi G_{1,1}^{1,0} \left(\begin{matrix} \\ 0 \end{matrix} \middle| z \right) - G_{1,2}^{2,0} \left(\begin{matrix} \\ 0, 0 \end{matrix} \middle| pz^q \exp_{\text{polar}}(i\pi) \right) - i\pi G_{1,1}^{0,1} \left(\begin{matrix} 1 \\ 0 \end{matrix} \middle| z \right)$$

Functions involving $\theta(-b + pz^q)$:

$$(-b + pz^q)^{a-1} \theta(-b + pz^q) = b^{a-1} G_{1,1}^{0,1} \left(\begin{matrix} a \\ 0 \end{matrix} \middle| \frac{pz^q}{b} \right) \Gamma(a), \text{ if } b > 0$$

$$(b - pz^q)^{a-1} \theta(b - pz^q) = b^{a-1} G_{1,1}^{1,0} \left(0 \left| \frac{pz^q}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

$$(-b + pz^q)^{a-1} \theta \left(z - \left(\frac{b}{p} \right)^{\frac{1}{q}} \right) = b^{a-1} G_{1,1}^{0,1} \left(a \left| \frac{pz^q}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

$$(b - pz^q)^{a-1} \theta \left(-z + \left(\frac{b}{p} \right)^{\frac{1}{q}} \right) = b^{a-1} G_{1,1}^{1,0} \left(0 \left| \frac{pz^q}{b} \right. \right) \Gamma(a), \text{ if } b > 0$$

Functions involving Shi (pz^q):

$$\text{Shi}(pz^q) = \frac{\sqrt{\pi} p}{4} z^q G_{1,3}^{1,1} \left(\frac{1}{2} \left| \frac{p^2}{4} z^{2q} \exp_{\text{polar}}(i\pi) \right. \right)$$

Functions involving Si (pz^q):

$$\text{Si}(pz^q) = \frac{\sqrt{\pi}}{2} G_{1,3}^{1,1} \left(\frac{1}{2} \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving I_a (pz^q):

$$I_a(pz^q) = \pi G_{1,3}^{1,0} \left(\frac{a}{2} \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving J_a (pz^q):

$$J_a(pz^q) = G_{0,2}^{1,0} \left(\frac{a}{2} \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving K_a (pz^q):

$$K_a(pz^q) = \frac{1}{2} G_{0,2}^{2,0} \left(\frac{a}{2}, -\frac{a}{2} \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving Y_a (pz^q):

$$Y_a(pz^q) = G_{1,3}^{2,0} \left(\frac{a}{2}, -\frac{a}{2} \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving \cos (pz^q):

$$\cos(pz^q) = \sqrt{\pi} G_{0,2}^{1,0} \left(0 \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving \cosh (pz^q):

$$\cosh(pz^q) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(0 \left| \frac{p^2}{4} z^{2q} \right. \right)$$

Functions involving E (pz^q):

$$E(pz^q) = -\frac{1}{4} G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{3}{2} \left| -pz^q \right. \right)$$

Functions involving K (pz^q):

$$K(pz^q) = \frac{1}{2} G_{2,2}^{1,2} \left(\frac{1}{2}, \frac{1}{2} \left| -pz^q \right. \right)$$

Functions involving $\operatorname{erf}(pz^q)$:

$$\operatorname{erf}(pz^q) = \frac{1}{\sqrt{\pi}} G_{1,2}^{1,1} \left(\frac{1}{2} \mid 0 \mid p^2 z^{2q} \right)$$

Functions involving $\operatorname{erfc}(pz^q)$:

$$\operatorname{erfc}(pz^q) = \frac{1}{\sqrt{\pi}} G_{1,2}^{2,0} \left(0, \frac{1}{2} \mid 1 \mid p^2 z^{2q} \right)$$

Functions involving $\operatorname{erfi}(pz^q)$:

$$\operatorname{erfi}(pz^q) = \frac{pz^q}{\sqrt{\pi}} G_{1,2}^{1,1} \left(\frac{1}{2} \mid 0 \mid -\frac{1}{2} \mid -p^2 z^{2q} \right)$$

Functions involving $E_a(pz^q)$:

$$E_a(pz^q) = G_{1,2}^{2,0} \left(a-1, 0 \mid a \mid pz^q \right)$$

Functions involving $C(pz^q)$:

$$C(pz^q) = \frac{1}{2} G_{1,3}^{1,1} \left(\frac{1}{4} \mid 0, \frac{3}{4} \mid \frac{\pi^2 p^4}{16} z^{4q} \right)$$

Functions involving $S(pz^q)$:

$$S(pz^q) = \frac{1}{2} G_{1,3}^{1,1} \left(\frac{3}{4} \mid 0, \frac{1}{4} \mid \frac{\pi^2 p^4}{16} z^{4q} \right)$$

Functions involving $\log(pz^q)$:

$$\log^n(pz^q) = \text{generated}$$

$$\begin{aligned} \log(a + pz^q) &= G_{1,1}^{1,0} \left(0 \mid 1 \mid z \right) \log(a) + G_{1,1}^{0,1} \left(1 \mid 0 \mid z \right) \log(a) + G_{2,2}^{1,2} \left(\frac{1}{1}, \frac{1}{0} \mid \frac{pz^q}{a} \right) \\ \log(|a - pz^q|) &= G_{1,1}^{1,0} \left(0 \mid 1 \mid z \right) \log(|a|) + G_{1,1}^{0,1} \left(1 \mid 0 \mid z \right) \log(|a|) + \pi G_{3,3}^{1,2} \left(\frac{1}{1}, \frac{1}{0}, \frac{1}{\frac{1}{2}} \mid \frac{pz^q}{a} \right) \end{aligned}$$

Functions involving $\sin(pz^q)$:

$$\sin(pz^q) = \sqrt{\pi} G_{0,2}^{1,0} \left(\frac{1}{2} \mid 0 \mid \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\sinh(pz^q)$:

$$\sinh(pz^q) = \pi^{\frac{3}{2}} G_{1,3}^{1,0} \left(\frac{1}{2} \mid 1, 0 \mid \frac{p^2}{4} z^{2q} \right)$$

Functions involving $\theta(-pz^q + 1)$, $\log(pz^q)$:

$$\log^n(pz^q) \theta(-pz^q + 1) = \text{generated}$$

$$\log^n(pz^q) \theta(pz^q - 1) = \text{generated}$$

3.9.4 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`.

See also:

[`diofant.integrals.integrals.Integral`](#) (page 553), [`diofant.integrals.integrals.Integral.doit`](#) (page 554)

Examples

```
>>> from diofant import integrate, log, exp, oo
>>> from diofant.abc import a, x, y
```

```
>>> integrate(x*y, x)
x**2*y/2
```

```
>>> integrate(log(x), x)
x*log(x) - x
```

```
>>> integrate(log(x), (x, 1, a))
a*log(a) - a + 1
```

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

Terms that are independent of x are dropped by indefinite integration:

```
>>> from diofant import sqrt
>>> integrate(sqrt(1 + x), (x, 0, x))
2*(x + 1)**(3/2)/3 - 2/3
>>> integrate(sqrt(1 + x), x)
2*(x + 1)**(3/2)/3
```

```
>>> integrate(x*y)
Traceback (most recent call last):
...
ValueError: specify integration variables to integrate x*y
```


Note that `integrate(x)` syntax is meant only for convenience in interactive sessions and should be avoided in library code.

```
>>> integrate(x**a*exp(-x), (x, 0, oo)) # same as conds='piecewise'
Piecewise((gamma(a + 1), -re(a) < 1),
          (Integral(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)
```

`diofant.integrals.integrals.line_integrate(field, Curve, variables)`
Compute the line integral.

See also:

[diofant.integrals.integrals.integrate](#) (page 546), [diofant.integrals.integrals.Integral](#) (page 553)

Examples

```
>>> from diofant import Curve, line_integrate, E, ln
>>> from diofant.abc import x, y, t
>>> C = Curve([E**t + 1, E**t - 1], (t, 0, ln(2)))
>>> line_integrate(x + y, C, [x, y])
3*sqrt(2)
```

`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.

See also:

[diofant.functions.special.delta_functions.DiracDelta](#) (page 357), [diofant.integrals.integrals.Integral](#) (page 553)

Examples

```
>>> from diofant.abc import x, y, z
>>> from diofant.integrals.deltafunctions import deltaintegrate
>>> from diofant import sin, cos, DiracDelta, Heaviside
>>> deltaintegrate(x*sin(x)*cos(x)*DiracDelta(x - 1), x)
sin(1)*cos(1)*Heaviside(x - 1)
>>> deltaintegrate(y**2*DiracDelta(x - z)*DiracDelta(y - z), y)
z**2*DiracDelta(x - z)*Heaviside(y - z)
```

`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 $f = g'$.

```
>>> from diofant.integrals.rationaltools import ratint
>>> from diofant.abc import x
```

```
>>> ratint(36/(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2), x)
(12*x + 6)/(x**2 - 1) + 4*log(x - 2) - 4*log(x + 1)
```

See also:

[diofant.integrals.integrals.Integral.doit](#) (page 554), [diofant.integrals.rationaltools.ratint_logpart](#) (page 550), [diofant.integrals.rationaltools.ratint_ratpart](#) (page 551)

References

[Bro05] (page 1269)

`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:

$$\frac{d}{dx} \frac{f}{g} = \sum_{i=1..n} \frac{d}{dx} \left(\frac{s_i}{q_i} \right) + \sum_{i=1..n} \frac{s_i}{q_i} \log(s_i(a, x))$$

$i=1..n \quad a \mid q_i(a) = 0$

See also:

[diofant.integrals.rationaltools.ratint](#) (page 550), [diofant.integrals.rationaltools.ratint_ratpart](#) (page 551)

Examples

```

>>> from diofant.integrals.rationaltools import ratint_logpart
>>> from diofant.abc import x
>>> from diofant import Poly
>>> ratint_logpart(Poly(1, x, domain='ZZ'),
... Poly(x**2 + x + 1, x, domain='ZZ'), x)
[(Poly(x + 3*_t/2 + 1/2, x, domain='QQ[_t]'),
... Poly(3*_t**2 + 1, _t, domain='ZZ'))]
>>> ratint_logpart(Poly(12, x, domain='ZZ'),
... Poly(x**2 - x - 2, x, domain='ZZ'), x)
[(Poly(x - 3*_t/8 - 1/2, x, domain='QQ[_t]'),
... Poly(-_t**2 + 16, _t, domain='ZZ'))]
    
```

`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.

See also:

[diofant.integrals.rationaltools.ratint](#) (page 550), [diofant.integrals.rationaltools.ratint_logpart](#) (page 550)

Examples

```

>>> from diofant.integrals.rationaltools import ratint_ratpart
>>> from diofant.abc import x, y
>>> from diofant import Poly
>>> ratint_ratpart(Poly(1, x, domain='ZZ'),
... Poly(x + 1, x, domain='ZZ'), x)
(0, 1/(x + 1))
>>> ratint_ratpart(Poly(1, x, domain='EX'),
... Poly(x**2 + y**2, x, domain='EX'), x)
(0, 1/(x**2 + y**2))
>>> ratint_ratpart(Poly(36, x, domain='ZZ'),
... Poly(x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2, x, domain='ZZ'), x)
((12*x + 6)/(x**2 - 1), 12/(x**2 - x - 2))
    
```

`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 with minimal, positive exponents.

```

>>> from diofant import cos, sin
>>> from diofant.abc import x, y
>>> from diofant.integrals.heurisch import components
    
```

```

>>> components(sin(x)*cos(x)**2, x) == {x, sin(x), cos(x)}
True
    
```

See also:

[diofant.integrals.heurisch.heurisch](#) (page 551)

```
diofant.integrals.heurisch.heurisch(f, x, rewrite=False, hints=None, map-  
pings=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" [R343] (page 1269).

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 `heurisch(f, x, rewrite=False, hints=None)`

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.

See also:

[diofant.integrals.integrals.Integral.doit](#) (page 554),
[diofant.integrals.integrals.Integral](#) (page 553), [dio-](#)
[fant.integrals.heurisch.components](#) (page 551)

References

[R343] (page 1269), [R344] (page 1269), [R345] (page 1269), [R346] (page 1269), [R347] (page 1269)

Examples

```
>>> from diofant import tan  
>>> from diofant.integrals.heurisch import heurisch  
>>> from diofant.abc import x, y
```

```
>>> heurisch(y*tan(x), x)  
y*log(tan(x)**2 + 1)/2
```

```

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.

See also:

[diofant.integrals.heurisch.heurisch](#) (page 551)

Examples

```

>>> from diofant.core import symbols
>>> from diofant.functions import cos
>>> from diofant.integrals.heurisch import heurisch, heurisch_wrapper
>>> n, x = symbols('n x')
>>> heurisch(cos(n*x), x)
sin(n*x)/n
>>> heurisch_wrapper(cos(n*x), x)
Piecewise((x, Eq(n, 0)), (sin(n*x)/n, true))

```

`diofant.integrals.trigonometry.trigintegrate(f, x, conds='piecewise')`
 Integrate $f = \text{Mul}(\text{trig})$ over x

```

>>> from diofant import Symbol, sin, cos, tan, sec, csc, cot
>>> from diofant.integrals.trigonometry import trigintegrate
>>> from diofant.abc import x

```

```

>>> trigintegrate(sin(x)*cos(x), x)
sin(x)**2/2

```

```

>>> trigintegrate(sin(x)**2, x)
x/2 - sin(x)*cos(x)/2

```

```

>>> trigintegrate(tan(x)*sec(x), x)
1/cos(x)

```

```

>>> trigintegrate(sin(x)*tan(x), x)
-log(sin(x) - 1)/2 + log(sin(x) + 1)/2 - sin(x)

```

http://en.wikibooks.org/wiki/Calculus/Integration_techniques

See also:

[diofant.integrals.integrals.Integral.doit](#) (page 554), [diofant.integrals.integrals.Integral](#) (page 553)

The class `Integral` represents an unevaluated integral and has some methods that help in the integration of an expression.

class `diofant.integrals.integrals.Integral`
 Represents unevaluated integral.

is_commutative

Returns whether all the free symbols in the integral are commutative.

as_sum(*n*, *method*='midpoint')

Approximates the definite integral by a sum.

method ... one of: left, right, midpoint, trapezoid

These are all basically the rectangle method [1], the only difference is where the function value is taken in each interval to define the rectangle.

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

See also:

diofant.integrals.integrals.Integral.doit (page 554) Perform the integration using any hints

Examples

```
>>> from diofant import sin, sqrt
>>> from diofant.abc import x
>>> from diofant.integrals import Integral
>>> e = Integral(sin(x), (x, 3, 7))
>>> e
Integral(sin(x), (x, 3, 7))
```

For demonstration purposes, this interval will only be split into 2 regions, bounded by [3, 5] and [5, 7].

The left-hand rule uses function evaluations at the left of each interval:

```
>>> e.as_sum(2, 'left')
2*sin(5) + 2*sin(3)
```

The midpoint rule uses evaluations at the center of each interval:

```
>>> e.as_sum(2, 'midpoint')
2*sin(4) + 2*sin(6)
```

The right-hand rule uses function evaluations at the right of each interval:

```
>>> e.as_sum(2, 'right')
2*sin(5) + 2*sin(7)
```

The trapezoid rule uses function evaluations on both sides of the intervals. This is equivalent to taking the average of the left and right hand rule results:

```
>>> e.as_sum(2, 'trapezoid')
2*sin(5) + sin(3) + sin(7)
>>> (e.as_sum(2, 'left') + e.as_sum(2, 'right'))/2 == _
True
```

All but the trapexoid method may be used when dealing with a function with a discontinuity. Here, the discontinuity at $x = 0$ can be avoided by using the midpoint or right-hand method:

```
>>> e = Integral(1/sqrt(x), (x, 0, 1))
>>> e.as_sum(5).n(4)
1.730
>>> e.as_sum(10).n(4)
1.809
>>> e.doit().n(4) # the actual value is 2
2.000
```

The left- or trapezoid method will encounter the discontinuity and return oo:

```
>>> e.as_sum(5, 'left')
oo
>>> e.as_sum(5, 'trapezoid')
oo
```

doit(hints)**

Perform the integration using any hints given.

See also:

[diofant.integrals.trigonometry.trigintegrate](#) (page 553),
[diofant.integrals.heurisch.heurisch](#) (page 551), [diofant.integrals.rationaltools.ratint](#) (page 550)

[diofant.integrals.integrals.Integral.as_sum](#) (page 553) Approximate the integral using a sum

Examples

```
>>> from diofant import Integral
>>> from diofant.abc import x, i
>>> Integral(x**i, (i, 1, 3)).doit()
Piecewise((2, Eq(log(x), 0)), (x**3/log(x) - x/log(x), true))
```

free_symbols

This method returns the symbols that will exist when the integral is evaluated. This is useful if one is trying to determine whether an integral depends on a certain symbol or not.

See also:

[diofant.concrete.expr_with_limits.ExprWithLimits.function](#) (page 289),
[diofant.concrete.expr_with_limits.ExprWithLimits.limits](#) (page 289), [diofant.concrete.expr_with_limits.ExprWithLimits.variables](#) (page 289)

Examples

```
>>> from diofant import Integral
>>> from diofant.abc import x, y
>>> Integral(x, (x, y, 1)).free_symbols == {y}
True
```

transform(x, u)

Performs a change of variables from x to u using the relationship given by x and u which will define the transformations f and F (which are inverses of each other) as follows:

- 1.If x is a Symbol (which is a variable of integration) then u will be interpreted as some function, $f(u)$, with inverse $F(u)$. This, in effect, just makes the substitution of x with $f(x)$.
- 2.If u is a Symbol then x will be interpreted as some function, $F(x)$, with inverse $f(u)$. This is commonly referred to as u -substitution.

Once f and F have been identified, the transformation is made as follows:

$$\int_a^b x dx \rightarrow \int_{F(a)}^{F(b)} f(x) \frac{d}{dx}$$

where $F(x)$ is the inverse of $f(x)$ and the limits and integrand have been corrected so as to retain the same value after integration.

See also:

[*diofant.concrete.expr_with_limits.ExprWithLimits.variables*](#) (page 289)

Lists the integration variables

[*diofant.concrete.expr_with_limits.ExprWithLimits.as_dummy*](#) (page 288)

Replace integration variables with dummy ones

Notes

The mappings, $F(x)$ or $f(u)$, must lead to a unique integral. Linear or rational linear expression, $2*x$, $1/x$ and $\text{sqrt}(x)$, will always work; quadratic expressions like $x**2 - 1$ are acceptable as long as the resulting integrand does not depend on the sign of the solutions (see examples).

The integral will be returned unchanged if x is not a variable of integration.

x must be (or contain) only one of 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 a, b, c, d, x, u, y
>>> from diofant import Integral, S, cos, sqrt
```

```
>>> i = Integral(x*cos(x**2 - 1), (x, 0, 1))
```

transform can change the variable of integration

```
>>> i.transform(x, u)
Integral(u*cos(u**2 - 1), (u, 0, 1))
```

transform can perform u -substitution as long as a unique integrand is obtained:

```
>>> i.transform(x**2 - 1, u)
Integral(cos(u)/2, (u, -1, 0))
```

This attempt fails because $x = +/-\text{sqrt}(u + 1)$ 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))
```

class diofant.integrals.transforms.IntegralTransform

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)`.

free_symbols

This method returns the symbols that will exist when the transform is evaluated.

function

The function to be transformed.

function_variable

The dependent variable of the function to be transformed.

transform_variable

The independent transform variable.

class `diofant.integrals.transforms.MellinTransform`

Class representing unevaluated Mellin transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute Mellin transforms, see the [mellin_transform\(\)](#) (page 526) docstring.

class `diofant.integrals.transforms.InverseMellinTransform`

Class representing unevaluated inverse Mellin transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute inverse Mellin transforms, see the [inverse_mellin_transform\(\)](#) (page 527) docstring.

class `diofant.integrals.transforms.LaplaceTransform`

Class representing unevaluated Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute Laplace transforms, see the [laplace_transform\(\)](#) (page 528) docstring.

class `diofant.integrals.transforms.InverseLaplaceTransform`

Class representing unevaluated inverse Laplace transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute inverse Laplace transforms, see the [inverse_laplace_transform\(\)](#) (page 528) docstring.

class `diofant.integrals.transforms.FourierTransform`

Class representing unevaluated Fourier transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute Fourier transforms, see the [fourier_transform\(\)](#) (page 528) docstring.

class `diofant.integrals.transforms.InverseFourierTransform`

Class representing unevaluated inverse Fourier transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute inverse Fourier transforms, see the [inverse_fourier_transform\(\)](#) (page 529) docstring.

class `diofant.integrals.transforms.SineTransform`

Class representing unevaluated sine transforms.

For usage of this class, see the [IntegralTransform](#) (page 557) docstring.

For how to compute sine transforms, see the `sine_transform()` (page 529) docstring.

class `diofant.integrals.transforms.InverseSineTransform`

Class representing unevaluated inverse sine transforms.

For usage of this class, see the `IntegralTransform` (page 557) docstring.

For how to compute inverse sine transforms, see the `inverse_sine_transform()` (page 530) docstring.

class `diofant.integrals.transforms.CosineTransform`

Class representing unevaluated cosine transforms.

For usage of this class, see the `IntegralTransform` (page 557) docstring.

For how to compute cosine transforms, see the `cosine_transform()` (page 530) docstring.

class `diofant.integrals.transforms.InverseCosineTransform`

Class representing unevaluated inverse cosine transforms.

For usage of this class, see the `IntegralTransform` (page 557) docstring.

For how to compute inverse cosine transforms, see the `inverse_cosine_transform()` (page 531) docstring.

class `diofant.integrals.transforms.HankelTransform`

Class representing unevaluated Hankel transforms.

For usage of this class, see the `IntegralTransform` (page 557) docstring.

For how to compute Hankel transforms, see the `hankel_transform()` (page 531) docstring.

class `diofant.integrals.transforms.InverseHankelTransform`

Class representing unevaluated inverse Hankel transforms.

For usage of this class, see the `IntegralTransform` (page 557) docstring.

For how to compute inverse Hankel transforms, see the `inverse_hankel_transform()` (page 532) docstring.

3.9.5 TODO and Bugs

There are still lots of functions that Diofant does not know how to integrate. For bugs related to this module, see <https://github.com/sympy/sympy/issues?q=label%3AIntegration>

3.10 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 [R348] (page 1269) points and weights.

The Gauss-Legendre quadrature approximates the integral:

$$\int_{-1}^1 f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of P_n and the weights w_i are given by:

$$w_i = \frac{2}{(1 - x_i^2) (P'_n(x_i))^2}$$

Parameters n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x , w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_gen_laguerre</code>	(page 562),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_hermite</code>	(page 561),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_t</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_u</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_jacobi</code>	(page 564)	

References

[R348] (page 1269), [R349] (page 1269)

Examples

```
>>> from diofant.integrals.quadrature import gauss_legendre
>>> from diofant.printing import pprint
>>> x, w = gauss_legendre(3, 5)
>>> pprint(x)
[-0.7746, 0, 0.7746]
>>> pprint(w)
[0.55556, 0.88889, 0.55556]
>>> x, w = gauss_legendre(4, 5)
>>> pprint(x)
[-0.86114, -0.33998, 0.33998, 0.86114]
>>> pprint(w)
[0.34786, 0.65215, 0.65215, 0.34786]
```

`diofant.integrals.quadrature.gauss_laguerre`(n , n_digits)

Computes the Gauss-Laguerre quadrature [R350] (page 1269) points and weights.

The Gauss-Laguerre quadrature approximates the integral:

$$\int_0^\infty e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n and the weights w_i are given by:

$$w_i = \frac{x_i}{(n+1)^2 (L_{n+1}(x_i))^2}$$

Parameters n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x,w) tuple of lists.

See also:

<i>diofant.integrals.quadrature.gauss_legendre</i>	(page 559),	<i>dio-</i>
<i>fant.integrals.quadrature.gauss_gen_laguerre</i>	(page 562),	<i>dio-</i>
<i>fant.integrals.quadrature.gauss_hermite</i>	(page 561),	<i>dio-</i>
<i>fant.integrals.quadrature.gauss_chebyshev_t</i>	(page 563),	<i>dio-</i>
<i>fant.integrals.quadrature.gauss_chebyshev_u</i>	(page 563),	<i>dio-</i>
<i>fant.integrals.quadrature.gauss_jacobi</i>	(page 564)	

References

[R350] (page 1269), [R351] (page 1269)

Examples

```
>>> from diofant.integrals.quadrature import gauss_laguerre
>>> from diofant.printing import pprint
>>> 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]
>>> pprint(w)
[0.45896, 0.417, 0.11337, 0.010399, 0.00026102, 8.9855e-7]
```

`diofant.integrals.quadrature.gauss_hermite(n, n_digits)`

Computes the Gauss-Hermite quadrature [R352] (page 1269) points and weights.

The Gauss-Hermite quadrature approximates the integral:

$$\int_{-\infty}^{\infty} e^{-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of H_n and the weights w_i are given by:

$$w_i = \frac{2^{n-1} n! \sqrt{\pi}}{n^2 (H_{n-1}(x_i))^2}$$

Parameters n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x,w) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_legendre</code>	(page 559),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_gen_laguerre</code>	(page 562),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_t</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_u</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_jacobi</code>	(page 564)	

References

[R352] (page 1269), [R353] (page 1269), [R354] (page 1269)

Examples

```
>>> from diofant.integrals.quadrature import gauss_hermite
>>> from diofant.printing import pprint
>>> 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]
>>> pprint(w)
[0.00453, 0.15707, 0.72463, 0.72463, 0.15707, 0.00453]
```

`diofant.integrals.quadrature.gauss_gen_laguerre`(*n*, *alpha*, *n_digits*)

Computes the generalized Gauss-Laguerre quadrature [R355] (page 1269) points and weights.

The generalized Gauss-Laguerre quadrature approximates the integral:

$$\int_0^{\infty} x^{\alpha} e^{-x} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of L_n^{α} and the weights w_i are given by:

$$w_i = \frac{\Gamma(\alpha + n)}{n\Gamma(n)L_{n-1}^{\alpha}(x_i)L_{n-1}^{\alpha+1}(x_i)}$$

Parameters **n** : the order of quadrature

alpha : the exponent of the singularity, $\alpha > -1$

n_digits : number of significant digits of the points and weights to return

Returns (**x**, **w**) : the **x** and **w** are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (**x**, **w**) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_legendre</code>	(page 559),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_hermite</code>	(page 561),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_t</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_u</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_jacobi</code>	(page 564)	

References

[R355] (page 1269), [R356] (page 1269)

Examples

```
>>> from diofant import S, pprint
>>> from diofant.integrals.quadrature import gauss_gen_laguerre
>>> x, w = gauss_gen_laguerre(3, -S.Half, 5)
>>> x
[0.19016, 1.7845, 5.5253]
>>> pprint(w)
[1.4493, 0.31413, 0.00906]
```

```
>>> x, w = gauss_gen_laguerre(4, 3*S.Half, 5)
>>> x
[0.97851, 2.9904, 6.3193, 11.712]
>>> w
[0.53087, 0.67721, 0.11895, 0.0023152]
```

`diofant.integrals.quadrature.gauss_chebyshev_t(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R357] (page 1269) points and weights of the first kind.

The Gauss-Chebyshev quadrature of the first kind approximates the integral:

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of T_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n}$$

Parameters n : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x,w) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_legendre</code>	(page 559),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_hermite</code>	(page 561),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_gen_laguerre</code>	(page 562),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_u</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_jacobi</code>	(page 564)	

References

[R357] (page 1269), [R358] (page 1269)

Examples

```
>>> from diofant import S, pprint
>>> from diofant.integrals.quadrature import gauss_chebyshev_t
>>> x, w = gauss_chebyshev_t(3, 5)
>>> x
[0.86602, 0, -0.86602]
>>> w
[1.0472, 1.0472, 1.0472]
```

```
>>> x, w = gauss_chebyshev_t(6, 5)
>>> x
[0.96593, 0.70711, 0.25882, -0.25882, -0.70711, -0.96593]
>>> pprint(w)
[0.5236, 0.5236, 0.5236, 0.5236, 0.5236, 0.5236]
```

`diofant.integrals.quadrature.gauss_chebyshev_u(n, n_digits)`

Computes the Gauss-Chebyshev quadrature [R359] (page 1269) points and weights of the second kind.

The Gauss-Chebyshev quadrature of the second kind approximates the integral:

$$\int_{-1}^1 \sqrt{1-x^2} f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of U_n and the weights w_i are given by:

$$w_i = \frac{\pi}{n+1} \sin^2\left(\frac{i}{n+1}\pi\right)$$

Parameters **n** : the order of quadrature

n_digits : number of significant digits of the points and weights to return

Returns (**x**, **w**) : the **x** and **w** are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (**x**, **w**) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_legendre</code>	(page 559),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_hermite</code>	(page 561),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_gen_laguerre</code>	(page 562),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_t</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_jacobi</code>	(page 564)	

References

[R359] (page 1269), [R360] (page 1269)

Examples

```
>>> from diofant import S, pprint
>>> from diofant.integrals.quadrature import gauss_chebyshev_u
>>> x, w = gauss_chebyshev_u(3, 5)
>>> x
[0.70711, 0, -0.70711]
>>> pprint(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]
```

`diofant.integrals.quadrature.gauss_jacobi`(*n*, *alpha*, *beta*, *n_digits*)
Computes the Gauss-Jacobi quadrature [R361] (page 1269) points and weights.

The Gauss-Jacobi quadrature of the first kind approximates the integral:

$$\int_{-1}^1 (1-x)^\alpha (1+x)^\beta f(x) dx \approx \sum_{i=1}^n w_i f(x_i)$$

The nodes x_i of an order n quadrature rule are the roots of $P_n^{(\alpha,\beta)}$ and the weights w_i are given by:

$$w_i = -\frac{2n + \alpha + \beta + 2}{n + \alpha + \beta + 1} \frac{\Gamma(n + \alpha + 1)\Gamma(n + \beta + 1)}{\Gamma(n + \alpha + \beta + 1)(n + 1)!} \frac{2^{\alpha+\beta}}{P'_n(x_i)P_{n+1}^{(\alpha,\beta)}(x_i)}$$

Parameters n : the order of quadrature

alpha : the first parameter of the Jacobi Polynomial, $\alpha > -1$

beta : the second parameter of the Jacobi Polynomial, $\beta > -1$

n_digits : number of significant digits of the points and weights to return

Returns (x, w) : the x and w are lists of points and weights as Floats.

The points x_i and weights w_i are returned as (x, w) tuple of lists.

See also:

<code>diofant.integrals.quadrature.gauss_legendre</code>	(page 559),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_laguerre</code>	(page 560),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_hermite</code>	(page 561),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_gen_laguerre</code>	(page 562),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_t</code>	(page 563),	<code>dio-</code>
<code>fant.integrals.quadrature.gauss_chebyshev_u</code>	(page 563)	

References

[R361] (page 1269), [R362] (page 1269), [R363] (page 1269)

Examples

```
>>> from diofant import S, pprint
>>> from diofant.integrals.quadrature import gauss_jacobi
>>> x, w = gauss_jacobi(3, S.Half, -S.Half, 5)
>>> x
[-0.90097, -0.22252, 0.62349]
>>> w
[1.7063, 1.0973, 0.33795]
```

```
>>> x, w = gauss_jacobi(6, 1, 1, 5)
>>> pprint(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]
```

3.11 Logic

3.11.1 Introduction

The logic module for Diofant allows to form and manipulate logic expressions using symbolic and Boolean values.

3.11.2 Forming logical expressions

You can build Boolean expressions with the standard python operators & ([And](#) (page 569)), | ([Or](#) (page 569)), ~ ([diofant.logic.boolalg.Not](#) (page 570)):

```
>>> from diofant import *
>>> x, y = symbols('x,y')
>>> y | (x & y)
Or(And(x, y), y)
>>> x | y
Or(x, y)
>>> ~x
Not(x)
```

You can also form implications with >> and <<:

```
>>> x >> y
Implies(x, y)
>>> x << y
Implies(y, x)
```

Like most types in Diofant, Boolean expressions inherit from [Basic](#) (page 42):

```
>>> (y & x).subs({x: True, y: True})
true
>>> (x | y).atoms() == {x, y}
True
```

The logic module also includes the following functions to derive boolean expressions from their truth tables-

`diofant.logic.boolalg.SOPform(variables, minterms, dontcares=None)`

The SOPform function uses `simplified_pairs` and a redundant group-eliminating algorithm to convert the list of all input combos that generate '1' (the minterms) into the smallest Sum of Products form.

The variables must be given as the first argument.

Return a logical Or function (i.e., the “sum of products” or “SOP” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

References

[R364] (page 1269)

Examples

```
>>> from diofant.logic import SOPform
>>> from diofant import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1],
...             [0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> SOPform([w, x, y, z], minterms, dontcares)
Or(And(Not(w), z), And(y, z))
```

`diofant.logic.boolalg.POSform(variables, minterms, dontcares=None)`

The POSform function uses `simplified_pairs` and a redundant-group eliminating algorithm to convert the list of all input combinations that generate '1' (the minterms) into the smallest Product of Sums form.

The variables must be given as the first argument.

Return a logical And function (i.e., the “product of sums” or “POS” form) that gives the desired outcome. If there are inputs that can be ignored, pass them as a list, too.

The result will be one of the (perhaps many) functions that satisfy the conditions.

References

[R365] (page 1269)

Examples

```
>>> from diofant.logic import POSform
>>> from diofant import symbols
>>> w, x, y, z = symbols('w x y z')
>>> minterms = [[0, 0, 0, 1], [0, 0, 1, 1], [0, 1, 1, 1],
...             [1, 0, 1, 1], [1, 1, 1, 1]]
>>> dontcares = [[0, 0, 0, 0], [0, 0, 1, 0], [0, 1, 0, 1]]
>>> POSform([w, x, y, z], minterms, dontcares)
And(Or(Not(w), y), z)
```

3.11.3 Boolean functions

class `diofant.logic.boolalg.BooleanTrue`

Diofant version of True, a singleton that can be accessed via `S.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.

See also:

[*`diofant.logic.boolalg.BooleanFalse`*](#) (page 568)

Notes

There is liable to be some confusion as to when `True` should be used and when `S.true` should be used in various contexts throughout Diofant. An important thing to remember is that `sympify(True)` returns `S.true`. This means that for the most part, you can just use `True` and it will automatically be converted to `S.true` when necessary, similar to how you can generally use 1 instead of `S.One`.

The rule of thumb is:

“If the boolean in question can be replaced by an arbitrary symbolic Boolean, like `Or(x,y)` or `x > 1`, use `S.true`. Otherwise, use `True`”

In other words, use `S.true` only on those contexts where the boolean is being used as a symbolic representation of truth. For example, if the object ends up in the `.args` of any expression, then it must necessarily be `S.true` instead of `True`, as elements of `.args` must be `Basic`. On the other hand, `==` is not a symbolic operation in 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 `S.true` and `S.false` represent a two-valued logic. When in doubt, use `True`.

“`S.true == True` is `True`.”

While “`S.true` is `True`” is `False`, “`S.true == True`” is `True`, so if there is any doubt over whether a function or expression will return `S.true` or `True`, just use `==` instead of `is` to do the comparison, and it will work in either case. Finally, for boolean flags, it’s better to just use `if x` instead of `if x is True`. To quote PEP 8:

Don’t compare boolean values to `True` or `False` using `==`.

- Yes: `if greeting:`
- No: `if greeting == True:`
- Worse: `if greeting is True:`

Examples

```
>>> from diofant import sympify, true, Or
>>> sympify(True)
true
>>> ~true
false
```

```
>>> ~True
-2
>>> Or(True, False)
true
```

class diofant.logic.boolalg.BooleanFalse

Diofant version of False, a singleton that can be accessed via `S.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.

See also:

[diofant.logic.boolalg.BooleanTrue](#) (page 567)

Notes

See note in `:py:class'~diofant.logic.boolalg.BooleanTrue'`

Examples

```
>>> from diofant import sympify, false, Or, true
>>> sympify(False)
false
>>> false >> false
true
>>> False >> False
0
>>> Or(True, False)
true
```

class diofant.logic.boolalg.And

Logical AND function.

It evaluates its arguments in order, giving False immediately if any of them are False, and True if they are all True.

Notes

The `&` operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise and. Hence, `And(a, b)` and `a & b` will return different things if `a` and `b` are integers.

```
>>> And(x, y).subs(x, 1)
y
```

Examples

```
>>> from diofant.core import symbols
>>> from diofant.abc import x, y
>>> from diofant.logic.boolalg import And
>>> x & y
And(x, y)
```

class diofant.logic.boolalg.Or

Logical OR function

It evaluates its arguments in order, giving True immediately if any of them are True, and False if they are all False.

Notes

The | operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise or. Hence, Or(a,b) and a | b will return different things if a and b are integers.

```
>>> Or(x, y).subs(x, 0)
y
```

Examples

```
>>> from diofant.core import symbols
>>> from diofant.abc import x, y
>>> from diofant.logic.boolalg import Or
>>> x | y
Or(x, y)
```

class diofant.logic.boolalg.Not

Logical Not function (negation)

Returns True if the statement is False Returns False if the statement is True

Notes

- The ~ operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise not. In particular, ~a and Not(a) will be different if a is an integer. Furthermore, since bools in Python subclass from int, ~True is the same as ~1 which is -2, which has a boolean value of True. To avoid this issue, use the Diofant boolean types true and false.

```
>>> from diofant import true
>>> ~True
-2
>>> ~true
false
```

Examples

```
>>> from diofant.logic.boolalg import Not, And, Or
>>> from diofant.abc import x, A, B
>>> Not(True)
false
>>> Not(False)
true
>>> Not(And(True, False))
true
>>> Not(Or(True, False))
false
>>> Not(And(And(True, x), Or(x, False)))
Not(x)
>>> ~x
Not(x)
>>> Not(And(Or(A, B), Or(~A, ~B)))
Not(And(Or(A, B), Or(Not(A), Not(B))))
```

class diofant.logic.boolalg.Xor

Logical XOR (exclusive OR) function.

Returns True if an odd number of the arguments are True and the rest are False.

Returns False if an even number of the arguments are True and the rest are False.

Notes

The \wedge operator is provided as a convenience, but note that its use here is different from its normal use in Python, which is bitwise xor. In particular, $a \wedge b$ and $Xor(a, b)$ will be different if a and b are integers.

```
>>> Xor(x, y).subs(y, 0)
x
```

Examples

```
>>> from diofant.logic.boolalg import Xor
>>> from diofant import symbols
>>> x, y = symbols('x y')
>>> Xor(True, False)
true
>>> Xor(True, True)
false
>>> Xor(True, False, True, True, False)
true
>>> Xor(True, False, True, False)
false
>>> x ^ y
Xor(x, y)
```

class diofant.logic.boolalg.Nand

Logical NAND function.

It evaluates its arguments in order, giving True immediately if any of them are False, and False if they are all True.

Returns True if any of the arguments are False Returns False if all arguments are True

Examples

```
>>> from diofant.logic.boolalg import Nand
>>> from diofant import symbols
>>> x, y = symbols('x y')
>>> Nand(False, True)
true
>>> Nand(True, True)
false
>>> Nand(x, y)
Not(And(x, y))
```

class diofant.logic.boolalg.Nor

Logical NOR function.

It evaluates its arguments in order, giving False immediately if any of them are True, and True if they are all False.

Returns False if any argument is True Returns True if all arguments are False

Examples

```
>>> from diofant.logic.boolalg import Nor
>>> from diofant import symbols
>>> x, y = symbols('x y')
```

```
>>> Nor(True, False)
false
>>> Nor(True, True)
false
>>> Nor(False, True)
false
>>> Nor(False, False)
true
>>> Nor(x, y)
Not(Or(x, y))
```

class diofant.logic.boolalg.Implies

Logical implication.

A implies B is equivalent to $\neg A \vee B$

Accepts two Boolean arguments; A and B. Returns False if A is True and B is False Returns True otherwise.

Notes

The `>>` and `<<` operators are provided as a convenience, but note that their use here is different from their normal use in Python, which is bit shifts. Hence, `Implies(a,b)` and `a >> b` will return different things if `a` and `b` are integers. In particular, since Python

considers True and False to be integers, True >> True will be the same as 1 >> 1, i.e., 0, which has a truth value of False. To avoid this issue, use the Diofant objects true and false.

```
>>> from diofant import true, false
>>> True >> False
1
>>> true >> false
false
```

Examples

```
>>> from diofant.logic.boolalg import Implies
>>> from diofant import symbols
>>> x, y = symbols('x y')
```

```
>>> Implies(True, False)
false
>>> Implies(False, False)
true
>>> Implies(True, True)
true
>>> Implies(False, True)
true
>>> x >> y
Implies(x, y)
>>> y << x
Implies(x, y)
```

class diofant.logic.boolalg.Equivalent

Equivalence relation.

Equivalent(A, B) is True iff A and B are both True or both False

Returns True if all of the arguments are logically equivalent. Returns False otherwise.

Examples

```
>>> from diofant.logic.boolalg import Equivalent, And
>>> from diofant.abc import x, y
>>> Equivalent(False, False, False)
true
>>> Equivalent(True, False, False)
false
>>> Equivalent(x, And(x, True))
true
```

class diofant.logic.boolalg.ITE

If then else clause.

ITE(A, B, C) evaluates and returns the result of B if A is true else it returns the result of C

Examples

```
>>> from diofant.logic.boolalg import ITE, And, Xor, Or
>>> from diofant.abc import x, y, z
>>> ITE(True, False, True)
false
>>> ITE(Or(True, False), And(True, True), Xor(True, True))
true
>>> ITE(x, y, z)
ITE(x, y, z)
>>> ITE(True, x, y)
x
>>> ITE(False, x, y)
y
>>> ITE(x, y, y)
y
```

The following functions can be used to handle Conjunctive and Disjunctive Normal forms-

`diofant.logic.boolalg.to_cnf(expr, simplify=False)`

Convert a propositional logical sentence *s* to conjunctive normal form. That is, of the form $((A \mid \sim B \mid \dots) \& (B \mid C \mid \dots) \& \dots)$ If `simplify` is `True`, the `expr` is evaluated to its simplest CNF form.

Examples

```
>>> from diofant.logic.boolalg import to_cnf
>>> from diofant.abc import A, B, D
>>> to_cnf(~(A | B) | D)
And(Or(D, Not(A)), Or(D, Not(B)))
>>> to_cnf((A | B) & (A | ~A), True)
Or(A, B)
```

`diofant.logic.boolalg.to_dnf(expr, simplify=False)`

Convert a propositional logical sentence *s* to disjunctive normal form. That is, of the form $((A \& \sim B \& \dots) \mid (B \& C \& \dots) \mid \dots)$ If `simplify` is `True`, the `expr` is evaluated to its simplest DNF form.

Examples

```
>>> from diofant.logic.boolalg import to_dnf
>>> from diofant.abc import A, B, C
>>> to_dnf(B & (A | C))
Or(And(A, B), And(B, C))
>>> to_dnf((A & B) | (A & ~B) | (B & C) | (~B & C), True)
Or(A, C)
```

`diofant.logic.boolalg.is_cnf(expr)`

Test whether or not an expression is in conjunctive normal form.

Examples

```
>>> from diofant.logic.boolalg import is_cnf
>>> from diofant.abc import A, B, C
>>> is_cnf(A | B | C)
True
>>> is_cnf(A & B & C)
True
>>> is_cnf((A & B) | C)
False
```

`diofant.logic.boolalg.is_dnf(expr)`

Test whether or not an expression is in disjunctive normal form.

Examples

```
>>> from diofant.logic.boolalg import is_dnf
>>> from diofant.abc import A, B, C
>>> is_dnf(A | B | C)
True
>>> is_dnf(A & B & C)
True
>>> is_dnf((A & B) | C)
True
>>> is_dnf(A & (B | C))
False
```

3.11.4 Simplification and equivalence-testing

`diofant.logic.boolalg.simplify_logic(expr, form=None, deep=True)`

This function simplifies a boolean function to its simplified version in SOP or POS form. The return type is an Or or And object in Diofant.

Parameters *expr* : string or boolean expression

form : string ('cnf' or 'dnf') or None (default).

If 'cnf' or 'dnf', the simplest expression in the corresponding normal form is returned; if None, the answer is returned according to the form with fewest args (in CNF by default).

deep : boolean (default True)

indicates whether to recursively simplify any non-boolean functions contained within the input.

Examples

```
>>> from diofant.logic import simplify_logic
>>> from diofant.abc import x, y, z
>>> from diofant import sympify
>>> b = (~x & ~y & ~z) | (~x & ~y & z)
>>> simplify_logic(b)
And(Not(x), Not(y))
```

```
>>> sympify(b)
Or(And(Not(x), Not(y), Not(z)), And(Not(x), Not(y), z))
>>> simplify_logic(_)
And(Not(x), Not(y))
```

Diofant's `simplify()` function can also be used to simplify logic expressions to their simplest forms.

`diofant.logic.boolalg.bool_map(bool1, bool2)`

Return the simplified version of `bool1`, and the mapping of variables that makes the two expressions `bool1` and `bool2` represent the same logical behaviour for some correspondence between the variables of each. If more than one mappings of this sort exist, one of them is returned. For example, `And(x, y)` is logically equivalent to `And(a, b)` for the mapping `{x: a, y:b}` or `{x: b, y:a}`. If no such mapping exists, return `False`.

Examples

```
>>> from diofant import SOPform, bool_map, Or, And, Not, Xor
>>> from diofant.abc import w, x, y, z, a, b, c, d
>>> function1 = SOPform([x, z, y],[[1, 0, 1], [0, 0, 1]])
>>> function2 = SOPform([a, b, c],[[1, 0, 1], [1, 0, 0]])
>>> bool_map(function1, function2) == (And(Not(z), y), {y: a, z: b})
True
```

The results are not necessarily unique, but they are canonical. Here, `(w,z)` could be `(a,d)` or `(d,a)`:

```
>>> eq = Or(And(Not(y), w), And(Not(y), z), And(x, y))
>>> eq2 = Or(And(Not(c), a), And(Not(c), d), And(b, c))
>>> bool_map(eq, eq2) == (Or(And(Not(y), w), And(Not(y), z),
...                          And(x, y)), {w: a, x: b, y: c, z: d})
True
>>> eq = And(Xor(a, b), c, And(c,d))
>>> bool_map(eq, eq.subs(c, x)) == (And(Or(Not(a), Not(b)),
...                                     Or(a, b), c, d),
...                               {a: a, b: b, c: d, d: x})
True
```

3.11.5 Inference

This module implements some inference routines in propositional logic.

The function `satisfiable` will test that a given Boolean expression is satisfiable, that is, you can assign values to the variables to make the sentence *True*.

For example, the expression `x & ~x` is not satisfiable, since there are no values for `x` that make this sentence *True*. On the other hand, `(x | y) & (x | ~y) & (~x | y)` is satisfiable with both `x` and `y` being *True*.

```
>>> from diofant.logic.inference import satisfiable
>>> from diofant import Symbol
>>> x = Symbol('x')
>>> y = Symbol('y')
>>> satisfiable(x & ~x)
False
```

```
>>> satisfiable((x | y) & (x | ~y) & (~x | y)) == {x: True, y: True}
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

```
>>> from diofant.abc import A, B
>>> from diofant.logic.inference import satisfiable
>>> satisfiable(A & ~B) == {A: True, B: False}
True
>>> 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}
True
>>> next(models) == {A: True, B: True}
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
```

3.12 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 629)

3.12.1 Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```
>>> from diofant.interactive.printing import init_printing
>>> init_printing(pretty_print=True, use_unicode=False, wrap_line=False, no_
↳global=True)
>>> from diofant.matrices import Matrix, eye, zeros, ones, diag, GramSchmidt
>>> M = Matrix([[1,0,0], [0,0,0]]); M
[1 0 0]
[ ]
[0 0 0]
>>> Matrix([M, (0, 0, -1)])
[1 0 0 ]
[ ]
[0 0 0 ]
[ ]
[0 0 -1]
>>> Matrix([[1, 2, 3]])
[1 2 3]
>>> Matrix([1, 2, 3])
[1]
[ ]
[2]
[ ]
[3]
```

In addition to creating a matrix from a list of appropriately-sized lists and/or matrices, 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])
[1 2 3]
[ ]
[4 5 6]
```

More interesting (and useful), is the ability to use a 2-variable function (or lambda) to create a matrix. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):
...     if i == j:
...         return 1
...     else:
...         return 0
...
>>> Matrix(4, 4, f)
[1 0 0 0]
[ ]
[0 1 0 0]
[ ]
[0 0 1 0]
[ ]
[0 0 0 1]
```

Finally let's use `lambda` to create a 1-line matrix with 1's in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1 0 1 0]
[ ]
[0 1 0 1]
[ ]
[1 0 1 0]
```

There are also a couple of special constructors for quick matrix construction: `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively, and `diag` to put matrices or elements along the diagonal:

```
>>> eye(4)
[1 0 0 0]
[ ]
[0 1 0 0]
[ ]
[0 0 1 0]
[ ]
[0 0 0 1]
>>> zeros(2)
[0 0]
[ ]
[0 0]
>>> zeros(2, 5)
[0 0 0 0 0]
[ ]
[0 0 0 0 0]
>>> ones(3)
[1 1 1]
[ ]
[1 1 1]
[ ]
[1 1 1]
>>> ones(1, 3)
[1 1 1]
>>> diag(1, Matrix([[1, 2], [3, 4]]))
[1 0 0]
[ ]
[0 1 2]
[ ]
[0 3 4]
```

Basic Manipulation

While learning to work with matrices, let's choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

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

Now, the more standard entry access is a pair of indices which will always return the value at the corresponding row and column of the matrix:

```
>>> M[1, 2]
6
>>> M[0, 0]
1
>>> M[1, 1]
5
```

Since this is Python we're also able to slice submatrices; slices always give a matrix in return, even if the dimension is 1 x 1:

```
>>> M[0:2, 0:2]
[1 2]
[ ]
[4 5]
>>> M[2:2, 2]
[]
>>> M[:, 2]
[3]
[ ]
[6]
>>> M[:1, 2]
[3]
```

In the second example above notice that the slice 2:2 gives an empty range. Note also (in keeping with 0-based indexing of Python) the first row/column is 0.

You cannot access rows or columns that are not present unless they are in a slice:

```
>>> M[:, 10] # the 10-th column (not there)
Traceback (most recent call last):
...
IndexError: Index out of range: a[[0, 10]]
>>> M[:, 10:11] # the 10-th column (if there)
[]
>>> M[:, :10] # all columns up to the 10-th
[1 2 3]
[ ]
[4 5 6]
```

Slicing an empty matrix works as long as you use a slice for the coordinate that has no size:

```
>>> Matrix(0, 3, [])[:, 1]
[]
```

Slicing gives a copy of what is sliced, so modifications of one object do not affect the other:

```
>>> M2 = M[:, :]
>>> M2[0, 0] = 100
>>> M[0, 0] == 100
False
```

Notice that changing M2 didn't change M. Since we can slice, we can also assign entries:

```
>>> M = Matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
>>> M
[1 2 3 4 ]
[ ]
[5 6 7 8 ]
[ ]
```



```

[9  10 11 12]
[
[13 14 15 16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1  2  3  0 ]
[
[5  6  7  8 ]
[
[9  10 0  12]
[
[13 14 15 16]

```

as well as assign slices:

```

>>> M = Matrix([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1  2  3  4]
[
[5  6  7  8]
[
[9  10 0  0]
[
[13 14 0  0]

```

All the standard arithmetic operations are supported:

```

>>> M = Matrix([[1,2,3],[4,5,6],[7,8,9]])
>>> M - M
[0  0  0]
[
[0  0  0]
[
[0  0  0]
>>> M + M
[2  4  6 ]
[
[8  10 12]
[
[14 16 18]
>>> M * M
[30  36  42 ]
[
[66  81  96 ]
[
[102 126 150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[
[29]
[
[47]
>>> M**2
[30  36  42 ]
[
[66  81  96 ]

```

```
[  
[102 126 150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)  
>>> M  
[4 5 6]  
[  
[7 8 9]  
>>> M.col_del(1)  
>>> M  
[4 6]  
[  
[7 9]  
>>> v1 = Matrix([1,2,3])  
>>> v2 = Matrix([4,5,6])  
>>> v3 = v1.cross(v2)  
>>> v1.dot(v2)  
32  
>>> v2.dot(v3)  
0  
>>> v1.dot(v3)  
0
```

Recall that the `row_del()` and `col_del()` operations don't return a value - they simply change the matrix object. We can also "glue" together matrices of the appropriate size:

```
>>> M1 = eye(3)  
>>> M2 = zeros(3, 4)  
>>> M1.row_join(M2)  
[1 0 0 0 0 0 0]  
[  
[0 1 0 0 0 0 0]  
[  
[0 0 1 0 0 0 0]  
>>> M3 = zeros(4, 3)  
>>> M1.col_join(M3)  
[1 0 0]  
[  
[0 1 0]  
[  
[0 0 1]  
[  
[0 0 0]  
[  
[0 0 0]  
[  
[0 0 0]  
[  
[0 0 0]  
[  
[0 0 0]
```

Operations on entries

We are not restricted to having multiplication between two matrices:

```

>>> M = eye(3)
>>> 2*M
[2 0 0]
[  ]
[0 2 0]
[  ]
[0 0 2]
>>> 3*M
[3 0 0]
[  ]
[0 3 0]
[  ]
[0 0 3]

```

but we can also apply functions to our matrix entries using `applyfunc()`. Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```

>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2 0 0]
[  ]
[0 2 0]
[  ]
[0 0 2]

```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```

>>> from diofant import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x 0 0]
[  ]
[0 x 0]
[  ]
[0 0 x]
>>> M.subs(x, 4)
[4 0 0]
[  ]
[0 4 0]
[  ]
[0 0 4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y 0 0]
[  ]
[0 y 0]
[  ]
[0 0 y]

```

Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course, one of the first things that comes to mind is the determinant:

```

>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0

```

Another common operation is the inverse: In 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 ]
[
[ $\sqrt{6}$    $-\sqrt{3}$    $\sqrt{2}$  ]
[-----]
[ 6      3      2 ]
[
[ $\sqrt{6}$    $\sqrt{3}$   0 ]
[-----]
[ 3      3 ]
]

```

```

>>> R
[
[
[  $\sqrt{6}$   $4*\sqrt{6}$  ]
[  $\sqrt{6}$   $\frac{4*\sqrt{6}}{3}$   $2*\sqrt{6}$  ]
[
[
[  $\sqrt{3}$  ]
[ 0  $\frac{\sqrt{3}}{3}$  0 ]
[
[
[ 0 0  $\sqrt{2}$  ]
>>> Q*R
[1 1 1]
[ ]
[1 1 3]
[ ]
[2 3 4]

```

In addition to the solvers in the `solver.py` file, we can solve the system $Ax=b$ by passing the `b` vector to the matrix `A`'s `LUsolve` function. Here we'll cheat a little choose `A` and `x` then multiply to get `b`. Then we can solve for `x` and check that it's correct:

```

>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]

```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize them with respect to another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to `False`. Let's take some vectors and orthogonalize them - one normalized and one not:

```

>>> L = [Matrix([2,3,5]), Matrix([3,6,2]), Matrix([8,3,6])]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)

```

Let's take a look at the vectors:

```

>>> for i in out1:
...     print(i)
...
Matrix([
[2],
[3],
[5]])
Matrix([
[ 23/19],
[ 63/19],
[-47/19]])
Matrix([

```

```
[ 1692/353],
[-1551/706],
[ -423/706]])
>>> for i in out2:
...     print(i)
...
Matrix([
[ sqrt(38)/19],
[3*sqrt(38)/38],
[5*sqrt(38)/38]])
Matrix([
[ 23*sqrt(6707)/6707],
[ 63*sqrt(6707)/6707],
[-47*sqrt(6707)/6707]])
Matrix([
[ 12*sqrt(706)/353],
[-11*sqrt(706)/706],
[ -3*sqrt(706)/706]])
```

We can spot-check their orthogonality with `dot()` and their normality with `norm()`:

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the `matrices.py` file for all functionality.

MatrixBase Class Reference

class diofant.matrices.matrices.**MatrixBase**

- C** By-element conjugation.
 - D** Return Dirac conjugate (if `self.rows == 4`).
- See also:**
- [***conjugate*** \(page 595\)](#) By-element conjugation
 - [***H*** \(page 587\)](#) Hermite conjugation

Examples

```
>>> from diofant import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([
[1 - I,      0,      0,      0],
[      0, 1 - I,      0,      0],
[      0,      0, -1 + I,      0],
[      2,      0,      0, -1 + I]])
```

If the matrix does not have 4 rows an `AttributeError` will be raised because this property is only defined for matrices with 4 rows.

```
>>> Matrix(eye(2)).D
Traceback (most recent call last):
...
AttributeError: Matrix has no attribute D.
```

H

Return Hermite conjugate.

See also:

[*conjugate* \(page 595\)](#) By-element conjugation

[*D* \(page 586\)](#) Dirac conjugation

Examples

```
>>> from diofant import Matrix, I
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m
Matrix([
[      0],
[1 + I],
[      2],
[      3]])
>>> m.H
Matrix([[0, 1 - I, 2, 3]])
```

LDLdecomposition()

Returns the LDL Decomposition (L, D) of matrix A, such that $L * D * L.T == A$ This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a square, symmetric, positive-definite and non-singular matrix.

See also:

[*cholesky* \(page 593\)](#), [*LUdecomposition* \(page 588\)](#), [*QRdecomposition* \(page 589\)](#)

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[ 1, 0, 0],
[ 3/5, 1, 0],
[-1/5, 1/3, 1]])
>>> D
Matrix([
[25, 0, 0],
[ 0, 9, 0],
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

LDLsolve(*rhs*)

Solves $Ax = B$ using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.

See also:

[LDLdecomposition](#) (page 587), [lower_triangular_solve](#) (page 610), [upper_triangular_solve](#) (page 619), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LUsolve](#) (page 589), [QRsolve](#) (page 590), [pinv_solve](#) (page 613)

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

LUdecomposition(*iszerofunc*=<*function _iszero*>)

Returns the decomposition LU and the row swaps p.

See also:

[cholesky](#) (page 593), [LDLdecomposition](#) (page 587), [QRdecomposition](#) (page 589), [LUdecomposition_Simple](#) (page 589), [LUdecompositionFF](#) (page 588), [LUsolve](#) (page 589)

Examples

```
>>> from diofant import Matrix
>>> a = Matrix([[4, 3], [6, 3]])
>>> L, U, _ = a.LUdecomposition()
>>> L
Matrix([
[ 1, 0],
```



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

LUdecompositionFF()

Compute a fraction-free LU decomposition.

Returns 4 matrices P, L, D, U such that $PA = L D^{-1} U$. If the elements of the matrix belong to some integral domain I, then all elements of L, D and U are guaranteed to belong to I.

Reference

- W. Zhou & D.J. Jeffrey, "Fraction-free matrix factors: new forms for LU and QR factors". *Frontiers in Computer Science in China*, Vol 2, no. 1, pp. 67-80, 2008.

See also:

[LUdecomposition](#) (page 588), [LUdecomposition_Simple](#) (page 589), [LUsolve](#) (page 589)

LUdecomposition_Simple(iszerofunc=<function_ishero>)

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 588), [LUdecompositionFF](#) (page 588), [LUsolve](#) (page 589)

LUsolve(rhs, iszerofunc=<function_ishero>)

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 610), [upper_triangular_solve](#) (page 619), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [QRsolve](#) (page 590), [pinv_solve](#) (page 613), [LUdecomposition](#) (page 588)

QRdecomposition()

Return Q, R where $A = Q^*R$, Q is orthogonal and R is upper triangular.

See also:

[cholesky](#) (page 593), [LDLdecomposition](#) (page 587), [LUdecomposition](#) (page 588), [QRsolve](#) (page 590)

Examples

This is the example from wikipedia:

```
>>> from diofant import Matrix
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[ 6/7, -69/175, -58/175],
```

```
[ 3/7, 158/175,  6/175],
[-2/7,  6/35, -33/35]])
>>> R
Matrix([
[14,  21, -14],
[ 0, 175, -70],
[ 0,  0,  35]])
>>> A == Q*R
True
```

QR factorization of an identity matrix:

```
>>> A = Matrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> R
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

QRsolve(*b*)

Solve the linear system $Ax = b$.

'self' is the matrix 'A', the method argument is the vector 'b'. The method returns the solution vector 'x'. If 'b' is a matrix, the system is solved for each column of 'b' and the return value is a matrix of the same shape as 'b'.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you don't need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use `mpmath.qr_solve`.

See also:

[lower_triangular_solve](#) (page 610), [upper_triangular_solve](#) (page 619), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [LUsolve](#) (page 589), [pinv_solve](#) (page 613), [QRdecomposition](#) (page 589)

T

Matrix transposition.

add(*b*)

Return self + b

adjoint()

Conjugate transpose or Hermitian conjugation.

adjugate(*method='berkowitz'*)

Returns the adjugate matrix.

Adjugate matrix is the transpose of the cofactor matrix.

<http://en.wikipedia.org/wiki/Adjugate>

See also:

cofactorMatrix (page 593), *transpose* (page 619), *berkowitz* (page 590)

atoms(*types)

Returns the atoms that form the current object.

Examples

```
>>> from diofant.abc import x, y
>>> from diofant.matrices import Matrix
>>> Matrix([[x]])
Matrix([[x]])
>>> _.atoms() == {x}
True
```

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 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.

```
>>> from diofant import Matrix
>>> from diofant.abc import x, y, z
```

```
>>> M = Matrix([[x, y, z], [1, 0, 0], [y, z, x]])
```

```
>>> p, q, r = M.berkowitz()
```

```
>>> p # 1 x 1 M's sub-matrix
(1, -x)
```

```
>>> q # 2 x 2 M's sub-matrix
(1, -x, -y)
```

```
>>> r # 3 x 3 M's sub-matrix
(1, -2*x, x**2 - y*z - y, x*y - z**2)
```

For more information on the implemented algorithm refer to:

[1] S.J. Berkowitz, On computing the determinant in small parallel time using a small number of processors, ACM, Information Processing Letters 18, 1984, pp. 147-150

[2] M. Keber, **Division-Free computation of sub-resultants** using Bezout matrices, Tech. Report MPI-I-2006-1-006, Saarbrucken, 2006

See also:

[berkowitz_det](#) (page 592), [berkowitz_minors](#) (page 592), [berkowitz_charpoly](#) (page 591), [berkowitz_eigenvals](#) (page 592)

berkowitz_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:

See also:

[berkowitz](#) (page 590)

Examples

```
>>> from diofant import Matrix
>>> from diofant.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying *x* is optional; a Dummy with name `lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that *x* doesn't clash with an existing symbol, so using the default (`lambda`) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
-2*x + _lambda**2 - _lambda
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

berkowitz_det()

Computes determinant using Berkowitz method.

See also:

[det](#) (page 595), [berkowitz](#) (page 590)

berkowitz_eigenvals(***flags*)

Computes eigenvalues of a Matrix using Berkowitz method.

See also:

[berkowitz](#) (page 590)

berkowitz_minors()

Computes principal minors using Berkowitz method.

See also:

[berkowitz](#) (page 590)

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:

See also:

[berkowitz](#) (page 590)

Examples

```
>>> from diofant import Matrix
>>> from diofant.abc import x, y
>>> A = Matrix([[1, 3], [2, 0]])
>>> A.berkowitz_charpoly(x) == A.berkowitz_charpoly(y)
True
```

Specifying *x* is optional; a Dummy with name `lambda` is used by default (which looks good when pretty-printed in unicode):

```
>>> A.berkowitz_charpoly().as_expr()
_lambda**2 - _lambda - 6
```

No test is done to see that *x* doesn't clash with an existing symbol, so using the default (`lambda`) or your own Dummy symbol is the safest option:

```
>>> A = Matrix([[1, 2], [x, 0]])
>>> A.charpoly().as_expr()
-2*x + _lambda**2 - _lambda
>>> A.charpoly(x).as_expr()
x**2 - 3*x
```

cholesky()

Returns the Cholesky decomposition *L* of a matrix *A* such that $L * L.T = A$

A must be a square, symmetric, positive-definite and non-singular matrix.

See also:

[LDLdecomposition](#) (page 587), [LUdecomposition](#) (page 588), [QRdecomposition](#) (page 589)

Examples

```
>>> from diofant.matrices import Matrix
>>> A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T
Matrix([
[25, 15, -5],
[15, 18, 0],
[-5, 0, 11]])
```

cholesky_solve(*rhs*)

Solves $Ax = B$ using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows > cols, the least squares solution is returned.

See also:

[lower_triangular_solve](#) (page 610), [upper_triangular_solve](#) (page 619), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [LUsolve](#) (page 589), [QRsolve](#) (page 590), [pinv_solve](#) (page 613)

cofactor(*i, j, method='berkowitz'*)

Calculate the cofactor of an element.

See also:

[cofactorMatrix](#) (page 593), [minorEntry](#) (page 611), [minorMatrix](#) (page 611)

cofactorMatrix(*method='berkowitz'*)

Return a matrix containing the cofactor of each element.

See also:

[cofactor](#) (page 593), [minorEntry](#) (page 611), [minorMatrix](#) (page 611), [adjugate](#) (page 590)

col_insert(*pos, mti*)

Insert one or more columns at the given column position.

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.sparse.SparseMatrixBase.col](#) (page 645), [row_insert](#) (page 616)

Examples

```
>>> from diofant import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.col_insert(1, V)
Matrix([
[0, 1, 0, 0],
[0, 1, 0, 0],
[0, 1, 0, 0]])
```

col_join(*bott*)

Concatenates two matrices along self's last and bott's first row

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.sparse.SparseMatrixBase.col](#) (page 645), [row_join](#) (page 616)

Examples

```
>>> from diofant import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.col_join(V)
Matrix([
```

```
[0, 0, 0],
[0, 0, 0],
[0, 0, 0],
[1, 1, 1]])
```

condition_number()

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

See also:

[singular_values](#) (page 617)

Examples

```
>>> from diofant import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

conjugate()

By-element conjugation.

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 597), [multiply](#) (page 611), [multiply_elementwise](#) (page 611)

det(*method*='bareis')

Computes the matrix determinant using the method "method".

Possible values for "method": bareis ... det_bareis berkowitz ... berkowitz_det det_LU ... det_LU_decomposition

See also:

[det_bareis](#) (page 595), [berkowitz_det](#) (page 592), [det_LU_decomposition](#) (page 595)

det_LU_decomposition()

Compute matrix determinant using LU decomposition

Note that this method fails if the LU decomposition itself fails. In particular, if the matrix has no inverse this method will fail.

TODO: Implement algorithm for sparse matrices (SFF), <http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps>.

See also:

[det](#) (page 595), [det_bareis](#) (page 595), [berkowitz_det](#) (page 592)

det_bareis()

Compute matrix determinant using Bareis' fraction-free algorithm which is an extension of the well known Gaussian elimination method. This approach is best suited

for dense symbolic matrices and will result in a determinant with minimal number of fractions. It means that less term rewriting is needed on resulting formulae.

TODO: Implement algorithm for sparse matrices (SFF), <http://www.eecis.udel.edu/~saunders/papers/sffge/it5.ps>.

See also:

det (page 595), *berkowitz_det* (page 592)

diagonal_solve(*rhs*)

Solves $Ax = B$ efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

See also:

lower_triangular_solve (page 610), *upper_triangular_solve* (page 619), *cholesky_solve* (page 593), *LDLsolve* (page 588), *LUsolve* (page 589), *QRsolve* (page 590), *pinv_solve* (page 613)

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

diagonalize(*reals_only=False, sort=False, normalize=False*)

Return (P, D), where D is diagonal and

$$D = P^{-1} * M * P$$

where M is current matrix.

See also:

is_diagonal (page 602), *is_diagonalizable* (page 603)

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
Matrix([
[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> (P, D) = m.diagonalize()
>>> D
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> P
Matrix([
[-1, 0, -1],
[ 0, 0, -1],
[ 2, 1,  2]])
```



```
>>> P.inv() * m * P
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

diff(*args)

Calculate the derivative of each element in the matrix.

See also:

[integrate](#) (page 600), [limit](#) (page 610)

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

dot(b)

Return the dot product of Matrix self and b relaxing the condition of compatible dimensions: if either the number of rows or columns are the same as the length of b then the dot product is returned. If self is a row or column vector, a scalar is returned. Otherwise, a list of results is returned (and in that case the number of columns in self must match the length of b).

See also:

[cross](#) (page 595), [multiply](#) (page 611), [multiply_elementwise](#) (page 611)

Examples

```
>>> from diofant import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = [1, 1, 1]
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> M.dot(v)
[6, 15, 24]
```

dual()

Returns the dual of a matrix, which is:

$(1/2) * \text{levicivita}(i, j, k, l) * M(k, l)$ 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(*prec=None, **options*)

Apply `evalf()` to each element of self.

exp()

Return the exponentiation of a square matrix.

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

Apply `core.function.expand` to each entry of the matrix.

Examples

```
>>> from diofant.abc import x
>>> from diofant.matrices import Matrix
>>> Matrix(1, 1, [x*(x+1)])
Matrix([[x*(x + 1)])]
>>> _.expand()
Matrix([[x**2 + x]])
```

extract(*rowsList, colsList*)

Return a submatrix by specifying a list of rows and columns. Negative indices can be given. All indices must be in the range $-n \leq i < n$ where n is the number of rows or columns.

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
```

```
[0, 1],
[3, 4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

free_symbols

Returns the free symbols within the matrix.

Examples

```
>>> from diofant.abc import x
>>> from diofant.matrices import Matrix
>>> Matrix([[x], [1]]).free_symbols == {x}
True
```

get_diag_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

Examples

```
>>> from diofant import Matrix
>>> from diofant.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
Matrix([
[1, 3],
[y, z**2]])
>>> a2
Matrix([[x]])
>>> a3
Matrix([[0]])
```

has(*patterns)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from diofant import Matrix, Float
>>> from diofant.abc import x, y
>>> A = Matrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

classmethod `hstack(*args)`

Return a matrix formed by joining args horizontally (i.e. by repeated application of `row_join`).

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> Matrix.hstack(eye(2), 2*eye(2))
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2]])
```

integrate(*args)

Integrate each element of the matrix.

See also:

limit (page 610), *diff* (page 596)

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([
[x**2/2, x*y],
[ x, 0]])
>>> M.integrate((x, 0, 2))
Matrix([
[2, 2*y],
[2, 0]])
```

inv(*method=None*, ***kwargs*)

Returns the inverse of the matrix

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).

$$\bullet K^{-1} = r \cdot \text{adj}(K) \pmod{m}.$$

Examples

```
>>> from diofant import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

inverse_ADJ(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using the adjugate matrix and a determinant.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 600), [inverse_LU](#) (page 601), [inverse_GE](#) (page 601)

inverse_GE(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using Gaussian elimination.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 600), [inverse_LU](#) (page 601), [inverse_ADJ](#) (page 600)

inverse_LU(*iszerofunc*=<function *_iszero*>)

Calculates the inverse using LU decomposition.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 600), [inverse_GE](#) (page 601), [inverse_ADJ](#) (page 600)

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

```
>>> from diofant import Matrix, symbols
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_anti_symmetric()
```

```
True
>>> x, y = symbols('x y')
>>> m = Matrix(2, 3, [0, 0, x, -y, 0, 0])
>>> m
Matrix([
[ 0, 0, x],
[-y, 0, 0]])
>>> m.is_anti_symmetric()
False
```

```
>>> from diofant.abc import x, y
>>> m = Matrix(3, 3, [0, x**2 + 2*x + 1, y,
...                 -(x + 1)**2, 0, x*y,
...                 -y, -x*y, 0])
```

Simplification of matrix elements is done by default so even though two elements which should be equal and opposite wouldn't pass an equality test, the matrix is still reported as anti-symmetric:

```
>>> m[0, 1] == -m[1, 0]
False
>>> m.is_anti_symmetric()
True
```

If 'simplify=False' is used for the case when a Matrix is already simplified, this will speed things up. Here, we see that without simplification the matrix does not appear anti-symmetric:

```
>>> m.is_anti_symmetric(simplify=False)
False
```

But if the matrix were already expanded, then it would appear anti-symmetric and simplification in the `is_anti_symmetric` routine is not needed:

```
>>> m = m.expand()
>>> m.is_anti_symmetric(simplify=False)
True
```

`is_diagonal()`

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

See also:

[is_lower](#) (page 604), [is_upper](#) (page 607), [is_diagonalizable](#) (page 603), [diagonalize](#) (page 596)

Examples

```
>>> from diofant import Matrix, diag
>>> m = Matrix(2, 2, [1, 0, 0, 2])
>>> m
Matrix([
[1, 0],
[0, 2]])
```

```
>>> m.is_diagonal()
True
```

```
>>> m = Matrix(2, 2, [1, 1, 0, 2])
>>> m
Matrix([
[1, 1],
[0, 2]])
>>> m.is_diagonal()
False
```

```
>>> m = diag(1, 2, 3)
>>> m
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> m.is_diagonal()
True
```

is_diagonalizable(*reals_only=False, clear_subproducts=True*)

Check if matrix is diagonalizable.

If *reals_only*==True then check that diagonalized matrix consists of the only not complex values.

Some subproducts could be used further in other methods to avoid double calculations, By default (if *clear_subproducts*==True) they will be deleted.

See also:

[is_diagonal](#) (page 602), [diagonalize](#) (page 596)

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(3, 3, [1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
Matrix([
[1, 2, 0],
[0, 3, 0],
[2, -4, 2]])
>>> m.is_diagonalizable()
True
>>> m = Matrix(2, 2, [0, 1, 0, 0])
>>> m
Matrix([
[0, 1],
[0, 0]])
>>> m.is_diagonalizable()
False
>>> m = Matrix(2, 2, [0, 1, -1, 0])
>>> m
Matrix([
[ 0, 1],
[-1, 0]])
>>> m.is_diagonalizable()
```

```
True
>>> m.is_diagonalizable(True)
False
```

is_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i .

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant import I
>>> from diofant.abc import x
>>> 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
```

is_lower

Check if matrix is a lower triangular matrix. True can be returned even if the matrix is not square.

See also:

[is_upper](#) (page 607), [is_diagonal](#) (page 602), [is_lower_hessenberg](#) (page 604)

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
 [1, 0],
 [0, 1]])
>>> m.is_lower
True
```

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



```
[6, 6, 5]])
>>> m.is_lower
True
```

```
>>> from diofant.abc import x, y
>>> m = Matrix(2, 2, [x**2 + y, y**2 + x, 0, x + y])
>>> m
Matrix([
[x**2 + y, x + y**2],
[      0,      x + y]])
>>> m.is_lower
False
```

is_lower_hessenberg

Checks if the matrix is in the lower-Hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

See also:

[is_upper_hessenberg](#) (page 607), [is_lower](#) (page 604)

Examples

```
>>> from diofant.matrices import Matrix
>>> a = Matrix([[1, 2, 0, 0], [5, 2, 3, 0], [3, 4, 3, 7], [5, 6, 1, 1]])
>>> a
Matrix([
[1, 2, 0, 0],
[5, 2, 3, 0],
[3, 4, 3, 7],
[5, 6, 1, 1]])
>>> a.is_lower_hessenberg
True
```

is_nilpotent()

Checks if a matrix is nilpotent.

A matrix B is nilpotent if for some integer k , B^{**k} is a zero matrix.

Examples

```
>>> from diofant import Matrix
>>> a = Matrix([[0, 0, 0], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
True
```

```
>>> a = Matrix([[1, 0, 1], [1, 0, 0], [1, 1, 0]])
>>> a.is_nilpotent()
False
```

is_square

Checks if a matrix is square.

A matrix is square if the number of rows equals the number of columns. The empty matrix is square by definition, since the number of rows and the number of columns are both zero.

Examples

```
>>> from diofant import Matrix
>>> a = Matrix([[1, 2, 3], [4, 5, 6]])
>>> b = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> c = Matrix([])
>>> a.is_square
False
>>> b.is_square
True
>>> c.is_square
True
```

is_symbolic()

Checks if any elements contain Symbols.

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.is_symbolic()
True
```

is_symmetric(simplify=True)

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using 'simplify=False'; while speeding things a bit, this may however induce false negatives.

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(2, 2, [0, 1, 1, 2])
>>> m
Matrix([
[0, 1],
[1, 2]])
>>> m.is_symmetric()
True
```

```
>>> m = Matrix(2, 2, [0, 1, 2, 0])
>>> m
Matrix([
[0, 1],
[2, 0]])
```

```
>>> m.is_symmetric()
False
```

```
>>> m = Matrix(2, 3, [0, 0, 0, 0, 0, 0])
>>> m
Matrix([
[0, 0, 0],
[0, 0, 0]])
>>> m.is_symmetric()
False
```

```
>>> from diofant.abc import x, y
>>> m = Matrix(3, 3, [1, x**2 + 2*x + 1, y, (x + 1)**2, 2, 0, y, 0, 3])
>>> m
Matrix([
[      1, x**2 + 2*x + 1, y],
[(x + 1)**2,      2, 0],
[      y,      0, 3]])
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up `is_symmetric()` test by using `'simplify=False'`.

```
>>> m.is_symmetric(simplify=False)
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

is_upper

Check if matrix is an upper triangular matrix. True can be returned even if the matrix is not square.

See also:

[is_lower](#) (page 604), [is_diagonal](#) (page 602), [is_upper_hessenberg](#) (page 607)

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(2, 2, [1, 0, 0, 1])
>>> m
Matrix([
[1, 0],
[0, 1]])
>>> m.is_upper
True
```

```
>>> m = Matrix(4, 3, [5, 1, 9, 0, 4, 6, 0, 0, 5, 0, 0, 0])
>>> m
Matrix([
[5, 1, 9],
[0, 4, 6],
[0, 0, 5],
[0, 0, 0]])
```

```
>>> m.is_upper
True
```

```
>>> m = Matrix(2, 3, [4, 2, 5, 6, 1, 1])
>>> m
Matrix([
  [4, 2, 5],
  [6, 1, 1]])
>>> m.is_upper
False
```

is_upper_hessenberg

Checks if the matrix is the upper-Hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

See also:

[is_lower_hessenberg](#) (page 604), [is_upper](#) (page 607)

Examples

```
>>> from diofant.matrices import Matrix
>>> a = Matrix([[1, 4, 2, 3], [3, 4, 1, 7], [0, 2, 3, 4], [0, 0, 1, 3]])
>>> a
Matrix([
  [1, 4, 2, 3],
  [3, 4, 1, 7],
  [0, 2, 3, 4],
  [0, 0, 1, 3]])
>>> a.is_upper_hessenberg
True
```

is_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```
>>> from diofant import Matrix, zeros
>>> from diofant.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
```

```
>>> d.is_zero
True
>>> e.is_zero
```

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).

See also:

[diofant.matrices.dense.hessian](#) (page 624), [diofant.matrices.dense.wronskian](#) (page 625)

Examples

```
>>> from diofant import sin, cos, Matrix
>>> from diofant.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi), rho*cos(phi)],
[ 2*rho, 0]])
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
Matrix([
[cos(phi), -rho*sin(phi)],
[sin(phi), rho*cos(phi)]]])
```

jordan_cells(calc_transformation=True)

Return a list of Jordan cells of current matrix. This list shape Jordan matrix J.

If `calc_transformation` is specified as `False`, then transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

will not be calculated.

See also:

[jordan_form](#) (page 609)

Notes

Calculation of transformation P is not implemented yet.

Examples

```
>>> from diofant import Matrix
>>> m = Matrix(4, 4, [
...     6, 5, -2, -3,
...     -3, -1, 3, 3,
...     2, 1, -2, -3,
...     -1, 1, 5, 5])
```

```
>>> P, Jcells = m.jordan_cells()
>>> Jcells[0]
Matrix([
[2, 1],
[0, 2]])
>>> Jcells[1]
Matrix([
[2, 1],
[0, 2]])
```

jordan_form(*calc_transformation=True*)

Return Jordan form J of current matrix.

Also the transformation P such that

$$J = P^{-1} \cdot M \cdot P$$

and the jordan blocks forming J will be calculated.

See also:

[jordan_cells](#) (page 609)

Examples

```
>>> from diofant import Matrix
>>> m = Matrix([
...     [ 6, 5, -2, -3],
...     [-3, -1, 3, 3],
...     [ 2, 1, -2, -3],
...     [-1, 1, 5, 5]])
>>> P, J = m.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

key2bounds(*keys*)

Converts a key with potentially mixed types of keys (integer and slice) into a tuple of ranges and raises an error if any index is out of self's range.

See also:

[key2ij](#) (page 610)

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 610)**limit(*args)**

Calculate the limit of each element in the matrix.

See also:[integrate](#) (page 600), [diff](#) (page 596)**Examples**

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

lower_triangular_solve(rhs)Solves $Ax = B$, where A is a lower triangular matrix.**See also:**[upper_triangular_solve](#) (page 619), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [LUsolve](#) (page 589), [QRsolve](#) (page 590), [pinv_solve](#) (page 613)**minorEntry(i, j, method='berkowitz')**

Calculate the minor of an element.

See also:[minorMatrix](#) (page 611), [cofactor](#) (page 593), [cofactorMatrix](#) (page 593)**minorMatrix(i, j)**

Creates the minor matrix of a given element.

See also:[minorEntry](#) (page 611), [cofactor](#) (page 593), [cofactorMatrix](#) (page 593)**multiply(b)**Returns $\text{self} * b$ **See also:**[dot](#) (page 597), [cross](#) (page 595), [multiply_elementwise](#) (page 611)**multiply_elementwise(b)**Return the Hadamard product (elementwise product) of A and B **See also:**[cross](#) (page 595), [dot](#) (page 597), [multiply](#) (page 611)**Examples**

```
>>> from diofant.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> A.multiply_elementwise(B)
Matrix([
 [ 0, 10, 200],
 [300, 40, 5]])
```

n(*prec=None, **options*)

Apply evalf() to each element of self.

norm(*ord=None*)

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the ord parameter

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	•does not exist
inf	-	max(abs(x))
-inf	-	min(abs(x))
1	-	as below
-1	-	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	•does not exist	sum(abs(x)**ord)**(1./ord)

See also:

[normalized](#) (page 612)

Examples

```
>>> from diofant import Matrix, Symbol, trigsimp, cos, sin, oo
>>> x = Symbol('x', real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp(v.norm())
1
>>> v.norm(10)
(sin(x)**10 + cos(x)**10)**(1/10)
>>> A = Matrix([[1, 1], [1, 1]])
>>> A.norm(2) # Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1
```


normalized()

Return the normalized version of self.

See also:

[norm](#) (page 611)

nullspace(*simplify=False*)

Returns list of vectors (Matrix objects) that span nullspace of self

permuteBkwd(*perm*)

Permute the rows of the matrix with the given permutation in reverse.

See also:

[permuteFwd](#) (page 613)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.permuteBkwd([[0, 1], [0, 2]])
Matrix([
[0, 1, 0],
[0, 0, 1],
[1, 0, 0]])
```

permuteFwd(*perm*)

Permute the rows of the matrix with the given permutation.

See also:

[permuteBkwd](#) (page 612)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.permuteFwd([[0, 1], [0, 2]])
Matrix([
[0, 0, 1],
[1, 0, 0],
[0, 1, 0]])
```

pinv()

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

See also:

[diofant.matrices.matrices.MatrixBase.inv](#) (page 600), [pinv_solve](#) (page 613)

References

[R366] (page 1269)

Examples

```
>>> from diofant import Matrix
>>> Matrix([[1, 2, 3], [4, 5, 6]]).pinv()
Matrix([
[-17/18,  4/9],
[ -1/9,  1/9],
[ 13/18, -2/9]])
```

pinv_solve(*B*, *arbitrary_matrix*=None)

Solve $Ax = B$ using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of *arbitrary_matrix*. If no solutions exist, the least-squares solution is returned.

Parameters **B** : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

arbitrary_matrix : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of *wn_m* will be used, with n and m being row and column position of each symbol.

Returns **x** : Matrix

The matrix that will satisfy $Ax = B$. Will have as many rows as matrix A has columns, and as many columns as matrix B.

See also:

[lower_triangular_solve](#) (page 610), [upper_triangular_solve](#) (page 619), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [LUsolve](#) (page 589), [QRsolve](#) (page 590), [pinv](#) (page 613)

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

[R367] (page 1270)

Examples

```

>>> from diofant import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
 [ _w0_0/6 - _w1_0/3 + _w2_0/6 - 55/18],
 [-_w0_0/3 + 2*_w1_0/3 - _w2_0/3 + 1/9],
 [ _w0_0/6 - _w1_0/3 + _w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
 [-55/18],
 [ 1/9],
 [ 59/18]])

```

print_nonzero(*symbol*='X')

Shows location of non-zero entries for fast shape lookup.

Examples

```

>>> from diofant.matrices import Matrix, eye
>>> m = Matrix(2, 3, lambda i, j: i*3+j)
>>> m
Matrix([
 [0, 1, 2],
 [3, 4, 5]])
>>> m.print_nonzero()
[ XX]
[XXX]
>>> m = eye(4)
>>> m.print_nonzero("x")
[x  ]
[ x ]
[  x]
[   x]

```

project(*v*)

Return the projection of self onto the line containing *v*.

Examples

```

>>> from diofant import Matrix, S, sqrt
>>> V = Matrix([sqrt(3)/2, S.Half])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])

```

rank(*iszerofunc*=<function *_iszero*>, *simplify*=False)

Returns the rank of a matrix

```
>>> from diofant import Matrix
>>> from diofant.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rank()
2
>>> n = Matrix(3, 3, range(1, 10))
>>> n.rank()
2
```

replace(*F*, *G*, *map=False*)

Replaces Function *F* in Matrix entries with Function *G*.

Examples

```
>>> from diofant import symbols, Function, Matrix
>>> F, G = symbols('F, G', cls=Function)
>>> M = Matrix(2, 2, lambda i, j: F(i+j)) ; M
Matrix([
[F(0), F(1)],
[F(1), F(2)]])
>>> N = M.replace(F,G)
>>> N
Matrix([
[G(0), G(1)],
[G(1), G(2)]])
```

row_insert(*pos*, *mti*)

Insert one or more rows at the given row position.

See also:

[diofant.matrices.dense.DenseMatrix.row](#) (page 632), [diofant.matrices.sparse.SparseMatrixBase.row](#) (page 648), [col_insert](#) (page 594)

Examples

```
>>> from diofant import zeros, ones
>>> M = zeros(3)
>>> V = ones(1, 3)
>>> M.row_insert(1, V)
Matrix([
[0, 0, 0],
[1, 1, 1],
[0, 0, 0],
[0, 0, 0]])
```

row_join(*rhs*)

Concatenates two matrices along self's last and rhs's first column

See also:

[diofant.matrices.dense.DenseMatrix.row](#) (page 632), [diofant.matrices.sparse.SparseMatrixBase.row](#) (page 648), [col_join](#) (page 594)

Examples

```
>>> from diofant import zeros, ones
>>> M = zeros(3)
>>> V = ones(3, 1)
>>> M.row_join(V)
Matrix([
[0, 0, 0, 1],
[0, 0, 0, 1],
[0, 0, 0, 1]])
```

rref(*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

```
>>> from diofant import Matrix
>>> from diofant.abc import x
>>> m = Matrix([[1, 2], [x, 1 - 1/x]])
>>> m.rref()
(Matrix([
[1, 0],
[0, 1]]), [0, 1])
```

shape

The shape (dimensions) of the matrix as the 2-tuple (rows, cols).

Examples

```
>>> from diofant.matrices import zeros
>>> M = zeros(2, 3)
>>> M.shape
(2, 3)
>>> M.rows
2
>>> M.cols
3
```

simplify(*ratio*=1.7, *measure*=<function *count_ops*>)

Apply simplify to each element of the matrix.

Examples

```
>>> from diofant.abc import x, y
>>> from diofant import sin, cos
>>> from diofant.matrices import SparseMatrix
>>> SparseMatrix(1, 1, [x*sin(y)**2 + x*cos(y)**2])
Matrix([[x*sin(y)**2 + x*cos(y)**2]])
```

```
>>> _.simplify()
Matrix([[x]])
```

singular_values()

Compute the singular values of a Matrix

See also:

[*condition_number*](#) (page 594)

Examples

```
>>> from diofant import Matrix, Symbol
>>> x = Symbol('x', extended_real=True)
>>> A = Matrix([[0, 1, 0], [0, x, 0], [-1, 0, 0]])
>>> A.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

solve(rhs, method='GE')

Return solution to $\text{self}*\text{soln} = \text{rhs}$ using given inversion method.

For a list of possible inversion methods, see the `.inv()` docstring.

solve_least_squares(rhs, method='CH')

Return the least-square fit to the data.

By default the `cholesky_solve` routine is used (`method='CH'`); other methods of matrix inversion can be used. To find out which are available, see the docstring of the `.inv()` method.

Examples

```
>>> from diofant.matrices import Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of `S` represent coefficients of $Ax + By$ and x and y are $[2, 3]$ then $S*xy$ is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of `xy`:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by $S*xy - r$:

```
>>> S*xy - r
Matrix([
  [1/3],
  [1/3],
  [1/3]])
>>> _.norm().n(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

subs(*args, **kwargs)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from diofant.abc import x, y
>>> from diofant.matrices import SparseMatrix, Matrix
>>> 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 StrPrinter())

rowstart is the string used to start each row (by default '[').

rowend is the string used to end each row (by default ']').

rowsep is the string used to separate rows (by default a newline).

colsep is the string used to separate columns (by default ', ').

align defines how the elements are aligned. Must be one of 'left', 'right', or 'center'. You can also use '<', '>', and '^' to mean the same thing, respectively.

This is used by the string printer for Matrix.

Examples

```
>>> from diofant import Matrix
>>> 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('[%s]' % M.table(printer, rowsep='\n'))
[[ 1, 2],
[-33, 4]]
>>> print(M.table(printer, colsep=' '))
[ 1 2]
[-33 4]
>>> print(M.table(printer, align='center'))
[ 1 , 2]
[-33, 4]
>>> print(M.table(printer, rowstart='{', rowend='}'))
{ 1, 2}
{-33, 4}

```

transpose()

Matrix transposition.

upper_triangular_solve(*rhs*)

Solves $Ax = B$, where A is an upper triangular matrix.

See also:

[lower_triangular_solve](#) (page 610), [cholesky_solve](#) (page 593), [diagonal_solve](#) (page 595), [LDLsolve](#) (page 588), [LUsolve](#) (page 589), [QRsolve](#) (page 590), [pinv_solve](#) (page 613)

values()

Return non-zero values of self.

vec()

Return the Matrix converted into a one column matrix by stacking columns

See also:

[vech](#) (page 620)

Examples

```

>>> from diofant import Matrix
>>> m=Matrix([[1, 3], [2, 4]])
>>> m
Matrix([
[1, 3],
[2, 4]])
>>> m.vec()
Matrix([
[1],
[2],
[3],
[4]])

```

vech(*diagonal=True, check_symmetry=True*)

Return the unique elements of a symmetric Matrix as a one column matrix by stacking the elements in the lower triangle.

Arguments: `diagonal` - include the diagonal cells of self or not `check_symmetry` - checks symmetry of self but not completely reliably

See also:

[vec](#) (page 620)

Examples

```
>>> from diofant import Matrix
>>> m=Matrix([[1, 2], [2, 3]])
>>> m
Matrix([
[1, 2],
[2, 3]])
>>> m.vech()
Matrix([
[1],
[2],
[3]])
>>> m.vech(diagonal=False)
Matrix([[2]])
```

classmethod vstack(*args)

Return a matrix formed by joining args vertically (i.e. by repeated application of `col_join`).

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> 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

```
>>> from diofant.abc import x, y
>>> from diofant.matrices import SparseMatrix, Matrix
>>> 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**

class diofant.matrices.matrices.**ShapeError**
Wrong matrix shape

class diofant.matrices.matrices.**NonSquareMatrixError**

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

```
>>> from diofant import Matrix, ImmutableMatrix
>>> from diofant.matrices.matrices import classof
>>> 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*

```
>>> from diofant.matrices import matrix_multiply_elementwise
>>> from diofant.matrices import Matrix
>>> A = Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = Matrix([[1, 10, 100], [100, 10, 1]])
>>> matrix_multiply_elementwise(A, B)
Matrix([
 [ 0, 10, 200],
 [300, 40,  5]])
```

See also:

[*diofant.matrices.dense.DenseMatrix.__mul__*](#) (page 630)

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 622), [*diofant.matrices.dense.eye*](#) (page 622), [*diofant.matrices.dense.diag*](#) (page 622)

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 622), [*diofant.matrices.dense.eye*](#) (page 622), [*diofant.matrices.dense.diag*](#) (page 622)

`diofant.matrices.dense.eye(n, cls=None)`

Create square identity matrix $n \times n$

See also:

[`diofant.matrices.dense.diag`](#) (page 622), [`diofant.matrices.dense.zeros`](#) (page 622), [`diofant.matrices.dense.ones`](#) (page 622)

`diofant.matrices.dense.diag(*values, **kwargs)`

Create a sparse, diagonal matrix from a list of diagonal values.

See also:

[`diofant.matrices.dense.eye`](#) (page 622)

Notes

When arguments are matrices they are fitted in resultant matrix.

The returned matrix is a mutable, dense matrix. To make it a different type, send the desired class for keyword `cls`.

Examples

```
>>> from diofant.matrices import diag, Matrix, ones
>>> diag(1, 2, 3)
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
>>> diag(*[1, 2, 3])
Matrix([
[1, 0, 0],
[0, 2, 0],
[0, 0, 3]])
```

The diagonal elements can be matrices; diagonal filling will continue on the diagonal from the last element of the matrix:

```
>>> from diofant.abc import x, y, z
>>> a = Matrix([x, y, z])
>>> b = Matrix([[1, 2], [3, 4]])
>>> c = Matrix([[5, 6]])
>>> diag(a, 7, b, c)
Matrix([
[x, 0, 0, 0, 0, 0],
[y, 0, 0, 0, 0, 0],
[z, 0, 0, 0, 0, 0],
[0, 7, 0, 0, 0, 0],
[0, 0, 1, 2, 0, 0],
[0, 0, 3, 4, 0, 0],
[0, 0, 0, 0, 5, 6]])
```

When diagonal elements are lists, they will be treated as arguments to `Matrix`:

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

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

A given band off the diagonal can be made by padding with a vertical or horizontal “kerning” vector:

```
>>> hpad = ones(0, 2)
>>> vpad = ones(2, 0)
>>> diag(vpad, 1, 2, 3, hpad) + diag(hpad, 4, 5, 6, vpad)
Matrix([
[0, 0, 4, 0, 0],
[0, 0, 0, 5, 0],
[1, 0, 0, 0, 6],
[0, 2, 0, 0, 0],
[0, 0, 3, 0, 0]])
```

The type is mutable by default but can be made immutable by setting the mutable flag to False:

```
>>> type(diag(1))
<class 'diofant.matrices.dense.MutableDenseMatrix'>
>>> from diofant.matrices import ImmutableMatrix
>>> type(diag(1, cls=ImmutableMatrix))
<class 'diofant.matrices.immutable.ImmutableMatrix'>
```

`diofant.matrices.dense.jordan_cell(eigenval, n)`

Create matrix of Jordan cell kind:

Examples

```
>>> from diofant.matrices import jordan_cell
>>> from diofant.abc import x
>>> 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.

See also:

[diofant.matrices.matrices.MatrixBase.jacobian](#) (page 608), [diofant.matrices.dense.wronskian](#) (page 625)

Casoratian is defined by $k \times k$ determinant:

$$\begin{array}{cccc}
 + & a(n) & b(n) & \dots z(n) & + \\
 | & a(n+1) & b(n+1) & \dots z(n+1) & | \\
 | & \cdot & \cdot & \cdot & \cdot & | \\
 | & \cdot & \cdot & \cdot & \cdot & | \\
 + & a(n+k-1) & b(n+k-1) & \dots z(n+k-1) & +
 \end{array}$$

It proves very useful in `rsolve_hyper()` where it is applied to a generating set of a recurrence to factor out linearly dependent solutions and return a basis:

```
>>> from diofant import Symbol, casoratian, factorial
>>> 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

```
>>> from diofant.matrices import randMatrix
>>> randMatrix(3)
[25, 45, 27]
[44, 54, 9]
[23, 96, 46]
>>> randMatrix(3, 2)
[87, 29]
[23, 37]
[90, 26]
>>> randMatrix(3, 3, 0, 2)
[0, 2, 0]
[2, 0, 1]
[0, 0, 1]
>>> randMatrix(3, symmetric=True)
[85, 26, 29]
[26, 71, 43]
[29, 43, 57]
>>> A = randMatrix(3, seed=1)
>>> B = randMatrix(3, seed=2)
>>> A == B
False
>>> A == randMatrix(3, seed=1)
True
>>> randMatrix(3, symmetric=True, percent=50)
[0, 68, 43]
[0, 68, 0]
[0, 91, 34]
```

Numpy Utility Functions Reference

`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 626)

`diofant.matrices.dense.matrix2numpy(m, dtype=<class 'object'>)`
 Converts Diofant's matrix to a NumPy array.

See also:

[`diofant.matrices.dense.list2numpy`](#) (page 626)

`diofant.matrices.dense.symarray(prefix, shape, **kwargs)`
 Create a numpy ndarray of symbols (as an object array).

The created symbols are named `prefix_i1_i2_...`. You should thus provide a non-empty prefix if you want your symbols to be unique for different output arrays, as Diofant symbols with identical names are the same object.

Parameters `prefix` : string

A prefix prepended to the name of every symbol.

shape : int or tuple

Shape of the created array. If an int, the array is one-dimensional; for more than one dimension the shape must be a tuple.

****kwargs** : dict

keyword arguments passed on to Symbol

Examples

These doctests require numpy.

```
>>> from diofant import symarray
>>> symarray('', 3)
[_0 _1 _2]
```

If you want multiple symarrays to contain distinct symbols, you *must* provide unique prefixes:

```
>>> a = symarray('', 3)
>>> b = symarray('', 3)
>>> a[0] == b[0]
True
>>> a = symarray('a', 3)
>>> b = symarray('b', 3)
>>> a[0] == b[0]
False
```

Creating symarrays with a prefix:

```
>>> symarray('a', 3)
[a_0 a_1 a_2]
```

For more than one dimension, the shape must be given as a tuple:

```
>>> symarray('a', (2, 3))
[[a_0_0 a_0_1 a_0_2]
 [a_1_0 a_1_1 a_1_2]]
>>> symarray('a', (2, 3, 2))
[[[a_0_0_0 a_0_0_1]
  [a_0_1_0 a_0_1_1]
  [a_0_2_0 a_0_2_1]]

 [a_1_0_0 a_1_0_1]
 [a_1_1_0 a_1_1_1]
 [a_1_2_0 a_1_2_1]]]
```

For setting assumptions of the underlying Symbols:

```
>>> [s.is_real for s in symarray('a', 2, real=True)]
[True, True]
```

`diofant.matrices.dense.rot_axis1(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis.

See also:

`diofant.matrices.dense.rot_axis2` (page 628) Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis

`diofant.matrices.dense.rot_axis3` (page 629) Returns a rotation matrix for a rotation of θ (in radians) about the 3-axis

Examples

```
>>> from diofant import pi
>>> from diofant.matrices import rot_axis1
```

A rotation of $\pi/3$ (60 degrees):

```
>>> theta = pi/3
>>> rot_axis1(theta)
Matrix([
  [1,      0,      0],
  [0,      1/2,  sqrt(3)/2],
  [0, -sqrt(3)/2,  1/2]])
```

If we rotate by $\pi/2$ (90 degrees):

```
>>> rot_axis1(pi/2)
Matrix([
  [1, 0, 0],
  [0, 0, 1],
  [0, -1, 0]])
```

`diofant.matrices.dense.rot_axis2(theta)`

Returns a rotation matrix for a rotation of θ (in radians) about the 2-axis.

See also:

`diofant.matrices.dense.rot_axis1` (page 627) Returns a rotation matrix for a rotation of θ (in radians) about the 1-axis

diofant.matrices.dense.rot_axis3 (page 629) Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis

Examples

```
>>> from diofant import pi
>>> from diofant.matrices import rot_axis2
```

A rotation of pi/3 (60 degrees):

```
>>> theta = pi/3
>>> rot_axis2(theta)
Matrix([
 [ 1/2, 0, -sqrt(3)/2],
 [ 0, 1, 0],
 [sqrt(3)/2, 0, 1/2]])
```

If we rotate by pi/2 (90 degrees):

```
>>> rot_axis2(pi/2)
Matrix([
 [0, 0, -1],
 [0, 1, 0],
 [1, 0, 0]])
```

diofant.matrices.dense.rot_axis3(theta)

Returns a rotation matrix for a rotation of theta (in radians) about the 3-axis.

See also:

diofant.matrices.dense.rot_axis1 (page 627) Returns a rotation matrix for a rotation of theta (in radians) about the 1-axis

diofant.matrices.dense.rot_axis2 (page 628) Returns a rotation matrix for a rotation of theta (in radians) about the 2-axis

Examples

```
>>> from diofant import pi
>>> from diofant.matrices import rot_axis3
```

A rotation of pi/3 (60 degrees):

```
>>> theta = pi/3
>>> rot_axis3(theta)
Matrix([
 [ 1/2, sqrt(3)/2, 0],
 [-sqrt(3)/2, 1/2, 0],
 [ 0, 0, 1]])
```

If we rotate by pi/2 (90 degrees):

```
>>> rot_axis3(pi/2)
Matrix([
 [ 0, 1, 0],
```

```
[-1, 0, 0],  
[ 0, 0, 1]])
```

`diofant.matrices.matrices.a2idx(j, n=None)`
Return integer after making positive and validating against n.

3.12.2 Dense Matrices

`diofant.matrices.dense.MutableMatrix`
alias of *MutableDenseMatrix* (page 633)

Matrix Class Reference

`class diofant.matrices.dense.DenseMatrix`

`__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

```
>>> from diofant import Matrix, I  
>>> 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

```
>>> from diofant import Matrix
>>> 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

```
>>> from diofant import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

`col(j)`

Elementary column selector.

See also:

[diofant.matrices.dense.DenseMatrix.row](#) (page 632), [diofant.matrices.dense.MutableDenseMatrix.col_op](#) (page 633), [diofant.matrices.dense.MutableDenseMatrix.col_swap](#) (page 633), [diofant.matrices.dense.MutableDenseMatrix.col_del](#) (page 633), [diofant.matrices.matrices.MatrixBase.col_join](#) (page 594), [diofant.matrices.matrices.MatrixBase.col_insert](#) (page 594)

Examples

```
>>> from diofant import eye
>>> eye(2).col(0)
Matrix([
[1],
[0]])
```

`equals(other, failing_expression=False)`

Applies equals to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None

(or the first failing expression if `failing_expression` is `True`) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

[diofant.core.expr.Expr.equals](#) (page 70)

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x
>>> from diofant import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
True
>>> A.equals(2)
False
```

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

```
>>> from diofant import Matrix
>>> 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]])
```

row(*i*)

Elementary row selector.

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.dense.MutableDenseMatrix.row_op](#) (page 635), [diofant.matrices.dense.MutableDenseMatrix.row_swap](#) (page 636), [diofant.matrices.dense.MutableDenseMatrix.row_del](#) (page 635), [diofant.matrices.matrices.MatrixBase.row_join](#) (page 616), [diofant.matrices.matrices.MatrixBase.row_insert](#) (page 616)

Examples

```
>>> from diofant import eye
>>> eye(2).row(0)
Matrix([[1, 0]])
```

`tolist()`

Return the Matrix as a nested Python list.

Examples

```
>>> from diofant import Matrix, ones
>>> 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, 0).tolist()
[[], [], []]
```

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`

`col_del(i)`

Delete the given column.

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.dense.MutableDenseMatrix.row_del](#) (page 635)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.col_del(1)
>>> M
Matrix([
[1, 0],
[0, 0],
[0, 1]])
```

col_op(j, f)

In-place operation on col j using two-arg functor whose args are interpreted as (self[i, j], i).

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.dense.MutableDenseMatrix.row_op](#) (page 635)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([
[1, 2, 0],
[0, 1, 0],
[0, 0, 1]])
```

col_swap(i, j)

Swap the two given columns of the matrix in-place.

See also:

[diofant.matrices.dense.DenseMatrix.col](#) (page 631), [diofant.matrices.dense.MutableDenseMatrix.row_swap](#) (page 636)

Examples

```
>>> from diofant.matrices import Matrix
>>> M = Matrix([[1, 0], [1, 0]])
>>> M
Matrix([
[1, 0],
[1, 0]])
>>> M.col_swap(0, 1)
>>> M
Matrix([
[0, 1],
[0, 1]])
```

copyin_list(key, value)

Copy in elements from a list.

Parameters **key** : slice

The section of this matrix to replace.

value : iterable

The iterable to copy values from.

See also:

[diofant.matrices.dense.MutableDenseMatrix.copyin_matrix](#) (page 634)

Examples

```
>>> from diofant.matrices import eye
>>> I = eye(3)
>>> I[:2, 0] = [1, 2] # col
>>> I
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
>>> I[1, :2] = [[3, 4]]
>>> I
Matrix([
[1, 0, 0],
[3, 4, 0],
[0, 0, 1]])
```

`copyin_matrix(key, value)`

Copy in values from a matrix into the given bounds.

Parameters `key` : slice

The section of this matrix to replace.

value : Matrix

The matrix to copy values from.

See also:

[diofant.matrices.dense.MutableDenseMatrix.copyin_list](#) (page 634)

Examples

```
>>> from diofant.matrices import Matrix, eye
>>> M = Matrix([[0, 1], [2, 3], [4, 5]])
>>> I = eye(3)
>>> I[:3, :2] = M
>>> I
Matrix([
[0, 1, 0],
[2, 3, 0],
[4, 5, 1]])
>>> I[0, 1] = M
>>> I
Matrix([
[0, 0, 1],
[2, 2, 3],
[4, 4, 5]])
```

`fill(value)`

Fill the matrix with the scalar value.

See also:

[diofant.matrices.dense.zeros](#) (page 622), [diofant.matrices.dense.ones](#) (page 622)

`row_del(i)`

Delete the given row.

See also:

diofant.matrices.dense.DenseMatrix.row (page 632), *diofant.matrices.dense.MutableDenseMatrix.col_del* (page 633)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.row_del(1)
>>> M
Matrix([
[1, 0, 0],
[0, 0, 1]])
```

row_op(i, f)

In-place operation on row *i* using two-arg functor whose args are interpreted as (self[i,j],j).

See also:

diofant.matrices.dense.DenseMatrix.row (page 632), *diofant.matrices.dense.MutableDenseMatrix.zip_row_op* (page 636), *diofant.matrices.dense.MutableDenseMatrix.col_op* (page 633)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

row_swap(i, j)

Swap the two given rows of the matrix in-place.

See also:

diofant.matrices.dense.DenseMatrix.row (page 632), *diofant.matrices.dense.MutableDenseMatrix.col_swap* (page 633)

Examples

```
>>> from diofant.matrices import Matrix
>>> M = Matrix([[0, 1], [1, 0]])
>>> M
Matrix([
[0, 1],
[1, 0]])
>>> M.row_swap(0, 1)
>>> M
Matrix([
```



```
[1, 0],
[0, 1]])
```

simplify(ratio=1.7, measure=<function count_ops>)

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 922)

zip_row_op(i, k, f)

In-place operation on row `i` using two-arg functor whose args are interpreted as `(self[i,j],self[k,j])`.

See also:

[diofant.matrices.dense.DenseMatrix.row](#) (page 632), [diofant.matrices.dense.MutableDenseMatrix.row_op](#) (page 635), [diofant.matrices.dense.MutableDenseMatrix.col_op](#) (page 633)

Examples

```
>>> from diofant.matrices import eye
>>> M = eye(3)
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([
[1, 0, 0],
[2, 1, 0],
[0, 0, 1]])
```

ImmutableMatrix Class Reference

class `diofant.matrices.immutable.ImmutableMatrix`

Create an immutable version of a matrix.

Examples

```
>>> from diofant import eye
>>> from diofant.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

C

By-element conjugation.

adjoint()

Conjugate transpose or Hermitian conjugation.

as_mutable()

Returns a mutable version of this matrix

Examples

```
>>> from diofant import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])
```

conjugate()

By-element conjugation.

equals(*other*, *failing_expression=False*)

Applies equals to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if *failing_expression* is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

[diofant.core.expr.Expr.equals](#) (page 70)

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x
>>> from diofant import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
True
>>> A.equals(2)
False
```

is_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```

>>> from diofant import Matrix, zeros
>>> from diofant.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero

```

3.12.3 Sparse Matrices

class `diofant.matrices.sparse.MutableSparseMatrix(*args)`

A sparse matrix (a matrix with a large number of zero elements).

See also:

[diofant.matrices.dense.DenseMatrix](#) (page 629)

Examples

```

>>> from diofant.matrices import SparseMatrix
>>> SparseMatrix(2, 2, range(4))
Matrix([
[0, 1],
[2, 3]])
>>> SparseMatrix(2, 2, {(1, 1): 2})
Matrix([
[0, 0],
[0, 2]])

```

col_del(*k*)

Delete the given column of the matrix.

See also:

[row_del](#) (page 641)

Examples

```

>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([

```

```
[0, 0],
[0, 1]])
>>> M.col_del(0)
>>> M
Matrix([
[0],
[1]])
```

`col_join(other)`

Returns B augmented beneath A (row-wise joining):

```
[A]
[B]
```

Examples

```
>>> from diofant import SparseMatrix, Matrix, ones
>>> A = SparseMatrix(ones(3))
>>> A
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]])
>>> B = SparseMatrix.eye(3)
>>> B
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.col_join(B); C
Matrix([
[1, 1, 1],
[1, 1, 1],
[1, 1, 1],
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C == A.col_join(Matrix(B))
True
```

Joining along columns is the same as appending rows at the end of the matrix:

```
>>> C == A.row_insert(A.rows, Matrix(B))
True
```

`col_op(j, f)`

In-place operation on col *j* using two-arg functor whose args are interpreted as (self[i, j], i) for i in range(self.rows).

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
```

```
>>> M[1, 0] = -1
>>> M.col_op(1, lambda v, i: v + 2*M[i, 0]); M
Matrix([
 [ 2, 4, 0],
 [-1, 0, 0],
 [ 0, 0, 2]])
```

col_swap(*i*, *j*)
Swap, in place, columns *i* and *j*.

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.col_swap(1, 0); S
Matrix([
 [0, 1, 0],
 [1, 0, 0],
 [2, 0, 1]])
```

fill(*value*)
Fill self with the given value.

Notes

Unless many values are going to be deleted (i.e. set to zero) this will create a matrix that is slower than a dense matrix in operations.

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix.zeros(3); M
Matrix([
 [0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]])
>>> M.fill(1); M
Matrix([
 [1, 1, 1],
 [1, 1, 1],
 [1, 1, 1]])
```

row_del(*k*)
Delete the given row of the matrix.

See also:

[col_del](#) (page 639)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix([[0, 0], [0, 1]])
>>> M
Matrix([
[0, 0],
[0, 1]])
>>> M.row_del(0)
>>> M
Matrix([[0, 1]])
```

`row_join(other)`

Returns B appended after A (column-wise augmenting):

```
[A B]
```

Examples

```
>>> from diofant import SparseMatrix, Matrix
>>> A = SparseMatrix(((1, 0, 1), (0, 1, 0), (1, 1, 0)))
>>> A
Matrix([
[1, 0, 1],
[0, 1, 0],
[1, 1, 0]])
>>> B = SparseMatrix(((1, 0, 0), (0, 1, 0), (0, 0, 1)))
>>> B
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> C = A.row_join(B); C
Matrix([
[1, 0, 1, 1, 0, 0],
[0, 1, 0, 0, 1, 0],
[1, 1, 0, 0, 0, 1]])
>>> C == A.row_join(Matrix(B))
True
```

Joining at row ends is the same as appending columns at the end of the matrix:

```
>>> C == A.col_insert(A.cols, B)
True
```

`row_op(i, f)`

In-place operation on row *i* using two-arg functor whose args are interpreted as (`self[i,j],j`).

See also:

[diofant.matrices.sparse.SparseMatrixBase.row](#) (page 648), [zip_row_op](#) (page 642), [col_op](#) (page 640)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.row_op(1, lambda v, j: v + 2*M[0, j]); M
Matrix([
[2, -1, 0],
[4,  0, 0],
[0,  0, 2]])
```

`row_swap(i, j)`

Swap, in place, columns i and j .

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> S = SparseMatrix.eye(3); S[2, 1] = 2
>>> S.row_swap(1, 0); S
Matrix([
[0, 1, 0],
[1, 0, 0],
[0, 2, 1]])
```

`zip_row_op(i, k, f)`

In-place operation on row i using two-arg functor whose args are interpreted as $(self[i, j], self[k, j])$.

See also:

[diofant.matrices.sparse.SparseMatrixBase.row](#) (page 648), [row_op](#) (page 642), [col_op](#) (page 640)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> M = SparseMatrix.eye(3)*2
>>> M[0, 1] = -1
>>> M.zip_row_op(1, 0, lambda v, u: v + 2*u); M
Matrix([
[2, -1, 0],
[4,  0, 0],
[0,  0, 2]])
```

`diofant.matrices.sparse.SparseMatrix`
alias of [MutableSparseMatrix](#) (page 639)

class `diofant.matrices.sparse.SparseMatrixBase(*args)`
A sparse matrix base class.

CL

Alternate faster representation

LDLdecomposition()

Returns the LDL Decomposition (matrices L and D) of matrix A , such that $L * D * L.T == A$. A must be a square, symmetric, positive-definite and non-singular.

This method eliminates the use of square root and ensures that all the diagonal entries of L are 1.

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> A = SparseMatrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
 [ 1,  0, 0],
 [ 3/5, 1, 0],
 [-1/5, 1/3, 1]])
>>> D
Matrix([
 [25, 0, 0],
 [ 0, 9, 0],
 [ 0, 0, 9]])
>>> L * D * L.T == A
True
```

RL

Alternate faster representation

add(*other*)

Add two sparse matrices with dictionary representation.

See also:

[multiply](#) (page 648)

Examples

```
>>> from diofant.matrices import SparseMatrix, eye, ones
>>> SparseMatrix(eye(3)).add(SparseMatrix(ones(3)))
Matrix([
 [2, 1, 1],
 [1, 2, 1],
 [1, 1, 2]])
>>> SparseMatrix(eye(3)).add(-SparseMatrix(eye(3)))
Matrix([
 [0, 0, 0],
 [0, 0, 0],
 [0, 0, 0]])
```

Only the non-zero elements are stored, so the resulting dictionary that is used to represent the sparse matrix is empty:

```
>>> _._smat
{}
```

applyfunc(*f*)

Apply a function to each element of the matrix.

Examples

```

>>> from diofant.matrices import SparseMatrix
>>> m = SparseMatrix(2, 2, lambda i, j: i*2+j)
>>> m
Matrix([
[0, 1],
[2, 3]])
>>> m.applyfunc(lambda i: 2*i)
Matrix([
[0, 2],
[4, 6]])

```

as_immutable()

Returns an Immutable version of this Matrix.

as_mutable()

Returns a mutable version of this matrix.

Examples

```

>>> from diofant import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
[1, 2],
[3, 5]])

```

cholesky()

Returns the Cholesky decomposition L of a matrix A such that $L * L.T = A$
A must be a square, symmetric, positive-definite and non-singular matrix

Examples

```

>>> from diofant.matrices import SparseMatrix
>>> A = SparseMatrix(((25,15,-5),(15,18,0),(-5,0,11)))
>>> A.cholesky()
Matrix([
[ 5, 0, 0],
[ 3, 3, 0],
[-1, 1, 3]])
>>> A.cholesky() * A.cholesky().T == A
True

```

col(j)

Returns column j from self as a column vector.

See also:

[row](#) (page 648), [col_list](#) (page 645)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.col(0)
Matrix([
[1],
[3]])
```

`col_list()`

Returns a column-sorted list of non-zero elements of the matrix.

See also:

[diofant.matrices.sparse.MutableSparseMatrix.col_op](#) (page 640), [row_list](#) (page 648)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> a=SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.CL
[(0, 0, 1), (1, 0, 3), (0, 1, 2), (1, 1, 4)]
```

`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

```
>>> from diofant import Matrix
>>> m = Matrix(4, 3, range(12))
>>> m
Matrix([
[0, 1, 2],
[3, 4, 5],
[6, 7, 8],
[9, 10, 11]])
>>> m.extract([0, 1, 3], [0, 1])
Matrix([
[0, 1],
[3, 4],
[9, 10]])
```

Rows or columns can be repeated:

```
>>> m.extract([0, 0, 1], [-1])
Matrix([
[2],
```

```
[2],
[5]])
```

Every other row can be taken by using range to provide the indices:

```
>>> m.extract(range(0, m.rows, 2), [-1])
Matrix([
[2],
[8]])
```

classmethod `eye(n)`

Return an $n \times n$ identity matrix.

has(*patterns)

Test whether any subexpression matches any of the patterns.

Examples

```
>>> from diofant import SparseMatrix, Float
>>> from diofant.abc import x, y
>>> A = SparseMatrix(((1, x), (0.2, 3)))
>>> A.has(x)
True
>>> A.has(y)
False
>>> A.has(Float)
True
```

is_hermitian

Checks if the matrix is Hermitian.

In a Hermitian matrix element i,j is the complex conjugate of element j,i .

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> from diofant import I
>>> from diofant.abc import x
>>> 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

```
>>> from diofant.matrices import SparseMatrix, eye
>>> M = SparseMatrix(eye(3))
>>> M.is_symmetric()
True
>>> M[0, 2] = 1
>>> M.is_symmetric()
False
```

liupc()
Liu's algorithm, for pre-determination of the Elimination Tree of the given matrix, used in row-based symbolic Cholesky factorization.

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Gron-delle (1999) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.liupc()
([[0], [], [0], [1, 2]], [4, 3, 4, 4])
```

multiply(*other*)
Fast multiplication exploiting the sparsity of the matrix.

See also:

[add](#) (page 643)

Examples

```
>>> from diofant.matrices import SparseMatrix, ones
>>> A, B = SparseMatrix(ones(4, 3)), SparseMatrix(ones(3, 4))
>>> A.multiply(B) == 3*ones(4)
True
```

nnz()
Returns the number of non-zero elements in Matrix.

reshape(*rows, cols*)
Reshape matrix while retaining original size.

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> S = SparseMatrix(4, 2, range(8))
>>> S.reshape(2, 4)
Matrix([
[0, 1, 2, 3],
[4, 5, 6, 7]])
```

`row(i)`

Returns column *i* from self as a row vector.

See also:

`col` (page 645), `row_list` (page 648)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.row(0)
Matrix([[1, 2]])
```

`row_list()`

Returns a row-sorted list of non-zero elements of the matrix.

See also:

`diofant.matrices.sparse.MutableSparseMatrix.row_op` (page 642), `col_list` (page 645)

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a
Matrix([
[1, 2],
[3, 4]])
>>> a.RL
[(0, 0, 1), (0, 1, 2), (1, 0, 3), (1, 1, 4)]
```

`row_structure_symbolic_cholesky()`

Symbolic cholesky factorization, for pre-determination of the non-zero structure of the Cholesky factorization.

References

Symbolic Sparse Cholesky Factorization using Elimination Trees, Jeroen Van Grondelle (1999) <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.39.7582>

Examples

```
>>> from diofant.matrices import SparseMatrix
>>> S = SparseMatrix([
... [1, 0, 3, 2],
... [0, 0, 1, 0],
... [4, 0, 0, 5],
... [0, 6, 7, 0]])
>>> S.row_structure_symbolic_cholesky()
[[0], [], [0], [1, 2]]
```

`scalar_multiply(scalar)`

Scalar element-wise multiplication

`solve(rhs, method='LDL')`

Return solution to $\text{self} * \text{soln} = \text{rhs}$ using given inversion method.

For a list of possible inversion methods, see the `.inv()` docstring.

`solve_least_squares(rhs, method='LDL')`

Return the least-square fit to the data.

By default the `cholesky_solve` routine is used (`method='CH'`); other methods of matrix inversion can be used. To find out which are available, see the docstring of the `.inv()` method.

Examples

```
>>> from diofant.matrices import SparseMatrix, Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = SparseMatrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of `S` represent coefficients of $Ax + By$ and x and y are `[2, 3]` then $S * xy$ is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy :

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by $S * xy - r$:

```
>>> S*xy - r
Matrix([
```

```
[1/3],
[1/3],
[1/3]])
>>> _.norm().n(2)
0.58
```

If a different `xy` is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

tolist()

Convert this sparse matrix into a list of nested Python lists.

Examples

```
>>> from diofant.matrices import SparseMatrix, ones
>>> a = SparseMatrix(((1, 2), (3, 4)))
>>> a.tolist()
[[1, 2], [3, 4]]
```

When there are no rows then it will not be possible to tell how many columns were in the original matrix:

```
>>> SparseMatrix(ones(0, 3)).tolist()
[]
```

classmethod zeros(*r*, *c=None*)

Return an $r \times c$ matrix of zeros, square if c is omitted.

ImmutableSparseMatrix Class Reference

class diofant.matrices.immutable.ImmutableSparseMatrix(*args)

Create an immutable version of a sparse matrix.

Examples

```
>>> from diofant import eye
>>> from diofant.matrices.immutable import ImmutableSparseMatrix
>>> ImmutableSparseMatrix(1, 1, {})
Matrix([[0]])
>>> ImmutableSparseMatrix(eye(3))
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableSparseMatrix
>>> _.shape
(3, 3)
```

is_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```
>>> from diofant import Matrix, zeros
>>> from diofant.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero
```

subs(*args, **kwargs)

Return a new matrix with subs applied to each entry.

Examples

```
>>> from diofant.abc import x, y
>>> from diofant.matrices import SparseMatrix, Matrix
>>> 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

```
>>> from diofant.abc import x, y
>>> from diofant.matrices import SparseMatrix, Matrix
>>> SparseMatrix(1, 1, [x])
Matrix([[x]])
>>> _.xreplace({x: y})
Matrix([[y]])
>>> Matrix(_).xreplace({y: x})
Matrix([[x]])
```


3.12.4 Immutable Matrices

The standard `Matrix` class in Diofant is mutable. This is important for performance reasons but means that standard matrices can not interact well with the rest of Diofant. This is because the `Basic` (page 42) object, from which most Diofant classes inherit, is immutable.

The mission of the `ImmutableMatrix` (page 653) 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 42) and can thus interact more naturally with the rest of Diofant. `ImmutableMatrix` (page 653) also inherits from `MatrixExpr` (page 655), allowing it to interact freely with Diofant's Matrix Expression module.

You can turn any Matrix-like object into an `ImmutableMatrix` (page 653) by calling the constructor

```
>>> from diofant import Matrix, ImmutableMatrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> M[1, 1] = 0
>>> IM = ImmutableMatrix(M)
>>> IM
Matrix([
  [1, 2, 3],
  [4, 0, 6],
  [7, 8, 9]])
>>> IM[1, 1] = 5
Traceback (most recent call last):
...
TypeError: Can not set values in Immutable Matrix. Use Matrix instead.
```

ImmutableMatrix Class Reference

class `diofant.matrices.immutable.ImmutableMatrix`
Create an immutable version of a matrix.

Examples

```
>>> from diofant import eye
>>> from diofant.matrices import ImmutableMatrix
>>> ImmutableMatrix(eye(3))
Matrix([
  [1, 0, 0],
  [0, 1, 0],
  [0, 0, 1]])
>>> _[0, 0] = 42
Traceback (most recent call last):
...
TypeError: Cannot set values of ImmutableDenseMatrix
```

C

By-element conjugation.

`adjoint()`

Conjugate transpose or Hermitian conjugation.

`as_mutable()`

Returns a mutable version of this matrix

Examples

```
>>> from diofant import ImmutableMatrix
>>> X = ImmutableMatrix([[1, 2], [3, 4]])
>>> Y = X.as_mutable()
>>> Y[1, 1] = 5 # Can set values in Y
>>> Y
Matrix([
  [1, 2],
  [3, 5]])
```

conjugate()

By-element conjugation.

equals(*other*, *failing_expression=False*)

Applies equals to corresponding elements of the matrices, trying to prove that the elements are equivalent, returning True if they are, False if any pair is not, and None (or the first failing expression if *failing_expression* is True) if it cannot be decided if the expressions are equivalent or not. This is, in general, an expensive operation.

See also:

[diofant.core.expr.Expr.equals](#) (page 70)

Examples

```
>>> from diofant.matrices import Matrix
>>> from diofant.abc import x
>>> from diofant import cos
>>> A = Matrix([x*(x - 1), 0])
>>> B = Matrix([x**2 - x, 0])
>>> A == B
False
>>> A.simplify() == B.simplify()
True
>>> A.equals(B)
True
>>> A.equals(2)
False
```

is_zero

Checks if a matrix is a zero matrix.

A matrix is zero if every element is zero. A matrix need not be square to be considered zero. The empty matrix is zero by the principle of vacuous truth. For a matrix that may or may not be zero (e.g. contains a symbol), this will be None

Examples

```
>>> from diofant import Matrix, zeros
>>> from diofant.abc import x
>>> a = Matrix([[0, 0], [0, 0]])
>>> b = zeros(3, 4)
>>> c = Matrix([[0, 1], [0, 0]])
>>> d = Matrix([])
```

```

>>> e = Matrix([[x, 0], [0, 0]])
>>> a.is_zero
True
>>> b.is_zero
True
>>> c.is_zero
False
>>> d.is_zero
True
>>> e.is_zero

```

3.12.5 Matrix Expressions

The Matrix expression module allows users to write down statements like

```

>>> from diofant import MatrixSymbol, Matrix
>>> X = MatrixSymbol('X', 3, 3)
>>> Y = MatrixSymbol('Y', 3, 3)
>>> (X.T*X).I*Y
 $X^{-1} * X^{-1} * Y$ 

```

```

>>> Matrix(X)
Matrix([
[X[0, 0], X[0, 1], X[0, 2]],
[X[1, 0], X[1, 1], X[1, 2]],
[X[2, 0], X[2, 1], X[2, 2]]])

```

```

>>> (X*Y)[1, 2]
X[1, 0]*Y[0, 2] + X[1, 1]*Y[1, 2] + X[1, 2]*Y[2, 2]

```

where X and Y are *MatrixSymbol* (page 656)'s rather than scalar symbols.

Matrix Expressions Core Reference

class diofant.matrices.expressions.**MatrixExpr**

Superclass for Matrix Expressions

MatrixExprs represent abstract matrices, linear transformations represented within a particular basis.

Examples

```

>>> from diofant import MatrixSymbol
>>> A = MatrixSymbol('A', 3, 3)
>>> y = MatrixSymbol('y', 3, 1)
>>> x = (A.T*A).I * A * y

```

T

Matrix transposition.

as_explicit()

Returns a dense Matrix with elements represented explicitly

Returns an object of type ImmutableMatrix.

See also:

[*as_mutable*](#) (page 656) returns mutable Matrix type

Examples

```
>>> from diofant import Identity
>>> I = Identity(3)
>>> I
I
>>> I.as_explicit()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

as_mutable()

Returns a dense, mutable matrix with elements represented explicitly

See also:

[*as_explicit*](#) (page 655) returns ImmutableMatrix

Examples

```
>>> from diofant import Identity
>>> I = Identity(3)
>>> I
I
>>> I.shape
(3, 3)
>>> I.as_mutable()
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

equals(*other*)

Test elementwise equality between matrices, potentially of different types

```
>>> from diofant import Identity, eye
>>> Identity(3).equals(eye(3))
True
```

class diofant.matrices.expressions.**MatrixSymbol**

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

```
>>> from diofant import MatrixSymbol, Identity
>>> A = MatrixSymbol('A', 3, 4) # A 3 by 4 Matrix
>>> B = MatrixSymbol('B', 4, 3) # A 4 by 3 Matrix
>>> A.shape
```

```
(3, 4)
>>> 2*A*B + Identity(3)
I + 2*A*B
```

class diofant.matrices.expressions.MatAdd

A Sum of Matrix Expressions

MatAdd inherits from and operates like Diofant Add

```
>>> from diofant import MatAdd, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 5)
>>> B = MatrixSymbol('B', 5, 5)
>>> C = MatrixSymbol('C', 5, 5)
>>> MatAdd(A, B, C)
A + B + C
```

class diofant.matrices.expressions.MatMul

A product of matrix expressions

Examples

```
>>> from diofant import MatMul, MatrixSymbol
>>> A = MatrixSymbol('A', 5, 4)
>>> B = MatrixSymbol('B', 4, 3)
>>> C = MatrixSymbol('C', 3, 6)
>>> MatMul(A, B, C)
A*B*C
```

class diofant.matrices.expressions.MatPow**class diofant.matrices.expressions.Inverse**

The multiplicative inverse of a matrix expression

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the inverse, use the `.inverse()` method of matrices.

Examples

```
>>> from diofant import MatrixSymbol, Inverse
>>> A = MatrixSymbol('A', 3, 3)
>>> B = MatrixSymbol('B', 3, 3)
>>> Inverse(A)
A-1
>>> A.inverse() == Inverse(A)
True
>>> (A*B).inverse()
B-1*A-1
>>> Inverse(A*B)
(A*B)-1
```

class diofant.matrices.expressions.Transpose

The transpose of a matrix expression.

This is a symbolic object that simply stores its argument without evaluating it. To actually compute the transpose, use the `transpose()` function, or the `.T` attribute of matrices.

Examples

```
>>> from diofant.matrices import MatrixSymbol, Transpose
>>> from diofant.functions import transpose
>>> A = MatrixSymbol('A', 3, 5)
>>> B = MatrixSymbol('B', 5, 3)
>>> Transpose(A)
A'
>>> A.T == transpose(A) == Transpose(A)
True
>>> Transpose(A*B)
(A*B)'
>>> transpose(A*B)
B'*A'
```

class diofant.matrices.expressions.Trace

Matrix Trace

Represents the trace of a matrix expression.

```
>>> from diofant import MatrixSymbol, Trace, eye
>>> A = MatrixSymbol('A', 3, 3)
>>> Trace(A)
Trace(A)
```

See Also: trace

class diofant.matrices.expressions.FunctionMatrix

Represents a Matrix using a function (Lambda)

This class is an alternative to SparseMatrix

```
>>> from diofant import FunctionMatrix, symbols, Lambda, MatMul, Matrix
>>> i, j = symbols('i,j')
>>> X = FunctionMatrix(3, 3, Lambda((i, j), i + j))
>>> Matrix(X)
Matrix([
[0, 1, 2],
[1, 2, 3],
[2, 3, 4]])
```

```
>>> Y = FunctionMatrix(1000, 1000, Lambda((i, j), i + j))
```

```
>>> isinstance(Y*Y, MatMul) # this is an expression object
True
```

```
>>> (Y**2)[10,10] # So this is evaluated lazily
342923500
```

class diofant.matrices.expressions.Identity

The Matrix Identity I - multiplicative identity

```
>>> from diofant.matrices import Identity, MatrixSymbol
>>> A = MatrixSymbol('A', 3, 5)
>>> I = Identity(3)
>>> I*A
A
```

class diofant.matrices.expressions.ZeroMatrix

The Matrix Zero 0 - additive identity

```
>>> from diofant import MatrixSymbol, ZeroMatrix
>>> 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 655) or *ImmutableMatrix* (page 653) objects.

class diofant.matrices.expressions.blockmatrix.BlockMatrix

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

```
>>> from diofant import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z*Y + Z]])
```

transpose()

Return transpose of matrix.

Examples

```
>>> from diofant import MatrixSymbol, BlockMatrix, ZeroMatrix
>>> from diofant.abc import l, m, n
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> B.transpose()
Matrix([
[X', 0],
```

```
[Z', Y']]
>>> _.transpose()
Matrix([
[X, Z],
[0, Y]])
```

class diofant.matrices.expressions.blockmatrix.**BlockDiagMatrix**

A BlockDiagMatrix is a BlockMatrix with matrices only along the diagonal

```
>>> from diofant import MatrixSymbol, BlockDiagMatrix, symbols, Identity
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> BlockDiagMatrix(X, Y)
Matrix([
[X, 0],
[0, Y]])
```

diofant.matrices.expressions.blockmatrix.**block_collapse**(*expr*)

Evaluates a block matrix expression

```
>>> from diofant import MatrixSymbol, BlockMatrix, symbols, Identity, Matrix, ZeroMatrix, block_collapse
>>> n,m,l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
```

```
>>> C = BlockMatrix([[Identity(n), Z]])
>>> print(C)
Matrix([[I, Z]])
```

```
>>> print(block_collapse(C*B))
Matrix([[X, Z*Y + Z]])
```

3.13 Polynomials Manipulation

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. Diofant has a dedicated module *diofant.polys* (page 660) 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.

3.13.1 Basic functionality of the module

Polynomial manipulation algorithms and algebraic objects.

Introduction

This tutorial tries to give an overview of the functionality concerning polynomials within Diofant. All code examples assume:

```
>>> from diofant import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(pretty_print=True, use_unicode=False, wrap_line=False, no_
↳global=True)
```

Basic functionality

These functions provide different algorithms dealing with polynomials in the form of Diofant expression, like symbols, sums etc.

Division

The function `div()` (page 675) 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 = 5*x**2 + 10*x + 3
>>> g = 2*x + 2

>>> q, r = div(f, g, domain='QQ')
>>> q
5*x  5
--- + -
 2   2
>>> r
-2
>>> (q*g + r).expand()
 2
5*x  + 10*x + 3
```

As you can see, q has a non-integer coefficient. If you want to do division only in the ring of polynomials with integer coefficients, you can specify an additional parameter:

```
>>> q, r = div(f, g, domain='ZZ')
>>> q
0
>>> r
 2
5*x  + 10*x + 3
```

But be warned, that this ring is no longer Euclidean and that the degree of the remainder doesn't need to be smaller than that of f . Since 2 doesn't divide 5, $2x$ doesn't divide $5x^2$, even if the degree is smaller. But:

```
>>> g = 5*x + 1
>>> q, r = div(f, g, domain='ZZ')
>>> q
x
>>> r
9*x + 3
>>> (q*g + r).expand()
      2
5*x  + 10*x + 3
```

This also works for polynomials with multiple variables:

```
>>> f = x*y + y*z
>>> g = 3*x + 3*z
>>> q, r = div(f, g, domain='QQ')
>>> q
y
-
3
>>> r
0
```

In the last examples, all of the three variables x , y and z are assumed to be variables of the polynomials. But if you have some unrelated constant as coefficient, you can specify the variables explicitly:

```
>>> a, b, c = symbols('a,b,c')
>>> f = a*x**2 + b*x + c
>>> g = 3*x + 2
>>> q, r = div(f, g, domain='QQ')
>>> q
a*x  2*a  b
--- - --- + -
 3    9   3

>>> r
4*a  2*b
--- - --- + c
 9    3
```

GCD and LCM

With division, there is also the computation of the greatest common divisor and the least common multiple.

When the polynomials have integer coefficients, the contents' gcd is also considered:

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

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

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

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

```
>>> f = x*y/2 + y**2
>>> g = 3*x + 6*y

>>> gcd(f, g)
x + 2*y
```

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

```
>>> f = x*y**2 + x**2*y
>>> g = x**2*y**2
>>> gcd(f, g)
x*y
>>> lcm(f, g)
 3 2  2 3
x *y  + x *y
>>> (f*g).expand()
 4 3  3 4
x *y  + x *y
>>> (gcd(f, g, x, y)*lcm(f, g, x, y)).expand()
 4 3  3 4
x *y  + x *y
```

Square-free factorization

The square-free factorization of a univariate polynomial is the product of all factors (not necessarily irreducible) of degree 1, 2 etc.:

```
>>> f = 2*x**2 + 5*x**3 + 4*x**4 + x**5
>>> sqf_list(f)
(1, [(x + 2, 1), (x, 2), (x + 1, 2)])

>>> sqf(f)
 2      2
x *(x + 1) *(x + 2)
```

Factorization

This function provides factorization of univariate and multivariate polynomials with rational coefficients:

```
>>> factor(x**4/2 + 5*x**3/12 - x**2/3)
 2
x *(2*x - 1)*(3*x + 4)
-----
      12
```

```
>>> factor(x**2 + 4*x*y + 4*y**2)
      2
(x + 2*y)
```

Groebner bases

Buchberger's algorithm is implemented, supporting various monomial orders:

```
>>> groebner([x**2 + 1, y**4*x + x**3], x, y, order='lex')
      / [ 2      4      ] \
GroebnerBasis\[x + 1, y - 1], x, y, domain=ZZ, order=lex/

>>> groebner([x**2 + 1, y**4*x + x**3, x*y*z**3], x, y, z, order='grevlex')
      / [ 4      3      2      ] \
GroebnerBasis\[y - 1, z , x + 1], x, y, z, domain=ZZ, order=grevlex/
```

Solving Equations

We have (incomplete) methods to find the complex or even symbolic roots of polynomials and to solve some systems of polynomial equations:

```
>>> from diofant import roots, solve_poly_system

>>> solve(x**3 + 2*x + 3, x)
      1  \ / 11 *I  1  \ / 11 *I
[-1, - - - - - , - + - - - - ]
      2      2      2      2

>>> p = Symbol('p')
>>> q = Symbol('q')

>>> solve(x**2 + p*x + q, x)
      / 2      \ / 2
      p  \ / p - 4*q  p  \ / p - 4*q
[- - - - - , - - + - - - - ]
      2      2      2      2

>>> solve_poly_system([y - x, x - 5], x, y)
[(5, 5)]

>>> solve_poly_system([y**2 - x**3 + 1, y*x], x, y)
[(0, -I), (0, I), (1, 0), (- - - - - , 0), (- - + - - - - , 0)]
      1  \ / 3 *I      1  \ / 3 *I
      2      2      2      2
```

3.13.2 Examples from Wester's Article

Introduction

In this tutorial we present examples from Wester's article concerning comparison and critique of mathematical abilities of several computer algebra systems (see [Wester1999] (page 1270)). All the examples are related to polynomial and algebraic computations and Diofant specific remarks were added to all of them.

Examples

All examples in this tutorial are computable, so one can just copy and paste them into a Python shell and do something useful with them. All computations were done using the following setup:

```
>>> from diofant import *

>>> init_printing(pretty_print=True, use_unicode=True, wrap_line=False, no_
↳global=True)

>>> var('x,y,z,s,c,n')
(x, y, z, s, c, n)
```

Simple univariate polynomial factorization

To obtain a factorization of a polynomial use `factor()` (page 685) function. By default `factor()` (page 685) returns the result in unevaluated form, so the content of the input polynomial is left unexpanded, as in the following example:

```
>>> factor(6*x - 10)
2*(3*x - 5)
```

To achieve the same effect in a more systematic way use `primitive()` (page 683) function, which returns the content and the primitive part of the input polynomial:

```
>>> primitive(6*x - 10)
(2, 3*x - 5)
```

Note: The content and the primitive part can be computed only over a ring. To simplify coefficients of a polynomial over a field use `monic()` (page 682).

Univariate GCD, resultant and factorization

Consider univariate polynomials f , g and h over integers:

```
>>> f = 64*x**34 - 21*x**47 - 126*x**8 - 46*x**5 - 16*x**60 - 81
>>> g = 72*x**60 - 25*x**25 - 19*x**23 - 22*x**39 - 83*x**52 + 54*x**10 + 81
>>> h = 34*x**19 - 25*x**16 + 70*x**7 + 20*x**3 - 91*x - 86
```

We can compute the greatest common divisor (GCD) of two polynomials using `gcd()` (page 681) function:

```
>>> gcd(f, g)
1
```

We see that `f` and `g` have no common factors. However, `f*h` and `g*h` have an obvious factor `h`:

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

The same can be verified using the resultant of univariate polynomials:

```
>>> resultant(expand(f*h), expand(g*h))
0
```

Factorization of large univariate polynomials (of degree 120 in this case) over integers is also possible:

```
>>> factor(expand(f*g))
( 60      47      34      8      5      ) ( 60      52      39      25      )
 23      10      )
-(16*x  + 21*x  - 64*x  + 126*x  + 46*x  + 81)·(72*x  - 83*x  - 22*x  - 25*x  -
 19*x  + 54*x  + 81)
```

Multivariate GCD and factorization

What can be done in univariate case, can be also done for multivariate polynomials. Consider the following polynomials `f`, `g` and `h` in $\mathbb{Z}[x, y, z]$:

```
>>> f = 24*x*y**19*z**8 - 47*x**17*y**5*z**8 + 6*x**15*y**9*z**2 - 3*x**22 + 5
>>> g = 34*x**5*y**8*z**13 + 20*x**7*y**7*z**7 + 12*x**9*y**16*z**4 + 80*y**14*z
>>> h = 11*x**12*y**7*z**13 - 23*x**2*y**8*z**10 + 47*x**17*y**5*z**8
```

As previously, we can verify that `f` and `g` have no common factors:

```
>>> gcd(f, g)
1
```

However, `f*h` and `g*h` have an obvious factor `h`:

```
>>> gcd(expand(f*h), expand(g*h)) - h
0
```

Multivariate factorization of large polynomials is also possible:

```
>>> factor(expand(f*g))
7 ( 9 9 3      7 6      5 12      7 ) ( 22      17 5 8      15 )
 9 2      19 8      )
-2·y  ·z·(6·x  ·y  ·z  + 10·x  ·z  + 17·x  ·y·z  + 40·y  )·(3·x  + 47·x  ·y  ·z  - 6·x  ·y·
  ·z  - 24·x·y  ·z  - 5)
```

Support for symbols in exponents

Polynomial manipulation functions provided by *diofant.polys* (page 660) are mostly used with integer exponents. However, it's perfectly valid to compute with symbolic exponents, e.g.:

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

Testing if polynomials have common zeros

To test if two polynomials have a root in common we can use *resultant()* (page 677) function. The theory says that the resultant of two polynomials vanishes if there is a common zero of those polynomials. For example:

```
>>> resultant(3*x**4 + 3*x**3 + x**2 - x - 2, x**3 - 3*x**2 + x + 5)
0
```

We can visualize this fact by factoring the polynomials:

```
>>> factor(3*x**4 + 3*x**3 + x**2 - x - 2)
(x + 1) * (3*x3 + x - 2)
>>> factor(x**3 - 3*x**2 + x + 5)
(x + 1) * (x2 - 4*x + 5)
```

In both cases we obtained the factor $x + 1$ which tells us that the common root is $x = -1$.

Normalizing simple rational functions

To remove common factors from the numerator and the denominator of a rational function the elegant way, use *cancel()* (page 688) function. For example:

```
>>> cancel((x**2 - 4)/(x**2 + 4*x + 4))
x - 2
-----
x + 2
```

Expanding expressions and factoring back

One can work easily with expressions in both expanded and factored forms. Consider a polynomial f in expanded form. We differentiate it and factor the result back:

```
>>> f = expand((x + 1)**20)
>>> g = diff(f, x)
>>> factor(g)
```

$$20 \cdot (x + 1)^{19}$$

The same can be achieved in factored form:

```
>>> diff((x + 1)**20, x)
20 \cdot (x + 1)^{19}
```

Factoring in terms of cyclotomic polynomials

Diofant can very efficiently decompose polynomials of the form $x^n \pm 1$ in terms of cyclotomic polynomials:

```
>>> factor(x**15 - 1)
(x - 1) \cdot (x^2 + x + 1) \cdot (x^4 + x^3 + x^2 + x + 1) \cdot (x^8 - x^7 + x^5 - x^4 + x^3 - x^2 + 1)
```

The original Wester's example was $x^{100} - 1$, but was truncated for readability purpose. Note that this is not a big struggle for `factor()` (page 685) to decompose polynomials of degree 1000 or greater.

Univariate factoring over Gaussian numbers

Consider a univariate polynomial f with integer coefficients:

```
>>> f = 4*x**4 + 8*x**3 + 77*x**2 + 18*x + 153
```

We want to obtain a factorization of f over Gaussian numbers. To do this we use `factor()` (page 685) as previously, but this time we set `gaussian` keyword to `True`:

```
>>> factor(f, gaussian=True)
4 \cdot \left( x - \frac{3 \cdot i}{2} \right) \cdot \left( x + \frac{3 \cdot i}{2} \right) \cdot (x + 1 - 4 \cdot i) \cdot (x + 1 + 4 \cdot i)
```

As the result we got a splitting factorization of f with monic factors (this is a general rule when computing in a field with Diofant). The `gaussian` keyword is useful for improving code readability, however the same result can be computed using more general syntax:

```
>>> factor(f, extension=I)
4 \cdot \left( x - \frac{3 \cdot i}{2} \right) \cdot \left( x + \frac{3 \cdot i}{2} \right) \cdot (x + 1 - 4 \cdot i) \cdot (x + 1 + 4 \cdot i)
```

Computing with automatic field extensions

Consider two univariate polynomials f and g :

```
>>> f = x**3 + (sqrt(2) - 2)*x**2 - (2*sqrt(2) + 3)*x - 3*sqrt(2)
>>> g = x**2 - 2
```


We would like to reduce degrees of the numerator and the denominator of a rational function f/g . To do this we employ `cancel()` (page 688) function:

```
>>> cancel(f/g)
      3      2
x  - 2·x  + √2 ·x  - 3·x - 2·√2 ·x - 3·√2
-----
                2
                x  - 2
```

Unfortunately nothing interesting happened. This is because by default Diofant treats $\sqrt{2}$ as a generator, obtaining a bivariate polynomial for the numerator. To make `cancel()` (page 688) recognize algebraic properties of $\sqrt{2}$, one needs to use `extension` keyword:

```
>>> cancel(f/g, extension=True)
      2
x  - 2·x - 3
-----
x - √2
```

Setting `extension=True` tells `cancel()` (page 688) to find minimal algebraic number domain for the coefficients of f/g . The automatically inferred domain is $\mathbb{Q}(\sqrt{2})$. If one doesn't want to rely on automatic inference, the same result can be obtained by setting the `extension` keyword with an explicit algebraic number:

```
>>> cancel(f/g, extension=sqrt(2))
      2
x  - 2·x - 3
-----
x - √2
```

Univariate factoring over various domains

Consider a univariate polynomial f with integer coefficients:

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

With `diofant.polys` (page 660) we can obtain factorizations of f over different domains, which includes:

- rationals:

```
>>> factor(f)
      2      2
(x  - x - 1)·(x  + x - 1)
```

- finite fields:

```
>>> factor(f, modulus=5)
      2      2
(x - 2) · (x + 2)
```

- algebraic numbers:

```
>>> alg = AlgebraicNumber((sqrt(5) - 1)/2, alias='alpha')
>>> factor(f, extension=alg)
(x -  $\alpha$ )·(x +  $\alpha$ )·(x - 1 -  $\alpha$ )·(x +  $\alpha$  + 1)
```

Factoring polynomials into linear factors

Currently Diofant can factor polynomials into irreducibles over various domains, which can result in a splitting factorization (into linear factors). However, there is currently no systematic way to infer a splitting field (algebraic number field) automatically. In future the following syntax will be implemented:

```
>>> factor(x**3 + x**2 - 7, split=True)
Traceback (most recent call last):
...
NotImplementedError: 'split' option is not implemented yet
```

Note this is different from `extension=True`, because the later only tells how expression parsing should be done, not what should be the domain of computation. One can simulate the `split` keyword for several classes of polynomials using `solve()` (page 1054) function.

Advanced factoring over finite fields

Consider a univariate polynomial f with integer coefficients:

```
>>> f = x**11 + x + 1
```

We can factor f over a large finite field F_{65537} :

```
>>> factor(f, modulus=65537)
( 2 ) ( 9 8 6 5 3 2 )
(x + x + 1)·(x - x + x - x + x - x + 1)
```

and expand the resulting factorization back:

```
>>> expand(_)
11
x + x + 1
```

obtaining polynomial f . This was done using symmetric polynomial representation over finite fields The same thing can be done using non-symmetric representation:

```
>>> factor(f, modulus=65537, symmetric=False)
( 2 ) ( 9 8 6 5 3 2 )
(x + x + 1)·(x + 65536·x + x + 65536·x + x + 65536·x + 1)
```

As with symmetric representation we can expand the factorization to get the input polynomial back. This time, however, we need to truncate coefficients of the expanded polynomial modulo 65537:

```
>>> trunc(expand(_), 65537)
11
x + x + 1
```

Working with expressions as polynomials

Consider a multivariate polynomial f in $\mathbb{Z}[x, y, z]$:

```
>>> f = expand((x - 2*y**2 + 3*z**3)**20)
```

We want to compute factorization of f . To do this we use `factor` as usually, however we note that the polynomial in consideration is already in expanded form, so we can tell the factorization routine to skip expanding f :

```
>>> factor(f, expand=False)
(
  2      3
 (x - 2·y + 3·z )
)
```

The default in *diofant.polys* (page 660) is to expand all expressions given as arguments to polynomial manipulation functions and *Poly* (page 690) class. If we know that expanding is unnecessary, then by setting `expand=False` we can save quite a lot of time for complicated inputs. This can be really important when computing with expressions like:

```
>>> g = expand((sin(x) - 2*cos(y)**2 + 3*tan(z)**3)**20)
>>> factor(g, expand=False)
(
  2      3
 (-sin(x) + 2·cos (y) - 3·tan (z))
)
```

Computing reduced Gröbner bases

To compute a reduced Gröbner basis for a set of polynomials use *groebner()* (page 689) function. The function accepts various monomial orderings, e.g.: `lex`, `grlex` and `grevlex`, or a user defined one, via `order` keyword. The `lex` ordering is the most interesting because it has elimination property, which means that if the system of polynomial equations to *groebner()* (page 689) is zero-dimensional (has finite number of solutions) the last element of the basis is a univariate polynomial. Consider the following example:

```
>>> f = expand((1 - c**2)**5 * (1 - s**2)**5 * (c**2 + s**2)**10)
>>> groebner([f, c**2 + s**2 - 1])
GroebnerBasis\([c2 + s2 - 1, c18 - 5·c16 + 10·c14 - 10·c12 + 5·c10 - c8], s, c,
↳domain=ℤ, order=lex)
```

The result is an ordinary Python list, so we can easily apply a function to all its elements, for example we can factor those elements:

```
>>> list(map(factor, _))
[ 2      10      5      5
 (c + s - 1, c ·(c - 1) ·(c + 1) )]
```

From the above we can easily find all solutions of the system of polynomial equations. Or we can use *solve()* (page 1054) to achieve this in a more systematic way:

```
>>> solve([f, s**2 + c**2 - 1], c, s)
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

Multivariate factoring over algebraic numbers

Computing with multivariate polynomials over various domains is as simple as in univariate case. For example consider the following factorization over $\mathbb{Q}(\sqrt{-3})$:

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

Note: Currently multivariate polynomials over finite fields aren't supported.

Partial fraction decomposition

Consider a univariate rational function f with integer coefficients:

```
>>> f = (x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2)
```

To decompose f into partial fractions use `apart()` (page 737) function:

```
>>> apart(f)
3      2      2
----- - ----- + -----
x + 2   x + 1   (x + 1)2
```

To return from partial fractions to the rational function use a composition of `together()` (page 736) and `cancel()` (page 688):

```
>>> cancel(together(_))
      2
      x  + 2*x + 3
-----
      3      2
      x  + 4*x  + 5*x + 2
```

Literature

3.13.3 Polynomials Manipulation Module Reference

Basic polynomial manipulation functions

`diofant.polys.polytools.poly(expr, *gens, **args)`
Efficiently transform an expression into a polynomial.

Examples

```
>>> from diofant import poly
>>> from diofant.abc import x
```

```
>>> poly(x*(x**2 + x - 1)**2)
Poly(x**5 + 2*x**4 - x**3 - 2*x**2 + x, x, domain='ZZ')
```

`diofant.polys.polytools.poly_from_expr(expr, *gens, **args)`

Construct a polynomial from an expression.

`diofant.polys.polytools.parallel_poly_from_expr(exprs, *gens, **args)`

Construct polynomials from expressions.

`diofant.polys.polytools.degree(f, *gens, **args)`

Return the degree of f in the given variable.

The degree of 0 is negative infinity.

Examples

```
>>> from diofant import degree
>>> from diofant.abc import x, y
```

```
>>> degree(x**2 + y*x + 1, gen=x)
2
>>> degree(x**2 + y*x + 1, gen=y)
1
>>> degree(0, x)
-oo
```

`diofant.polys.polytools.degree_list(f, *gens, **args)`

Return a list of degrees of f in all variables.

Examples

```
>>> from diofant import degree_list
>>> from diofant.abc import x, y
```

```
>>> degree_list(x**2 + y*x + 1)
(2, 1)
```

`diofant.polys.polytools.LC(f, *gens, **args)`

Return the leading coefficient of f .

Examples

```
>>> from diofant import LC
>>> from diofant.abc import x, y
```

```
>>> 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

```
>>> from diofant import LM
>>> from diofant.abc import x, y
```

```
>>> 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

```
>>> from diofant import LT
>>> from diofant.abc import x, y
```

```
>>> LT(4*x**2 + 2*x*y**2 + x*y + 3*y)
4*x**2
```

`diofant.polys.polytools.pdiv(f, g, *gens, **args)`
Compute polynomial pseudo-division of *f* and *g*.

Examples

```
>>> from diofant import pdiv
>>> from diofant.abc import x
```

```
>>> pdiv(x**2 + 1, 2*x - 4)
(2*x + 4, 20)
```

`diofant.polys.polytools.prem(f, g, *gens, **args)`
Compute polynomial pseudo-remainder of *f* and *g*.

Examples

```
>>> from diofant import prem
>>> from diofant.abc import x
```

```
>>> prem(x**2 + 1, 2*x - 4)
20
```

`diofant.polys.polytools.pquo(f, g, *gens, **args)`
Compute polynomial pseudo-quotient of *f* and *g*.

Examples

```
>>> from diofant import pquo
>>> from diofant.abc import x
```

```
>>> pquo(x**2 + 1, 2*x - 4)
2*x + 4
>>> pquo(x**2 - 1, 2*x - 1)
2*x + 1
```

`diofant.polys.polytools.pexquo(f, g, *gens, **args)`
 Compute polynomial exact pseudo-quotient of *f* and *g*.

Examples

```
>>> from diofant import pexquo
>>> from diofant.abc import x
```

```
>>> pexquo(x**2 - 1, 2*x - 2)
2*x + 2
```

```
>>> pexquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`diofant.polys.polytools.div(f, g, *gens, **args)`
 Compute polynomial division of *f* and *g*.

Examples

```
>>> from diofant import div, ZZ, QQ
>>> from diofant.abc import x
```

```
>>> div(x**2 + 1, 2*x - 4, domain=ZZ)
(0, x**2 + 1)
>>> div(x**2 + 1, 2*x - 4, domain=QQ)
(x/2 + 1, 5)
```

`diofant.polys.polytools.rem(f, g, *gens, **args)`
 Compute polynomial remainder of *f* and *g*.

Examples

```
>>> from diofant import rem, ZZ, QQ
>>> from diofant.abc import x
```

```
>>> rem(x**2 + 1, 2*x - 4, domain=ZZ)
x**2 + 1
```

```
>>> rem(x**2 + 1, 2*x - 4, domain=QQ)
5
```

`diofant.polys.polytools.quo(f, g, *gens, **args)`
Compute polynomial quotient of *f* and *g*.

Examples

```
>>> from diofant import quo
>>> from diofant.abc import x
```

```
>>> quo(x**2 + 1, 2*x - 4)
x/2 + 1
>>> quo(x**2 - 1, x - 1)
x + 1
```

`diofant.polys.polytools.exquo(f, g, *gens, **args)`
Compute polynomial exact quotient of *f* and *g*.

Examples

```
>>> from diofant import exquo
>>> from diofant.abc import x
```

```
>>> exquo(x**2 - 1, x - 1)
x + 1
```

```
>>> exquo(x**2 + 1, 2*x - 4)
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

`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

```
>>> from diofant import half_gcdex
>>> from diofant.abc import x
```

```
>>> half_gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x + 1)
```

`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

```
>>> from diofant import gcdex
>>> from diofant.abc import x
```

```
>>> gcdex(x**4 - 2*x**3 - 6*x**2 + 12*x + 15, x**3 + x**2 - 4*x - 4)
(-x/5 + 3/5, x**2/5 - 6*x/5 + 2, x + 1)
```

`diofant.polys.polytools.invert(f, g, *gens, **args)`
 Invert *f* modulo *g* when possible.

Examples

```
>>> from diofant import invert
>>> from diofant.abc import x
```

```
>>> invert(x**2 - 1, 2*x - 1)
-4/3
```

```
>>> invert(x**2 - 1, x - 1)
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

`diofant.polys.polytools.subresultants(f, g, *gens, **args)`
 Compute subresultant PRS of *f* and *g*.

Examples

```
>>> from diofant import subresultants
>>> from diofant.abc import x
```

```
>>> subresultants(x**2 + 1, x**2 - 1)
[x**2 + 1, x**2 - 1, -2]
```

`diofant.polys.polytools.resultant(f, g, *gens, **args)`
 Compute resultant of *f* and *g*.

Examples

```
>>> from diofant import resultant
>>> from diofant.abc import x
```

```
>>> resultant(x**2 + 1, x**2 - 1)
4
```

`diofant.polys.polytools.discriminant(f, *gens, **args)`
 Compute discriminant of *f*.

Examples

```
>>> from diofant import discriminant
>>> from diofant.abc import x
```

```
>>> discriminant(x**2 + 2*x + 3)
-8
```

`diofant.polys.dispersion.dispersion(p, q=None, *gens, **args)`

Compute the *dispersion* of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned} \text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\} \end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$. Note that we make the definition $\max\{\} := -\infty$.

See also:

[*diofant.polys.dispersion.dispersionset*](#) (page 679)

References

[R422] (page 1270), [R423] (page 1270), [R424] (page 1270), [R425] (page 1270)

Examples

```
>>> from diofant import poly
>>> from diofant.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be $-\infty$ as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`diofant.polys.dispersion.dispersionset`(*p*, *q=None*, **gens*, ***args*)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[diofant.polys.dispersion.dispersion](#) (page 677)

References

[R426] (page 1270), [R427] (page 1270), [R428] (page 1270), [R429] (page 1270)

Examples

```
>>> from diofant import poly
>>> from diofant.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
```

```
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`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.

See also:

[diofant.core.exprtools.gcd_terms](#) (page 156), [diofant.core.exprtools.factor_terms](#) (page 157)

Examples

```
>>> from diofant import terms_gcd, cos
>>> from diofant.abc import x, y
>>> terms_gcd(x**6*y**2 + x**3*y, x, y)
x**3*y*(x**3*y + 1)
```

The default action of polys routines is to expand the expression given to them. `terms_gcd` follows this behavior:

```
>>> terms_gcd((3+3*x)*(x+x*y))
3*x*(x*y + x + y + 1)
```

If this is not desired then the hint `expand` can be set to False. In this case the expression will be treated as though it were comprised of one or more terms:

```
>>> terms_gcd((3+3*x)*(x+x*y), expand=False)
(3*x + 3)*(x*y + x)
```

In order to traverse factors of a Mul or the arguments of other functions, the deep hint can be used:

```
>>> terms_gcd((3 + 3*x)*(x + x*y), expand=False, deep=True)
3*x*(x + 1)*(y + 1)
>>> terms_gcd(cos(x + x*y), deep=True)
cos(x*(y + 1))
```

Rationals are factored out by default:

```
>>> terms_gcd(x + y/2)
(2*x + y)/2
```

Only the y-term had a coefficient that was a fraction; if one does not want to factor out the 1/2 in cases like this, the flag `clear` can be set to `False`:

```
>>> terms_gcd(x + y/2, clear=False)
x + y/2
>>> terms_gcd(x*y/2 + y**2, clear=False)
y*(x/2 + y)
```

The `clear` flag is ignored if all coefficients are fractions:

```
>>> terms_gcd(x/3 + y/2, clear=False)
(2*x + 3*y)/6
```

`diofant.polys.polytools.cofactors(f, g, *gens, **args)`
Compute GCD and cofactors of `f` and `g`.

Returns polynomials (`h,cff,cfg`) such that $h = \text{gcd}(f,g)$, and $\text{cff} = \text{quo}(f,h)$ and $\text{cfg} = \text{quo}(g,h)$ are, so called, cofactors of `f` and `g`.

Examples

```
>>> from diofant import cofactors
>>> from diofant.abc import x
```

```
>>> cofactors(x**2 - 1, x**2 - 3*x + 2)
(x - 1, x + 1, x - 2)
```

`diofant.polys.polytools.gcd(f, g=None, *gens, **args)`
Compute GCD of `f` and `g`.

Examples

```
>>> from diofant import gcd
>>> from diofant.abc import x
```

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

`diofant.polys.polytools.gcd_list(seq, *gens, **args)`
Compute GCD of a list of polynomials.

Examples

```
>>> from diofant import gcd_list
>>> from diofant.abc import x
```

```
>>> gcd_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x - 1
```

`diofant.polys.polytools.lcm(f, g=None, *gens, **args)`
Compute LCM of f and g.

Examples

```
>>> from diofant import lcm
>>> from diofant.abc import x
```

```
>>> lcm(x**2 - 1, x**2 - 3*x + 2)
x**3 - 2*x**2 - x + 2
```

`diofant.polys.polytools.lcm_list(seq, *gens, **args)`
Compute LCM of a list of polynomials.

Examples

```
>>> from diofant import lcm_list
>>> from diofant.abc import x
```

```
>>> lcm_list([x**3 - 1, x**2 - 1, x**2 - 3*x + 2])
x**5 - x**4 - 2*x**3 - x**2 + x + 2
```

`diofant.polys.polytools.trunc(f, p, *gens, **args)`
Reduce f modulo a constant p.

Examples

```
>>> from diofant import trunc
>>> from diofant.abc import x
```

```
>>> trunc(2*x**3 + 3*x**2 + 5*x + 7, 3)
-x**3 - x + 1
```

`diofant.polys.polytools.monic(f, *gens, **args)`
Divide all coefficients of f by LC(f).

Examples

```
>>> from diofant import monic
>>> from diofant.abc import x
```

```
>>> monic(3*x**2 + 4*x + 2)
x**2 + 4*x/3 + 2/3
```

`diofant.polys.polytools.content(f, *gens, **args)`
 Compute GCD of coefficients of `f`.

Examples

```
>>> from diofant import content
>>> from diofant.abc import x
```

```
>>> content(6*x**2 + 8*x + 12)
2
```

`diofant.polys.polytools.primitive(f, *gens, **args)`
 Compute content and the primitive form of `f`.

Examples

```
>>> from diofant.polys.polytools import primitive
>>> from diofant.abc import x
```

```
>>> primitive(6*x**2 + 8*x + 12)
(2, 3*x**2 + 4*x + 6)
```

```
>>> eq = (2 + 2*x)*x + 2
```

Expansion is performed by default:

```
>>> primitive(eq)
(2, x**2 + x + 1)
```

Set `expand` to `False` to shut this off. Note that the extraction will not be recursive; use the `as_content_primitive` method for recursive, non-destructive Rational extraction.

```
>>> primitive(eq, expand=False)
(1, x*(2*x + 2) + 2)
```

```
>>> eq.as_content_primitive()
(2, x*(x + 1) + 1)
```

`diofant.polys.polytools.compose(f, g, *gens, **args)`
 Compute functional composition `f(g)`.

Examples

```
>>> from diofant import compose
>>> from diofant.abc import x
```

```
>>> compose(x**2 + x, x - 1)
x**2 - x
```

`diofant.polys.polytools.decompose(f, *gens, **args)`
Compute functional decomposition of f .

Examples

```
>>> from diofant import decompose
>>> from diofant.abc import x
```

```
>>> decompose(x**4 + 2*x**3 - x - 1)
[x**2 - x - 1, x**2 + x]
```

`diofant.polys.polytools.sturm(f, *gens, **args)`
Compute Sturm sequence of f .

Examples

```
>>> from diofant import sturm
>>> from diofant.abc import x
```

```
>>> sturm(x**3 - 2*x**2 + x - 3)
[x**3 - 2*x**2 + x - 3, 3*x**2 - 4*x + 1, 2*x/9 + 25/9, -2079/4]
```

`diofant.polys.polytools.gff_list(f, *gens, **args)`
Compute a list of greatest factorial factors of f .

Examples

```
>>> from diofant import gff_list, ff
>>> from diofant.abc import x
```

```
>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

```
>>> gff_list(f)
[(x, 1), (x + 2, 4)]
```

```
>>> (ff(x, 1)*ff(x + 2, 4)).expand() == f
True
```

`diofant.polys.polytools.gff(f, *gens, **args)`
Compute greatest factorial factorization of f .

`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

```
>>> from diofant import sqf_norm, sqrt
>>> from diofant.abc import x
```

```
>>> sqf_norm(x**2 + 1, extension=[sqrt(3)])
(1, x**2 - 2*sqrt(3)*x + 4, x**4 - 4*x**2 + 16)
```

`diofant.polys.polytools.sqf_part(f, *gens, **args)`
Compute square-free part of f.

Examples

```
>>> from diofant import sqf_part
>>> from diofant.abc import x
```

```
>>> sqf_part(x**3 - 3*x - 2)
x**2 - x - 2
```

`diofant.polys.polytools.sqf_list(f, *gens, **args)`
Compute a list of square-free factors of f.

Examples

```
>>> from diofant import sqf_list
>>> from diofant.abc import x
```

```
>>> sqf_list(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
(2, [(x + 1, 2), (x + 2, 3)])
```

`diofant.polys.polytools.sqf(f, *gens, **args)`
Compute square-free factorization of f.

Examples

```
>>> from diofant import sqf
>>> from diofant.abc import x
```

```
>>> sqf(2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16)
2*(x + 1)**2*(x + 2)**3
```

`diofant.polys.polytools.factor_list(f, *gens, **args)`
Compute a list of irreducible factors of f.

Examples

```
>>> from diofant import factor_list
>>> from diofant.abc import x, y
```

```
>>> factor_list(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
(2, [(x + y, 1), (x**2 + 1, 2)])
```

`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 690) and generators are not specified, then the former mode is used. Otherwise, the formal mode is used.

In symbolic mode, `factor()` (page 685) will traverse the expression tree and factor its components without any prior expansion, unless an instance of `Add` (page 107) is encountered (in this case formal factorization is used). This way `factor()` (page 685) can handle large or symbolic exponents.

By default, the factorization is computed over the rationals. To factor over other domain, e.g. an algebraic or finite field, use appropriate options: `extension`, `modulus` or `domain`.

See also:

`diofant.ntheory.factor_.factorint` (page 261)

Examples

```
>>> from diofant import factor, sqrt
>>> from diofant.abc import x, y
```

```
>>> factor(2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y)
2*(x + y)*(x**2 + 1)**2
```

```
>>> factor(x**2 + 1)
x**2 + 1
>>> factor(x**2 + 1, modulus=2)
(x + 1)**2
>>> factor(x**2 + 1, gaussian=True)
(x - I)*(x + I)
```

```
>>> factor(x**2 - 2, extension=sqrt(2))
(x - sqrt(2))*(x + sqrt(2))
```

```
>>> factor((x**2 - 1)/(x**2 + 4*x + 4))
(x - 1)*(x + 1)/(x + 2)**2
>>> factor((x**2 + 4*x + 4)**10000000*(x**2 + 1))
(x + 2)**20000000*(x**2 + 1)
```

By default, `factor` deals with an expression as a whole:

```
>>> eq = 2*(x**2 + 2*x + 1)
>>> factor(eq)
2*(x**2 + 2*x + 1)
```

If the `deep` flag is `True` then subexpressions will be factored:

```
>>> factor(eq, deep=True)
2**((x + 1)**2)
```

`diofant.polys.polytools.intervals`(*F*, *all=False*, *eps=None*, *inf=None*, *sup=None*, *strict=False*, *fast=False*, *sqf=False*)
 Compute isolating intervals for roots of *f*.

Examples

```
>>> from diofant import intervals
>>> from diofant.abc import x
```

```
>>> intervals(x**2 - 3)
[((-2, -1), 1), ((1, 2), 1)]
>>> intervals(x**2 - 3, eps=1e-2)
[(-26/15, -19/11), 1), ((19/11, 26/15), 1)]
```

`diofant.polys.polytools.refine_root`(*f*, *s*, *t*, *eps=None*, *steps=None*, *fast=False*, *check_sqf=False*)
 Refine an isolating interval of a root to the given precision.

Examples

```
>>> from diofant import refine_root
>>> from diofant.abc import x
```

```
>>> refine_root(x**2 - 3, 1, 2, eps=1e-2)
(19/11, 26/15)
```

`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

```
>>> from diofant import count_roots, I
>>> from diofant.abc import x
```

```
>>> count_roots(x**4 - 4, -3, 3)
2
>>> count_roots(x**4 - 4, 0, 1 + 3*I)
1
```

`diofant.polys.polytools.real_roots`(*f*, *multiple=True*)
 Return a list of real roots with multiplicities of *f*.

Examples

```
>>> from diofant import real_roots
>>> from diofant.abc import x
```

```
>>> real_roots(2*x**3 - 7*x**2 + 4*x + 4)
[-1/2, 2, 2]
```

`diofant.polys.polytools.nroots(f, n=15, maxsteps=50, cleanup=True)`
Compute numerical approximations of roots of f .

Examples

```
>>> from diofant import nroots
>>> from diofant.abc import x
```

```
>>> nroots(x**2 - 3, n=15)
[-1.73205080756888, 1.73205080756888]
>>> nroots(x**2 - 3, n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

`diofant.polys.polytools.ground_roots(f, *gens, **args)`
Compute roots of f by factorization in the ground domain.

Examples

```
>>> from diofant import ground_roots
>>> from diofant.abc import x
```

```
>>> ground_roots(x**6 - 4*x**4 + 4*x**3 - x**2)
{0: 2, 1: 2}
```

`diofant.polys.polytools.nth_power_roots_poly(f, n, *gens, **args)`
Construct a polynomial with n -th powers of roots of f .

Examples

```
>>> from diofant import nth_power_roots_poly, factor, roots
>>> from diofant.abc import x
```

```
>>> f = x**4 - x**2 + 1
>>> g = factor(nth_power_roots_poly(f, 2))
```

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

```
>>> R_f = [ (r**2).expand() for r in roots(f) ]
>>> R_g = roots(g).keys()
```

```
>>> set(R_f) == set(R_g)
True
```

`diofant.polys.polytools.cancel(f, *gens, **args)`
Cancel common factors in a rational function f .

Examples

```
>>> from diofant import cancel, sqrt, Symbol
>>> from diofant.abc import x
>>> A = Symbol('A', commutative=False)
```

```
>>> cancel((2*x**2 - 2)/(x**2 - 2*x + 1))
(2*x + 2)/(x - 1)
>>> cancel((sqrt(3) + sqrt(15)*A)/(sqrt(2) + sqrt(10)*A))
sqrt(6)/2
```

`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

```
>>> from diofant import reduced
>>> from diofant.abc import x, y
```

```
>>> reduced(2*x**4 + y**2 - x**2 + y**3, [x**3 - x, y**3 - y])
([2*x, 1], x**2 + y**2 + y)
```

`diofant.polys.polytools.groebner(F, *gens, **args)`

Computes the reduced Groebner basis for a set of polynomials.

Use the order argument to set the monomial ordering that will be used to compute the basis. Allowed orders are `lex`, `grlex` and `grevlex`. If no order is specified, it defaults to `lex`.

For more information on Groebner bases, see the references and the docstring of `solve_poly_system()`.

References

[R430] (page 1270), [R431] (page 1270)

Examples

Example taken from [1].

```
>>> from diofant import groebner
>>> from diofant.abc import x, y
```

```
>>> F = [x*y - 2*y, 2*y**2 - x**2]
```

```
>>> groebner(F, x, y, order='lex')
GroebnerBasis([x**2 - 2*y**2, x*y - 2*y, y**3 - 2*y], x, y,
              domain='ZZ', order='lex')
>>> groebner(F, x, y, order='grlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grlex')
>>> groebner(F, x, y, order='grevlex')
GroebnerBasis([y**3 - 2*y, x**2 - 2*y**2, x*y - 2*y], x, y,
              domain='ZZ', order='grevlex')
```

By default, an improved implementation of the Buchberger algorithm is used. Optionally, an implementation of the F5B algorithm can be used. The algorithm can be set using method flag or with the function `setup()` (page 864):

```
>>> F = [x**2 - x - 1, (2*x - 1) * y - (x**10 - (1 - x)**10)]
```

```
>>> groebner(F, x, y, method='buchberger')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
>>> groebner(F, x, y, method='f5b')
GroebnerBasis([x**2 - x - 1, y - 55], x, y, domain='ZZ', order='lex')
```

`diofant.polys.polytools.is_zero_dimensional(F, *gens, **args)`

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of F is bounded.

References

[R432] (page 1270)

class `diofant.polys.polytools.Poly`

Generic class for representing polynomial expressions.

EC(*order=None*)

Returns the last non-zero coefficient of `self`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).EC()
3
```

EM(*order=None*)

Returns the last non-zero monomial of `self`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).EM()
x**0*y**1
```

ET(*order=None*)
Returns the last non-zero term of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).ET()
(x**0*y**1, 3)
```

LC(*order=None*)
Returns the leading coefficient of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(4*x**3 + 2*x**2 + 3*x, x).LC()
4
```

LM(*order=None*)
Returns the leading monomial of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LM()
x**2*y**0
```

LT(*order=None*)
Returns the leading term of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(4*x**2 + 2*x*y**2 + x*y + 3*y, x, y).LT()
(x**2*y**0, 4)
```

TC()
Returns the trailing coefficient of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).TC()
0
```

abs()

Make all coefficients in self positive.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).abs()
Poly(x**2 + 1, x, domain='ZZ')
```

add(*other*)

Add two polynomials self and other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).add(Poly(x - 2, x))
Poly(x**2 + x - 1, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) + Poly(x - 2, x)
Poly(x**2 + x - 1, x, domain='ZZ')
```

add_ground(*coeff*)

Add an element of the ground domain to self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x + 1).add_ground(2)
Poly(x + 3, x, domain='ZZ')
```

all_coeffs()

Returns all coefficients from a univariate polynomial self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_coeffs()
[1, 0, 2, -1]
```

`all_monoms()`

Returns all monomials from a univariate polynomial `self`.

See also:

[all_terms](#) (page 693)

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_monoms()
[(3,), (2,), (1,), (0,)]
```

`all_roots(multiple=True, radicals=True)`

Return a list of real and complex roots with multiplicities.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).all_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).all_roots()
[RootOf(x**3 + x + 1, 0),
 RootOf(x**3 + x + 1, 1),
 RootOf(x**3 + x + 1, 2)]
```

`all_terms()`

Returns all terms from a univariate polynomial `self`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x - 1, x).all_terms()
[((3,), 1), ((2,), 0), ((1,), 2), ((0,), -1)]
```

`args`

Don't mess up with the core.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).args
(x**2 + 1, x)
```

as_dict(*native=False*, *zero=False*)
Switch to a dict representation.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 - y, x, y).as_dict()
{(0, 1): -1, (1, 2): 2, (2, 0): 1}
```

as_expr(**gens*)
Convert a Poly instance to an Expr instance.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = Poly(x**2 + 2*x*y**2 - y, x, y)
```

```
>>> f.as_expr()
x**2 + 2*x*y**2 - y
>>> f.as_expr({x: 5})
10*y**2 - y + 25
>>> f.as_expr(5, 6)
379
```

as_list(*native=False*)
Switch to a list representation.

cancel(*other*, *include=False*)
Cancel common factors in a rational function *self/other*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x))
(1, Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

```
>>> Poly(2*x**2 - 2, x).cancel(Poly(x**2 - 2*x + 1, x), include=True)
(Poly(2*x + 2, x, domain='ZZ'), Poly(x - 1, x, domain='ZZ'))
```

clear_denoms(*convert=False*)

Clear denominators, but keep the ground domain.

Examples

```
>>> from diofant import Poly, Rational, QQ
>>> from diofant.abc import x
```

```
>>> f = Poly(x/2 + Rational(1, 3), x, domain=QQ)
```

```
>>> f.clear_denoms()
(6, Poly(3*x + 2, x, domain='QQ'))
>>> f.clear_denoms(convert=True)
(6, Poly(3*x + 2, x, domain='ZZ'))
```

coeff_monomial(*monom*)

Returns the coefficient of *monom* in *self* if there, else *None*.

See also:

nth (page 715) more efficient query using exponents of the monomial's generators

Examples

```
>>> from diofant import Poly, exp
>>> from diofant.abc import x, y
```

```
>>> p = Poly(24*x*y*exp(8) + 23*x, x, y)
```

```
>>> p.coeff_monomial(x)
23
>>> p.coeff_monomial(y)
0
>>> p.coeff_monomial(x*y)
24*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.

See also:

all_coeffs (page 692), *coeff_monomial* (page 695), *nth* (page 715)

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 + 2*x + 3, x).coeffs()
[1, 2, 3]
```

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 $h = \text{gcd}(f,g)$, and $cff = \text{quo}(f,h)$ and $cfg = \text{quo}(g,h)$ are, so called, cofactors of *f* and *g*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).cofactors(Poly(x**2 - 3*x + 2, x))
(Poly(x - 1, x, domain='ZZ'),
 Poly(x + 1, x, domain='ZZ'),
 Poly(x - 2, x, domain='ZZ'))
```

compose(*other*)

Computes the functional composition of *self* and *other*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + x, x).compose(Poly(x - 1, x))
Poly(x**2 - x, x, domain='ZZ')
```

content()

Returns the GCD of polynomial coefficients.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(6*x**2 + 8*x + 12, x).content()
2
```

count_roots(*inf=None, sup=None*)

Return the number of roots of self in [inf, sup] interval.

Examples

```
>>> from diofant import Poly, I
>>> from diofant.abc import x
```

```
>>> Poly(x**4 - 4, x).count_roots(-3, 3)
2
>>> Poly(x**4 - 4, x).count_roots(0, 1 + 3*I)
1
```

decompose()

Computes a functional decomposition of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**4 + 2*x**3 - x - 1, x, domain='ZZ').decompose()
[Poly(x**2 - x - 1, x, domain='ZZ'), Poly(x**2 + x, x, domain='ZZ')]
```

deflate()

Reduce degree of self by mapping x_i^{**m} to y_i .

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3 + 1, x, y).deflate()
((3, 2), Poly(x**2*y + x + 1, x, y, domain='ZZ'))
```

degree(*gen=0*)

Returns degree of self in x_j .

The degree of 0 is negative infinity.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree()
2
>>> Poly(x**2 + y*x + y, x, y).degree(y)
1
```

```
>>> Poly(0, x).degree()
-oo
```

degree_list()

Returns a list of degrees of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).degree_list()
(2, 1)
```

diff(*specs, **kwargs)

Computes partial derivative of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).diff()
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x*y**2 + x, x, y).diff((0, 0), (1, 1))
Poly(2*x*y, x, y, domain='ZZ')
```

discriminant()

Computes the discriminant of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 2*x + 3, x).discriminant()
-8
```

dispersion(other=None)

Compute the *dispersion* of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned} \text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\} \end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$.

See also:

[*diofant.polys.polytools.Poly.dispersionset*](#) (page 699)

References

[R433] (page 1270), [R434] (page 1270), [R435] (page 1270), [R436] (page 1270)

Examples

```
>>> from diofant import poly
>>> from diofant.polys.dispersion import dispersion, dispersionset
>>> from diofant.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +
↳ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

`dispersionset` (*other=None*)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[diofant.polys.polytools.Poly.dispersion](#) (page 698)

References

[R437] (page 1270), [R438] (page 1270), [R439] (page 1270), [R440] (page 1270)

Examples

```
>>> from diofant import poly
>>> from diofant.polys.dispersion import dispersion, dispersionset
>>> from diofant.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

Computing the dispersion also works over field extensions:

```
>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 +
↳ 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

div(other, auto=True)

Polynomial division with remainder of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x))
(Poly(1/2*x + 1, x, domain='QQ'), Poly(5, x, domain='QQ'))
```

```
>>> Poly(x**2 + 1, x).div(Poly(2*x - 4, x), auto=False)
(Poly(0, x, domain='ZZ'), Poly(x**2 + 1, x, domain='ZZ'))
```

domain

Get the ground domain of self.

eject(*gens)

Eject selected generators into the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x, y)
```

```
>>> f.eject(x)
Poly(x*y**3 + (x**2 + x)*y + 1, y, domain='ZZ[x]')
>>> f.eject(y)
Poly(y*x**2 + (y**3 + y)*x + 1, x, domain='ZZ[y]')
```

eval(x, a=None, auto=True)

Evaluate self at a in the given variable.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y, z
```

```
>>> Poly(x**2 + 2*x + 3, x).eval(2)
11
```

```
>>> Poly(2*x*y + 3*x + y + 2, x, y).eval(x, 2)
Poly(5*y + 8, y, domain='ZZ')
```

```
>>> f = Poly(2*x*y + 3*x + y + 2*z, x, y, z)
```

```
>>> f.eval({x: 2})
Poly(5*y + 2*z + 6, y, z, domain='ZZ')
>>> f.eval({x: 2, y: 5})
Poly(2*z + 31, z, domain='ZZ')
>>> f.eval({x: 2, y: 5, z: 7})
45
```

```
>>> f.eval((2, 5))
Poly(2*z + 31, z, domain='ZZ')
>>> f(2, 5)
Poly(2*z + 31, z, domain='ZZ')
```

exclude()

Remove unnecessary generators from self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import a, b, c, d, x
```

```
>>> Poly(a + x, 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

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).exquo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).exquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

exquo_ground(*coeff*)

Exact quotient of self by a an element of the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x + 4).exquo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).exquo_ground(2)
Traceback (most recent call last):
...
ExactQuotientFailed: 2 does not divide 3 in ZZ
```

factor_list()

Returns a list of irreducible factors of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list()
(2, [(Poly(x + y, x, y, domain='ZZ'), 1),
      (Poly(x**2 + 1, x, y, domain='ZZ'), 2)])
```

factor_list_include()

Returns a list of irreducible factors of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = 2*x**5 + 2*x**4*y + 4*x**3 + 4*x**2*y + 2*x + 2*y
```

```
>>> Poly(f).factor_list_include()
[(Poly(2*x + 2*y, x, y, domain='ZZ'), 1),
 (Poly(x**2 + 1, x, y, domain='ZZ'), 2)]
```

free_symbols

Free symbols of a polynomial expression.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 1).free_symbols == {x}
True
>>> Poly(x**2 + y).free_symbols == {x, y}
True
>>> Poly(x**2 + y, x).free_symbols == {x, y}
True
```

free_symbols_in_domain

Free symbols of the domain of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 1).free_symbols_in_domain == set()
True
>>> Poly(x**2 + y).free_symbols_in_domain == set()
True
>>> Poly(x**2 + y, x).free_symbols_in_domain == {y}
True
```

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

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).gcd(Poly(x**2 - 3*x + 2, x))
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

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'),
 Poly(1/5*x**2 - 6/5*x + 2, x, domain='QQ'),
 Poly(x + 1, x, domain='QQ'))
```

gen

Return the principal generator.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).gen
x
```

get_domain()

Get the ground domain of self.

get_modulus()

Get the modulus of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, modulus=2).get_modulus()
2
```

gff_list()

Computes greatest factorial factorization of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = x**5 + 2*x**4 - x**3 - 2*x**2
```

```
>>> Poly(f).gff_list()
[(Poly(x, x, domain='ZZ'), 1), (Poly(x + 2, x, domain='ZZ'), 4)]
```

ground_roots()

Compute roots of self by factorization in the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**6 - 4*x**4 + 4*x**3 - x**2).ground_roots()
{0: 2, 1: 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

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = x**4 - 2*x**3 - 6*x**2 + 12*x + 15
>>> g = x**3 + x**2 - 4*x - 4
```

```
>>> Poly(f).half_gcdex(Poly(g))
(Poly(-1/5*x + 3/5, x, domain='QQ'), Poly(x + 1, x, domain='QQ'))
```

has_only_gens(*gens)

Return True if `Poly(f, *gens)` retains ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y, z
```

```
>>> Poly(x*y + 1, x, y, z).has_only_gens(x, y)
True
>>> Poly(x*y + z, x, y, z).has_only_gens(x, y)
False
```

homogeneous_order()

Returns the homogeneous order of self.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. This degree is the homogeneous order of `f`. If you only want to check if a polynomial is homogeneous, then use [`Poly.is_homogeneous\(\)`](#) (page 709).

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = Poly(x**5 + 2*x**3*y**2 + 9*x*y**4)
>>> f.homogeneous_order()
5
```

homogenize(s)

Returns the homogeneous polynomial of self.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you only want to check if a polynomial is homogeneous, then use [`Poly.is_homogeneous\(\)`](#) (page 709). If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use [`Poly.homogeneous_order\(\)`](#) (page 706).

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y, z
```

```
>>> f = Poly(x**5 + 2*x**2*y**2 + 9*x*y**3)
>>> f.homogenize(z)
Poly(x**5 + 2*x**2*y**2*z + 9*x*y**3*z, x, y, z, domain='ZZ')
```

inject(*front=False*)

Inject ground domain generators into self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = Poly(x**2*y + x*y**3 + x*y + 1, x)
```

```
>>> f.inject()
Poly(x**2*y + x*y**3 + x*y + 1, x, y, domain='ZZ')
>>> f.inject(front=True)
Poly(y**3*x + y*x**2 + y*x + 1, y, x, domain='ZZ')
```

integrate(**specs, **args*)

Computes indefinite integral of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 2*x + 1, x).integrate()
Poly(1/3*x**3 + x**2 + x, x, domain='QQ')
```

```
>>> Poly(x*y**2 + x, x, y).integrate((0, 1), (1, 0))
Poly(1/2*x**2*y**2 + 1/2*x**2, x, y, domain='QQ')
```

intervals(*all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)

Compute isolating intervals for roots of self.

For real roots the Vincent-Akritas-Strzebonski (VAS) continued fractions method is used.

References

[R441] (page 1270), [R442] (page 1270)

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 3, x).intervals()
[(-2, -1), 1], ((1, 2), 1)]
>>> Poly(x**2 - 3, x).intervals(eps=1e-2)
[(-26/15, -19/11), 1], ((19/11, 26/15), 1)]
```

invert(*other*, *auto=True*)
 Invert self modulo other when possible.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).invert(Poly(2*x - 1, x))
Poly(-4/3, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).invert(Poly(x - 1, x))
Traceback (most recent call last):
...
NotInvertible: zero divisor
```

is_cyclotomic
 Returns True if self is a cyclotomic polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> Poly(f).is_cyclotomic
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> Poly(g).is_cyclotomic
True
```

is_ground
 Returns True if self is an element of the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```



```
>>> Poly(x, x).is_ground
False
>>> Poly(2, x).is_ground
True
>>> Poly(y, x).is_ground
True
```

is_homogeneous

Returns True if self is a homogeneous polynomial.

A homogeneous polynomial is a polynomial whose all monomials with non-zero coefficients have the same total degree. If you want not only to check if a polynomial is homogeneous but also compute its homogeneous order, then use [Poly.homogeneous_order\(\)](#) (page 706).

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + x*y, x, y).is_homogeneous
True
>>> Poly(x**3 + x*y, x, y).is_homogeneous
False
```

is_irreducible

Returns True if self has no factors over its domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + x + 1, x, modulus=2).is_irreducible
True
>>> Poly(x**2 + 1, x, modulus=2).is_irreducible
False
```

is_linear

Returns True if self is linear in all its variables.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x + y + 2, x, y).is_linear
True
>>> Poly(x*y + 2, x, y).is_linear
False
```

is_monic

Returns True if the leading coefficient of self is one.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x + 2, x).is_monic
True
>>> Poly(2*x + 2, x).is_monic
False
```

is_monomial

Returns True if self is zero or has only one term.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(3*x**2, x).is_monomial
True
>>> Poly(3*x**2 + 1, x).is_monomial
False
```

is_multivariate

Returns True if self is a multivariate polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_multivariate
False
>>> Poly(x*y**2 + x*y + 1, x, y).is_multivariate
True
>>> Poly(x*y**2 + x*y + 1, x).is_multivariate
False
>>> Poly(x**2 + x + 1, x, y).is_multivariate
True
```

is_one

Returns True if self is a unit polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(0, x).is_one
False
>>> Poly(1, x).is_one
True
```

is_primitive

Returns True if GCD of the coefficients of self is one.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**2 + 6*x + 12, x).is_primitive
False
>>> Poly(x**2 + 3*x + 6, x).is_primitive
True
```

is_quadratic

Returns True if self is quadratic in all its variables.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x*y + 2, x, y).is_quadratic
True
>>> Poly(x*y**2 + 2, x, y).is_quadratic
False
```

is_sqf

Returns True if self is a square-free polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).is_sqf
False
>>> Poly(x**2 - 1, x).is_sqf
True
```

is_univariate

Returns True if self is a univariate polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + x + 1, x).is_univariate
True
>>> Poly(x*y**2 + x*y + 1, x, y).is_univariate
False
>>> Poly(x*y**2 + x*y + 1, x).is_univariate
True
>>> Poly(x**2 + x + 1, x, y).is_univariate
False
```

`is_zero`

Returns True if self is a zero polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(0, x).is_zero
True
>>> Poly(1, x).is_zero
False
```

`l1_norm()`

Returns l1 norm of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).l1_norm()
6
```

`lcm(other)`

Returns polynomial LCM of self and other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).lcm(Poly(x**2 - 3*x + 2, x))
Poly(x**3 - 2*x**2 - x + 2, x, domain='ZZ')
```

`length()`

Returns the number of non-zero terms in self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 2*x - 1).length()
3
```

`lift()`

Convert algebraic coefficients to rationals.

Examples

```
>>> from diofant import Poly, I
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + I*x + 1, x, extension=I).lift()
Poly(x**4 + 3*x**2 + 1, x, domain='QQ')
```

`ltrim(gen)`

Remove dummy generators from the “left” of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y, z
```

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

`max_norm()`

Returns maximum norm of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(-x**2 + 2*x - 3, x).max_norm()
3
```

`monic(auto=True)`

Divides all coefficients by $LC(f)$.

Examples

```
>>> from diofant import Poly, ZZ
>>> from diofant.abc import x
```

```
>>> Poly(3*x**2 + 6*x + 9, x, domain=ZZ).monic()
Poly(x**2 + 2*x + 3, x, domain='QQ')
```

```
>>> Poly(3*x**2 + 4*x + 2, x, domain=ZZ).monic()
Poly(x**2 + 4/3*x + 2/3, x, domain='QQ')
```

monoms (*order=None*)

Returns all non-zero monomials from *self* in lex order.

See also:

[*all_monoms*](#) (page 693)

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).monoms()
[(2, 0), (1, 2), (1, 1), (0, 1)]
```

mul (*other*)

Multiply two polynomials *self* and *other*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).mul(Poly(x - 2, x))
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x)*Poly(x - 2, x)
Poly(x**3 - 2*x**2 + x - 2, x, domain='ZZ')
```

mul_ground (*coeff*)

Multiply *self* by a an element of the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x + 1).mul_ground(2)
Poly(2*x + 2, x, domain='ZZ')
```

neg ()

Negate all coefficients in *self*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).neg()
Poly(-x**2 + 1, x, domain='ZZ')
```

```
>>> -Poly(x**2 - 1, x)
Poly(-x**2 + 1, x, domain='ZZ')
```

classmethod `new(rep, *gens)`

Construct `Poly` (page 690) 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

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 3).nroots(n=15)
[-1.73205080756888, 1.73205080756888]
>>> Poly(x**2 - 3).nroots(n=30)
[-1.73205080756887729352744634151, 1.73205080756887729352744634151]
```

nth (`*N`)

Returns the n-th coefficient of `f` where `N` are the exponents of the generators in the term of interest.

See also:

`coeff_monomial` (page 695)

Examples

```
>>> from diofant import Poly, sqrt
>>> from diofant.abc import x, y
```

```
>>> Poly(x**3 + 2*x**2 + 3*x, x).nth(2)
2
>>> Poly(x**3 + 2*x*y**2 + y**2, x, y).nth(1, 2)
2
>>> Poly(4*sqrt(x)*y)
Poly(4*y*sqrt(x), y, sqrt(x), domain='ZZ')
```

```
>>> _.nth(1, 1)
4
```

nth_power_roots_poly(*n*)

Construct a polynomial with *n*-th powers of roots of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = Poly(x**4 - x**2 + 1)
```

```
>>> f.nth_power_roots_poly(2)
Poly(x**4 - 2*x**3 + 3*x**2 - 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(3)
Poly(x**4 + 2*x**2 + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(4)
Poly(x**4 + 2*x**3 + 3*x**2 + 2*x + 1, x, domain='ZZ')
>>> f.nth_power_roots_poly(12)
Poly(x**4 - 4*x**3 + 6*x**2 - 4*x + 1, x, domain='ZZ')
```

one

Return one polynomial with self's properties.

pdiv(*other*)

Polynomial pseudo-division of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).pdiv(Poly(2*x - 4, x))
(Poly(2*x + 4, x, domain='ZZ'), Poly(20, x, domain='ZZ'))
```

per(*rep*, *gens=None*, *remove=None*)

Create a Poly out of the given representation.

Examples

```
>>> from diofant import Poly, ZZ
>>> from diofant.abc import x, y
```

```
>>> from diofant.polys.polyclasses import DMP
```

```
>>> a = Poly(x**2 + 1)
```

```
>>> a.per(DMP([ZZ(1), ZZ(1)], ZZ), gens=[y])
Poly(y + 1, y, domain='ZZ')
```


pexquo(*other*)

Polynomial exact pseudo-quotient of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 1, x).pexquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).pexquo(Poly(2*x - 4, x))
Traceback (most recent call last):
...
ExactQuotientFailed: 2*x - 4 does not divide x**2 + 1
```

pow(*n*)

Raise self to a non-negative power *n*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x - 2, x).pow(3)
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**3
Poly(x**3 - 6*x**2 + 12*x - 8, x, domain='ZZ')
```

pquo(*other*)

Polynomial pseudo-quotient of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).pquo(Poly(2*x - 4, x))
Poly(2*x + 4, x, domain='ZZ')
```

```
>>> Poly(x**2 - 1, x).pquo(Poly(2*x - 2, x))
Poly(2*x + 2, x, domain='ZZ')
```

prem(*other*)

Polynomial pseudo-remainder of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).prem(Poly(2*x - 4, x))
Poly(20, x, domain='ZZ')
```

primitive()

Returns the content and a primitive form of `self`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**2 + 8*x + 12, x).primitive()
(2, Poly(x**2 + 4*x + 6, x, domain='ZZ'))
```

quo(*other*, *auto=True*)

Computes polynomial quotient of `self` by `other`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).quo(Poly(2*x - 4, x))
Poly(1/2*x + 1, x, domain='QQ')
```

```
>>> Poly(x**2 - 1, x).quo(Poly(x - 1, x))
Poly(x + 1, x, domain='ZZ')
```

quo_ground(*coeff*)

Quotient of `self` by a an element of the ground domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x + 4).quo_ground(2)
Poly(x + 2, x, domain='ZZ')
```

```
>>> Poly(2*x + 3).quo_ground(2)
Poly(x + 1, x, domain='ZZ')
```

rat_clear_denoms(*other*)

Clear denominators in a rational function `self/other`.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> f = Poly(x**2/y + 1, x)
>>> g = Poly(x**3 + y, x)
```

```
>>> p, q = f.rat_clear_denoms(g)
```

```
>>> p
Poly(x**2 + y, x, domain='ZZ[y]')
>>> q
Poly(y*x**3 + y**2, x, domain='ZZ[y]')
```

real_roots(*multiple=True, radicals=True*)
Return a list of real roots with multiplicities.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**3 - 7*x**2 + 4*x + 4).real_roots()
[-1/2, 2, 2]
>>> Poly(x**3 + x + 1).real_roots()
[RootOf(x**3 + x + 1, 0)]
```

refine_root(*s, t, eps=None, steps=None, fast=False, check_sqf=False*)
Refine an isolating interval of a root to the given precision.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 3, x).refine_root(1, 2, eps=1e-2)
(19/11, 26/15)
```

rem(*other, auto=True*)
Computes the polynomial remainder of self by other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x))
Poly(5, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x).rem(Poly(2*x - 4, x), auto=False)
Poly(x**2 + 1, x, domain='ZZ')
```

reorder(*gens, **args)
Efficiently apply new order of generators.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

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

replace(x, y=None)
Replace x with y in generators list.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 1, x).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 677) separately.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = Poly(x**2 + 1, x)
```

```
>>> f.resultant(Poly(x**2 - 1, x))
4
>>> f.resultant(Poly(x**2 - 1, x), includePRS=True)
(4, [Poly(x**2 + 1, x, domain='ZZ'), Poly(x**2 - 1, x, domain='ZZ'),
     Poly(-2, x, domain='ZZ')])
```

retract(field=None)
Recalculate the ground domain of a polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = Poly(x**2 + 1, x, domain='QQ[y]')
>>> f
Poly(x**2 + 1, x, domain='QQ[y]')
```

```
>>> f.retract()
Poly(x**2 + 1, x, domain='ZZ')
>>> f.retract(field=True)
Poly(x**2 + 1, x, domain='QQ')
```

revert(*n*)

Compute $\text{self}^{*(-1)} \bmod x^{*n}$.

root(*index*, *radicals=True*)

Get an indexed root of a polynomial.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = Poly(2*x**3 - 7*x**2 + 4*x + 4)
```

```
>>> f.root(0)
-1/2
>>> f.root(1)
2
>>> f.root(2)
2
>>> f.root(3)
Traceback (most recent call last):
...
IndexError: root index out of [-3, 2] range, got 3
```

```
>>> Poly(x**5 + x + 1).root(0)
RootOf(x**3 - x**2 + 1, 0)
```

set_domain(*domain*)

Set the ground domain of self.

set_modulus(*modulus*)

Set the modulus of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(5*x**2 + 2*x - 1, x).set_modulus(2)
Poly(x**2 + 1, x, modulus=2)
```

shift(*a*)

Efficiently compute Taylor shift $f(x + a)$.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 - 2*x + 1, x).shift(2)
Poly(x**2 + 2*x + 1, x, domain='ZZ')
```

slice(*x*, *m*, *n=None*)

Take a continuous subsequence of terms of self.

sqf_list(*all=False*)

Returns a list of square-free factors of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f = 2*x**5 + 16*x**4 + 50*x**3 + 76*x**2 + 56*x + 16
```

```
>>> Poly(f).sqf_list()
(2, [(Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])
```

```
>>> Poly(f).sqf_list(all=True)
(2, [(Poly(1, x, domain='ZZ'), 1),
      (Poly(x + 1, x, domain='ZZ'), 2),
      (Poly(x + 2, x, domain='ZZ'), 3)])
```

sqf_list_include(*all=False*)

Returns a list of square-free factors of self.

Examples

```
>>> from diofant import Poly, expand
>>> from diofant.abc import x
```

```
>>> f = expand(2*(x + 1)**3*x**4)
>>> f
2*x**7 + 6*x**6 + 6*x**5 + 2*x**4
```

```
>>> Poly(f).sqf_list_include()
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

```
>>> Poly(f).sqf_list_include(all=True)
[(Poly(2, x, domain='ZZ'), 1),
 (Poly(1, x, domain='ZZ'), 2),
 (Poly(x + 1, x, domain='ZZ'), 3),
 (Poly(x, x, domain='ZZ'), 4)]
```

sqf_norm()

Computes square-free norm of 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

```
>>> from diofant import Poly, sqrt
>>> from diofant.abc import x
```

```
>>> s, f, r = Poly(x**2 + 1, x, extension=[sqrt(3)]).sqf_norm()
```

```
>>> s
1
>>> f
Poly(x**2 - 2*sqrt(3)*x + 4, x, domain='QQ<sqrt(3)>')
>>> r
Poly(x**4 - 4*x**2 + 16, x, domain='QQ')
```

sqf_part()

Computes square-free part of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 - 3*x - 2, x).sqf_part()
Poly(x**2 - x - 2, x, domain='ZZ')
```

sqr()

Square a polynomial self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x - 2, x).sqr()
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

```
>>> Poly(x - 2, x)**2
Poly(x**2 - 4*x + 4, x, domain='ZZ')
```

sturm(*auto=True*)
Computes the Sturm sequence of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**3 - 2*x**2 + x - 3, x).sturm()
[Poly(x**3 - 2*x**2 + x - 3, x, domain='QQ'),
 Poly(3*x**2 - 4*x + 1, x, domain='QQ'),
 Poly(2/9*x + 25/9, x, domain='QQ'),
 Poly(-2079/4, x, domain='QQ')]
```

sub(*other*)
Subtract two polynomials self and other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).sub(Poly(x - 2, x))
Poly(x**2 - x + 3, x, domain='ZZ')
```

```
>>> Poly(x**2 + 1, x) - Poly(x - 2, x)
Poly(x**2 - x + 3, x, domain='ZZ')
```

sub_ground(*coeff*)
Subtract an element of the ground domain from self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x + 1).sub_ground(2)
Poly(x - 1, x, domain='ZZ')
```

subresultants(*other*)
Computes the subresultant PRS of self and other.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x).subresultants(Poly(x**2 - 1, x))
[Poly(x**2 + 1, x, domain='ZZ'),
 Poly(x**2 - 1, x, domain='ZZ'),
 Poly(-2, x, domain='ZZ')]
```

terms(*order=None*)

Returns all non-zero terms from *self* in lex order.

See also:

all_terms (page 693)

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + 2*x*y**2 + x*y + 3*y, x, y).terms()
[((2, 0), 1), ((1, 2), 2), ((1, 1), 1), ((0, 1), 3)]
```

terms_gcd()

Remove GCD of terms from the polynomial *self*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**6*y**2 + x**3*y, x, y).terms_gcd()
((3, 1), Poly(x**3*y + 1, x, y, domain='ZZ'))
```

termwise(*func, *gens, **args*)

Apply a function to all terms of *self*.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> def func(k, coeff):
...     k = k[0]
...     return coeff//10**(2-k)
```

```
>>> Poly(x**2 + 20*x + 400).termwise(func)
Poly(x**2 + 2*x + 4, x, domain='ZZ')
```

to_exact()

Make the ground domain exact.

Examples

```
>>> from diofant import Poly, RR
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1.0, x, domain=RR).to_exact()
Poly(x**2 + 1, x, domain='QQ')
```

to_field()

Make the ground domain a field.

Examples

```
>>> from diofant import Poly, ZZ
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, x, domain=ZZ).to_field()
Poly(x**2 + 1, x, domain='QQ')
```

to_ring()

Make the ground domain a ring.

Examples

```
>>> from diofant import Poly, QQ
>>> from diofant.abc import x
```

```
>>> Poly(x**2 + 1, domain=QQ).to_ring()
Poly(x**2 + 1, x, domain='ZZ')
```

total_degree()

Returns the total degree of self.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x, y
```

```
>>> Poly(x**2 + y*x + 1, x, y).total_degree()
2
>>> Poly(x + y**5, x, y).total_degree()
5
```

trunc(p)

Reduce self modulo a constant p.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> Poly(2*x**3 + 3*x**2 + 5*x + 7, x).trunc(3)
Poly(-x**3 - x + 1, x, domain='ZZ')
```

unify(*other*)

Make self and other belong to the same domain.

Examples

```
>>> from diofant import Poly
>>> from diofant.abc import x
```

```
>>> f, g = Poly(x/2 + 1), Poly(2*x + 1)
```

```
>>> f
Poly(1/2*x + 1, x, domain='QQ')
>>> g
Poly(2*x + 1, x, domain='ZZ')
```

```
>>> F, G = f.unify(g)
```

```
>>> F
Poly(1/2*x + 1, x, domain='QQ')
>>> G
Poly(2*x + 1, x, domain='QQ')
```

unit

Return unit polynomial with self's properties.

zero

Return zero polynomial with self's properties.

class diofant.polys.polytools.**PurePoly**

Class for representing pure polynomials.

free_symbols

Free symbols of a polynomial.

Examples

```
>>> from diofant import PurePoly
>>> from diofant.abc import x, y
```

```
>>> PurePoly(x**2 + 1).free_symbols == set()
True
>>> PurePoly(x**2 + y).free_symbols == set()
True
>>> PurePoly(x**2 + y, x).free_symbols == {y}
True
```

class diofant.polys.polytools.**GroebnerBasis**

Represents a reduced Groebner basis.

contains(*poly*)

Check if *poly* belongs the ideal generated by *self*.

Examples

```
>>> from diofant import groebner
>>> from diofant.abc import x, y
```

```
>>> f = 2*x**3 + y**3 + 3*y
>>> G = groebner([x**2 + y**2 - 1, x*y - 2])
```

```
>>> G.contains(f)
True
>>> G.contains(f + 1)
False
```

fglm(*order*)

Convert a Groebner basis from one ordering to another.

The FGLM algorithm converts reduced Groebner bases of zero-dimensional ideals from one ordering to another. This method is often used when it is infeasible to compute a Groebner basis with respect to a particular ordering directly.

References

[R443] (page 1270)

Examples

```
>>> from diofant.abc import x, y
>>> from diofant import groebner
```

```
>>> F = [x**2 - 3*y - x + 1, y**2 - 2*x + y - 1]
>>> G = groebner(F, x, y, order='grlex')
```

```
>>> list(G.fglm('lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
>>> list(groebner(F, x, y, order='lex'))
[2*x - y**2 - y + 1, y**4 + 2*y**3 - 3*y**2 - 16*y + 7]
```

is_zero_dimensional

Checks if the ideal generated by a Groebner basis is zero-dimensional.

The algorithm checks if the set of monomials not divisible by the leading monomial of any element of *F* is bounded.

References

[R444] (page 1270)

reduce(*expr*, *auto=True*)

Reduces a polynomial modulo a Groebner basis.

Given a polynomial f and a set of polynomials $G = (g_1, \dots, g_n)$, computes a set of quotients $q = (q_1, \dots, q_n)$ and the remainder r such that $f = q_1*f_1 + \dots + q_n*f_n + r$, where r vanishes or r is a completely reduced polynomial with respect to G .

Examples

```
>>> from diofant import groebner, expand
>>> from diofant.abc import x, y
```

```
>>> f = 2*x**4 - x**2 + y**3 + y**2
>>> G = groebner([x**3 - x, y**3 - y])
```

```
>>> G.reduce(f)
([2*x, 1], x**2 + y**2 + y)
>>> Q, r = _
```

```
>>> expand(sum(q*g for q, g in zip(Q, G)) + r)
2*x**4 - x**2 + y**3 + y**2
>>> _ == f
True
```

Extra polynomial manipulation functions

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

```
>>> from diofant.abc import x, y
```

```
>>> symmetrize(x**2 + y**2)
(-2*x*y + (x + y)**2, 0)
```

```
>>> symmetrize(x**2 + y**2, formal=True)
(s1**2 - 2*s2, 0, [(s1, x + y), (s2, x*y)])
```

```
>>> symmetrize(x**2 - y**2)
(-2*x*y + (x + y)**2, -2*y**2)
```

```
>>> symmetrize(x**2 - y**2, formal=True)
(s1**2 - 2*s2, -2*y**2, [(s1, x + y), (s2, x*y)])
```

`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 ([1]).

References

[R445] (page 1270)

Examples

```
>>> from diofant.abc import x, y, a, b, c, d, e
```

```
>>> horner(9*x**4 + 8*x**3 + 7*x**2 + 6*x + 5)
x*(x*(x*(9*x + 8) + 7) + 6) + 5
```

```
>>> horner(a*x**4 + b*x**3 + c*x**2 + d*x + e)
e + x*(d + x*(c + x*(a*x + b)))
```

```
>>> f = 4*x**2*y**2 + 2*x**2*y + 2*x*y**2 + x*y
```

```
>>> horner(f, wrt=x)
x*(x*y*(4*y + 2) + y*(2*y + 1))
```

```
>>> horner(f, wrt=y)
y*(x*y*(4*x + 2) + x*(2*x + 1))
```

`diofant.polys.polyfuncs.interpolate`(*data*, *x*)

Construct an interpolating polynomial for the data points.

Examples

```
>>> from diofant.abc import x
```

A list is interpreted as though it were paired with a range starting from 1:

```
>>> interpolate([1, 4, 9, 16], x)
x**2
```

This can be made explicit by giving a list of coordinates:

```
>>> interpolate([(1, 1), (2, 4), (3, 9)], x)
x**2
```

The (x, y) coordinates can also be given as keys and values of a dictionary (and the points need not be equispaced):

```
>>> interpolate([(-1, 2), (1, 2), (2, 5)], x)
x**2 + 1
>>> interpolate({-1: 2, 1: 2, 2: 5}, x)
x**2 + 1
```

`diofant.polys.polyfuncs.viete(f, roots=None, *gens, **args)`
Generate Viète's formulas for f.

Examples

```
>>> from diofant import symbols
```

```
>>> x, a, b, c, r1, r2 = symbols('x,a:c,r1:3')
```

```
>>> viete(a*x**2 + b*x + c, [r1, r2], x)
[(r1 + r2, -b/a), (r1*r2, c/a)]
```

Domain constructors

`diofant.polys.constructor.construct_domain(obj, **args)`
Construct a minimal domain for the list of coefficients.

Algebraic number fields

`diofant.polys.numberfields.minimal_polynomial(ex, x=None, **args)`
Computes the minimal polynomial of an algebraic element.

Parameters `ex` : algebraic element expression

`x` : independent variable of the minimal polynomial

compose : boolean, optional

If True (default), the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. Else a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

polys : boolean, optional

if True returns a Poly object (the default is False).

domain : Domain, optional

If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> from diofant import minimal_polynomial, sqrt, solve, QQ
>>> from diofant.abc import x, y
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`diofant.polys.numberfields.minpoly`(*ex*, *x=None*, ***args*)
 Computes the minimal polynomial of an algebraic element.

Parameters *ex* : algebraic element expression

x : independent variable of the minimal polynomial

compose : boolean, optional

If True (default), the minimal polynomial of the subexpressions of *ex* are computed, then the arithmetic operations on them are performed using the resultant and factorization. Else a bottom-up algorithm is used with groebner. The default algorithm stalls less frequently.

polys : boolean, optional

if True returns a Poly object (the default is False).

domain : Domain, optional

If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> from diofant import minimal_polynomial, sqrt, solve, QQ
>>> from diofant.abc import x, y
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

`diofant.polys.numberfields.primitive_element`(*extension*, *x=None*, ***args*)
 Construct a common number field for all extensions.

`diofant.polys.numberfields.field_isomorphism`(*a*, *b*, ***args*)
 Construct an isomorphism between two number fields.

`diofant.polys.numberfields.to_number_field`(*extension*, *theta=None*, ***args*)
 Express *extension* in the field generated by *theta*.

`diofant.polys.numberfields.isolate(alg, eps=None, fast=False)`

Give a rational isolating interval for an algebraic number.

class `diofant.polys.numberfields.AlgebraicNumber`

Class for representing algebraic numbers in Diofant.

as_expr(*x=None*)

Create a Basic expression from self.

as_poly(*x=None*)

Create a Poly instance from self.

coeffs()

Returns all Diofant coefficients of an algebraic number.

is_aliased

Returns True if alias was set.

native_coeffs()

Returns all native coefficients of an algebraic number.

to_algebraic_integer()

Convert self to an algebraic integer.

Monomials encoded as tuples

class `diofant.polys.monomials.Monomial(monom, gens=None)`

Class representing a monomial, i.e. a product of powers.

`diofant.polys.monomials.itermonomials(variables, degree)`

Generate a set of monomials of the given total degree or less.

Given a set of variables V and a total degree N generate a set of monomials of degree at most N . The total number of monomials is huge and is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

For example if we would like to generate a dense polynomial of a total degree $N = 50$ in 5 variables, assuming that exponents and all of coefficients are 32-bit long and stored in an array we would need almost 80 GiB of memory! Fortunately most polynomials, that we will encounter, are sparse.

Examples

Consider monomials in variables x and y :

```
>>> from diofant.polys.orderings import monomial_key
>>> from diofant.abc import x, y

>>> sorted(itermonomials([x, y], 2), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]

>>> sorted(itermonomials([x, y], 3), key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2, x**3, x**2*y, x*y**2, y**3]
```

`diofant.polys.monomials.monomial_count(V , N)`

Computes the number of monomials.

The number of monomials is given by the following formula:

$$\frac{(\#V + N)!}{\#V!N!}$$

where N is a total degree and V is a set of variables.

Examples

```
>>> from diofant.polys.orderings import monomial_key
>>> from diofant.abc import x, y
```

```
>>> monomial_count(2, 2)
6
```

```
>>> M = itermonomials([x, y], 2)
```

```
>>> sorted(M, key=monomial_key('grlex', [y, x]))
[1, x, y, x**2, x*y, y**2]
>>> len(M)
6
```

Orderings of monomials

class diofant.polys.orderings.**LexOrder**
Lexicographic order of monomials.

class diofant.polys.orderings.**GradedLexOrder**
Graded lexicographic order of monomials.

class diofant.polys.orderings.**ReversedGradedLexOrder**
Reversed graded lexicographic order of monomials.

Formal manipulation of roots of polynomials

class diofant.polys.rootoftools.**RootOf**
Represents k -th root of a univariate polynomial.

class diofant.polys.rootoftools.**RootSum**
Represents a sum of all roots of a univariate polynomial.

Symbolic root-finding algorithms

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 [1]) the `trig` flag can be set to `True` to have the solutions returned in terms of cosine and inverse cosine functions.

To get roots from a specific domain set the `filter` flag with one of the following specifiers: `Z`, `Q`, `R`, `I`, `C`. By default all roots are returned (this is equivalent to setting `filter='C'`).

By default a dictionary is returned giving a compact result in case of multiple roots. However to get a list containing all those roots set the `multiple` flag to `True`; the list will have identical roots appearing next to each other in the result. (For a given `Poly`, the `all_roots` method will give the roots in sorted numerical order.)

References

[R446] (page 1270)

Examples

```
>>> from diofant import Poly, roots, sqrt
>>> from diofant.abc import x, y
```

```
>>> roots(x**2 - 1, x) == {-1: 1, 1: 1}
True
```

```
>>> p = Poly(x**2-1, x)
>>> roots(p) == {-1: 1, 1: 1}
True
```

```
>>> p = Poly(x**2-y, x, y)
```

```
>>> roots(Poly(p, x)) == {-sqrt(y): 1, sqrt(y): 1}
True
```

```
>>> roots(x**2 - y, x) == {-sqrt(y): 1, sqrt(y): 1}
True
```

```
>>> roots([1, 0, -1]) == {-1: 1, 1: 1}
True
```

Special polynomials

`diofant.polys.specialpolys.swinnerton_dyer_poly(n, x=None, **args)`
Generates *n*-th Swinnerton-Dyer polynomial in *x*.

`diofant.polys.specialpolys.interpolating_poly(n, x, X='x', Y='y')`
Construct Lagrange interpolating polynomial for *n* data points.

`diofant.polys.specialpolys.cyclotomic_poly(n, x=None, **args)`
Generates cyclotomic polynomial of order *n* in *x*.

`diofant.polys.specialpolys.symmetric_poly(n, *gens, **args)`
Generates symmetric polynomial of order *n*.

`diofant.polys.specialpolys.random_poly(x, n, inf, sup, domain=ZZ, polys=False)`
Return a polynomial of degree n with coefficients in $[inf, sup]$.

Orthogonal polynomials

`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.legendre_poly(n, x=None, **args)`
Generates Legendre 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.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

```
>>> from diofant import Symbol
```

```
>>> z = Symbol("z")
>>> 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
```

Manipulation of rational functions

`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 688). However, `together()` (page 736) 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 736) is a complement to `apart()` (page 737), so `apart(together(expr))` should return `expr` unchanged. Note however, that `together()` (page 736) uses only symbolic methods, so it might be necessary to use `cancel()` (page 688) to perform algebraic simplification and minimise degree of the numerator and denominator.

Examples

```
>>> from diofant import together, exp
>>> from diofant.abc import x, y, z
```

```
>>> together(1/x + 1/y)
(x + y)/(x*y)
>>> together(1/x + 1/y + 1/z)
(x*y + x*z + y*z)/(x*y*z)
```

```
>>> together(1/(x*y) + 1/y**2)
(x + y)/(x*y**2)
```

```
>>> together(1/(1 + 1/x) + 1/(1 + 1/y))
(x*(y + 1) + y*(x + 1))/((x + 1)*(y + 1))
```

```
>>> together(exp(1/x + 1/y))
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
```

Partial fraction decomposition

`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).

See also:

[apart_list](#) (page 738), [assemble_partfrac_list](#) (page 740)

Examples

```
>>> from diofant.abc import x, y
```

By default, using the undetermined coefficients method:

```
>>> apart(y/(x + 2)/(x + 1), x)
-y/(x + 2) + y/(x + 1)
```

The undetermined coefficients method does not provide a result when the denominators roots are not rational:

```
>>> apart(y/(x**2 + x + 1), x)
y/(x**2 + x + 1)
```

You can choose Bronstein's algorithm by setting `full=True`:

```
>>> apart(y/(x**2 + x + 1), x, full=True)
RootSum(_w**2 + _w + 1, Lambda(_a, (-2*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)
```

`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`.

See also:

`apart` (page 737), `assemble_partfrac_list` (page 740)

References

[R447] (page 1270)

Examples

A first example:

```
>>> from diofant.abc import x, t
```

```
>>> f = (2*x**3 - 2*x) / (x**2 - 2*x + 1)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(2*x + 4, x, domain='ZZ'),
[(Poly(_w - 1, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, 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)
```

`diofant.polys.partfrac.assemble_partfrac_list`(*partial_list*)

Reassemble a full partial fraction decomposition from a structured result obtained by the function `apart_list`.

See also:

[apart](#) (page 737), [apart_list](#) (page 738)

Examples

This example is taken from Bronstein's original paper:

```
>>> from diofant.abc import x, y
```

```
>>> f = 36 / (x**5 - 2*x**4 - 2*x**3 + 4*x**2 + x - 2)
>>> pfd = apart_list(f)
>>> pfd
(1,
Poly(0, x, domain='ZZ'),
[(Poly(_w - 2, _w, domain='ZZ'), Lambda(_a, 4), Lambda(_a, 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)))
```

```
>>> from diofant import Dummy, Poly, Lambda, sqrt
>>> a = Dummy("a")
>>> pfd = (1, Poly(0, x, domain='ZZ'), [[sqrt(2), -sqrt(2)], Lambda(a, a/2),
↳ Lambda(a, -a + x), 1])
```

```
>>> assemble_partfrac_list(pfd)
-sqrt(2)/(2*(x + sqrt(2))) + sqrt(2)/(2*(x - sqrt(2)))
```

Dispersion of Polynomials

`diofant.polys.dispersion.dispersionset`(*p*, *q=None*, **gens*, ***args*)

Compute the *dispersion set* of two polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion set $J(f, g)$ is defined as:

$$\begin{aligned} J(f, g) &:= \{a \in \mathbb{N}_0 \mid \gcd(f(x), g(x+a)) \neq 1\} \\ &= \{a \in \mathbb{N}_0 \mid \deg \gcd(f(x), g(x+a)) \geq 1\} \end{aligned}$$

For a single polynomial one defines $J(f) := J(f, f)$.

See also:

[diofant.polys.dispersion.dispersion](#) (page 677)

References

[R448] (page 1270), [R449] (page 1270), [R450] (page 1270), [R451] (page 1271)

Examples

```
>>> from diofant import poly
>>> from diofant.abc import x
```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```

>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo

```

Computing the dispersion also works over field extensions:

```

>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]

```

We can even perform the computations for polynomials having symbolic coefficients:

```

>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]

```

`diofant.polys.dispersion.dispersion(p, q=None, *gens, **args)`

Compute the *dispersion* of polynomials.

For two polynomials $f(x)$ and $g(x)$ with $\deg f > 0$ and $\deg g > 0$ the dispersion $\text{dis}(f, g)$ is defined as:

$$\begin{aligned} \text{dis}(f, g) &:= \max\{J(f, g) \cup \{0\}\} \\ &= \max\{a \in \mathbb{N} \mid \gcd(f(x), g(x+a)) \neq 1\} \cup \{0\}\} \end{aligned}$$

and for a single polynomial $\text{dis}(f) := \text{dis}(f, f)$. Note that we make the definition $\max\{ \} := -\infty$.

See also:

[*diofant.polys.dispersion.dispersionset*](#) (page 679)

References

[R452] (page 1271), [R453] (page 1271), [R454] (page 1271), [R455] (page 1271)

Examples

```

>>> from diofant import poly
>>> from diofant.abc import x

```

Dispersion set and dispersion of a simple polynomial:

```
>>> fp = poly((x - 3)*(x + 3), x)
>>> sorted(dispersionset(fp))
[0, 6]
>>> dispersion(fp)
6
```

Note that the definition of the dispersion is not symmetric:

```
>>> fp = poly(x**4 - 3*x**2 + 1, x)
>>> gp = fp.shift(-3)
>>> sorted(dispersionset(fp, gp))
[2, 3, 4]
>>> dispersion(fp, gp)
4
>>> sorted(dispersionset(gp, fp))
[]
>>> dispersion(gp, fp)
-oo
```

The maximum of an empty set is defined to be $-\infty$ as seen in this example.

Computing the dispersion also works over field extensions:

```
>>> from diofant import sqrt
>>> fp = poly(x**2 + sqrt(5)*x - 1, x, domain='QQ<sqrt(5)>')
>>> gp = poly(x**2 + (2 + sqrt(5))*x + sqrt(5), x, domain='QQ<sqrt(5)>')
>>> sorted(dispersionset(fp, gp))
[2]
>>> sorted(dispersionset(gp, fp))
[1, 4]
```

We can even perform the computations for polynomials having symbolic coefficients:

```
>>> from diofant.abc import a
>>> fp = poly(4*x**4 + (4*a + 8)*x**3 + (a**2 + 6*a + 4)*x**2 + (a**2 + 2*a)*x, x)
>>> sorted(dispersionset(fp))
[0, 1]
```

3.13.4 AGCA - Algebraic Geometry and Commutative Algebra Module

Introduction

Algebraic geometry is a mixture of the ideas of two Mediterranean cultures. It is the superposition of the Arab science of the lightning calculation of the solutions of equations over the Greek art of position and shape. This tapestry was originally woven on European soil and is still being refined under the influence of international fashion. Algebraic geometry studies the delicate balance between the geometrically plausible and the algebraically possible. Whenever one side of this mathematical teeter-totter outweighs the other, one immediately loses interest and runs off in search of a more exciting amusement.

George R. Kempf 1944 - 2002

Algebraic Geometry refers to the study of geometric problems via algebraic methods (and sometimes vice versa). While this is a rather old topic, algebraic geometry as understood today is very much a 20th century development. Building on ideas of e.g. Riemann and

Dedekind, it was realized that there is an intimate connection between properties of the set of solutions of a system of polynomial equations (called an algebraic variety) and the behavior of the set of polynomial functions on that variety (called the coordinate ring).

As in many geometric disciplines, we can distinguish between local and global questions (and methods). Local investigations in algebraic geometry are essentially equivalent to the study of certain rings, their ideals and modules. This latter topic is also called commutative algebra. It is the basic local toolset of algebraic geometers, in much the same way that differential analysis is the local toolset of differential geometers.

A good conceptual introduction to commutative algebra is [Atiyah69] (page 1272). An introduction more geared towards computations, and the work most of the algorithms in this module are based on, is [Greuel2008] (page 1272).

This module aims to eventually allow expression and solution of both local and global geometric problems, both in the classical case over a field and in the more modern arithmetic cases. So far, however, there is no geometric functionality at all. Currently the module only provides tools for computational commutative algebra over fields.

All code examples assume:

```
>>> from diofant import *
>>> x, y, z = symbols('x,y,z')
>>> init_printing(pretty_print=True, use_unicode=True, wrap_line=False, no_
↳global=True)
```

Reference

In this section we document the usage of the AGCA module. For convenience of the reader, some definitions and examples/explanations are interspersed.

Base Rings

Almost all computations in commutative algebra are relative to a “base ring”. (For example, when asking questions about an ideal, the base ring is the ring the ideal is a subset of.) In principle all polys “domains” can be used as base rings. However, useful functionality is only implemented for polynomial rings over fields, and various localizations and quotients thereof.

As demonstrated in the examples below, the most convenient method to create objects you are interested in is to build them up from the ground field, and then use the various methods to create new objects from old. For example, in order to create the local ring of the nodal cubic $y^2 = x^3$ at the origin, over \mathbb{Q} , you do:

```
>>> lr = QQ.old_poly_ring(x, y, order="ilex") / [y**2 - x**3]
>>> lr
Q[x, y, order=ilex]
-----
 /   3   2\
 \- x  + y /
```

Note how the python list notation can be used as a short cut to express ideals. You can use the `convert` method to return ordinary diofant objects into objects understood by the AGCA module (although in many cases this will be done automatically – for example the list was automatically turned into an ideal, and in the process the symbols x and y were automatically converted into other representations). For example:

```
>>> X, Y = lr.convert(x), lr.convert(y) ; X
      / 3    2\
x + \- x  + y /

>>> x**3 == y**2
False

>>> X**3 == Y**2
True
```

When no localisation is needed, a more mathematical notation can be used. For example, let us create the coordinate ring of three-dimensional affine space \mathbb{A}^3 :

```
>>> ar = QQ.old_poly_ring(x, y, z); ar
Q[x, y, z]
```

For more details, refer to the following class documentation. Note that the base rings, being domains, are the main point of overlap between the AGCA module and the rest of the polys module. All domains are documented in detail in the polys reference, so we show here only an abridged version, with the methods most pertinent to the AGCA module.

class diofant.polys.domains.ring.Ring

Represents a ring domain.

free_module(rank)

Generate a free module of rank rank over self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

ideal(*gens)

Generate an ideal of self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

quotient_ring(e)

Form a quotient ring of self.

Here e can be an ideal or an iterable.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

diofant.polys.domains.polynomialring.**PolynomialRing**(*domain_or_ring*, *syms=None*, *order=None*)

A class for representing multivariate polynomial rings.

class diofant.polys.domains.quotientring.**QuotientRing**(ring, ideal)

Class representing (commutative) quotient rings.

You should not usually instantiate this by hand, instead use the constructor from the base ring in the construction.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**3 + 1)
>>> QQ.old_poly_ring(x).quotient_ring(I)
QQ[x]/<x**3 + 1>
```

Shorter versions are possible:

```
>>> QQ.old_poly_ring(x)/I
QQ[x]/<x**3 + 1>
```

```
>>> QQ.old_poly_ring(x)/[x**3 + 1]
QQ[x]/<x**3 + 1>
```

Attributes:

- ring - the base ring
- base_ideal - the ideal used to form the quotient

Modules, Ideals and their Elementary Properties

Let A be a ring. An A -module is a set M , together with two binary operations $+$: $M \times M \rightarrow M$ and \times : $R \times M \rightarrow M$ called addition and scalar multiplication. These are required to satisfy certain axioms, which can be found in e.g. [Atiyah69] (page 1272). In this way modules are a direct generalisation of both vector spaces (A being a field) and abelian groups ($A = \mathbb{Z}$). A *submodule* of the A -module M is a subset $N \subset M$, such that the binary operations restrict to N , and N becomes an A -module with these operations.

The ring A itself has a natural A -module structure where addition and multiplication in the module coincide with addition and multiplication in the ring. This A -module is also written as A . An A -submodule of A is called an *ideal* of A . Ideals come up very naturally in algebraic geometry. More general modules can be seen as a technically convenient “elbow room” beyond talking only about ideals.

If M, N are A -modules, then there is a natural (componentwise) A -module structure on $M \times N$. Similarly there are A -module structures on cartesian products of more components. (For the categorically inclined: the cartesian product of finitely many A -modules, with this A -module structure, is the finite biproduct in the category of all A -modules. With infinitely many components, it is the direct product (but the infinite direct sum has to be constructed differently).) As usual, repeated product of the A -module M is denoted $M, M^2, M^3 \dots$, or M^I for arbitrary index sets I .

An A -module M is called *free* if it is isomorphic to the A -module A^I for some (not necessarily finite) index set I (refer to the next section for a definition of isomorphism). The cardinality of I is called the *rank* of M ; one may prove this is well-defined. In general, the AGCA module only works with free modules of finite rank, and other closely related modules. The easiest way to create modules is to use member methods of the objects they are made up from. For example, let us create a free module of rank 4 over the coordinate ring of \mathbb{A}^2 we created above, together with a submodule:

```
>>> F = ar.free_module(4) ; F
      4
Q[x, y, z]

>>> S = F.submodule([1, x, x**2, x**3], [0, 1, 0, y]) ; S
/[ 2 3 \
 \1, x, x , x ], [0, 1, 0, y]/
```

Note how python lists can be used as a short-cut notation for module elements (vectors). As usual, the `convert` method can be used to convert diofant/python objects into the internal AGCA representation (see detailed reference below).

Here is the detailed documentation of the classes for modules, free modules, and submodules:

class `diofant.polys.agca.modules.Module`(*ring*)

Abstract base class for modules.

Do not instantiate - use ring explicit constructors instead:

```
>>> from diofant import QQ
>>> from diofant.abc import x
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

Attributes:

- `dtype` - type of elements
- `ring` - containing ring

Non-implemented methods:

- `submodule`
- `quotient_module`
- `is_zero`
- `is_submodule`
- `multiply_ideal`

The method `convert` likely needs to be changed in subclasses.

contains(*elem*)

Return True if *elem* is an element of this module.

convert(*elem*, *M=None*)

Convert *elem* into internal representation of this module.

If *M* is not None, it should be a module containing it.

identity_hom()

Return the identity homomorphism on *self*.

is_submodule(*other*)

Returns True if *other* is a submodule of *self*.

is_zero()

Returns True if *self* is a zero module.

multiply_ideal(*other*)

Multiply *self* by the ideal *other*.

quotient_module(*other*)
Generate a quotient module.

submodule(**gens*)
Generate a submodule.

subset(*other*)
Returns True if *other* is a subset of *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.subset([(1, x), (x, 2)])
True
>>> F.subset([(1/x, x), (x, 2)])
False
```

class `diofant.polys.agca.modules.FreeModule`(*ring*, *rank*)
Abstract base class for free modules.

Additional attributes:

- rank - rank of the free module

Non-implemented methods:

- submodule

basis()
Return a set of basis elements.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(3).basis()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

convert(*elem*, *M=None*)
Convert *elem* into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.convert([1, 0])
[1, 0]
```

dtype
alias of `FreeModuleElement` (page 753)

identity_hom()
Return the identity homomorphism on *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2).identity_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```


is_submodule(*other*)

Returns True if *other* is a submodule of *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> F.is_submodule(F)
True
>>> F.is_submodule(M)
True
>>> M.is_submodule(F)
False
```

is_zero()

Returns True if *self* is a zero module.

(If, as this implementation assumes, the coefficient ring is not the zero ring, then this is equivalent to the rank being zero.)

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(0).is_zero()
True
>>> QQ.old_poly_ring(x).free_module(1).is_zero()
False
```

multiply_ideal(*other*)

Multiply *self* by the ideal *other*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> I = QQ.old_poly_ring(x).ideal(x)
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.multiply_ideal(I)
<[x, 0], [0, x]>
```

quotient_module(*submodule*)

Return a quotient module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = QQ.old_poly_ring(x).free_module(2)
>>> M.quotient_module(M.submodule([1, x], [x, 2]))
QQ[x]**2/<[1, x], [x, 2]>
```

Or more concisely, using the overloaded division operator:

```
>>> QQ.old_poly_ring(x).free_module(2) / [[1, x], [x, 2]]
QQ[x]**2/<[1, x], [x, 2]>
```

class diofant.polys.agca.modules.**SubModule**(*gens*, *container*)

Base class for submodules.

Attributes:

- *container* - containing module
- *gens* - generators (subset of containing module)
- *rank* - rank of containing module

Non-implemented methods:

- `_contains`
- `_syzygies`
- `_in_terms_of_generators`
- `_intersect`
- `_module_quotient`

Methods that likely need change in subclasses:

- `reduce_element`

convert(*elem*, *M=None*)

Convert *elem* into the internal representation.

Mostly called implicitly.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, x])
>>> M.convert([2, 2*x])
[2, 2*x]
```

identity_hom()

Return the identity homomorphism on *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).identity_hom()
Matrix([
[1, 0], : <[x, x]> -> <[x, x]>
[0, 1]])
```

in_terms_of_generators(*e*)

Express element *e* of *self* in terms of the generators.

```
>>> from diofant.abc import x
>>> from diofant import QQ
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([1, 0], [1, 1])
>>> M.in_terms_of_generators([x, x**2])
[-x**2 + x, x**2]
```

inclusion_hom()

Return a homomorphism representing the inclusion map of *self*.

That is, the natural map from *self* to *self*.container.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([x, x]).inclusion_hom()
Matrix([
[1, 0], : <[x, x]> -> QQ[x]**2
[0, 1]])
```

intersect(*other*, ***options*)

Returns the intersection of *self* with submodule *other*.

```
>>> from diofant.abc import x, y
>>> from diofant import QQ
```

```
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> F.submodule([x, x]).intersect(F.submodule([y, y]))
<[x*y, x*y]>
```

Some implementation allow further options to be passed. Currently, to only one implemented is `relations=True`, in which case the function will return a triple `(res, rela, relb)`, where `res` is the intersection module, and `rela` and `relb` are lists of coefficient vectors, expressing the generators of `res` in terms of the generators of `self` (`rela`) and other (`relb`).

```
>>> F.submodule([x, x]).intersect(F.submodule([y, y]), relations=True)
(<[x*y, x*y]>, [(y,)], [(x,)])
```

The above result says: the intersection module is generated by the single element $(-xy, -xy) = -y(x, x) = -x(y, y)$, where (x, x) and (y, y) respectively are the unique generators of the two modules being intersected.

`is_full_module()`

Return True if `self` is the entire free module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

`is_submodule(other)`

Returns True if `other` is a submodule of `self`.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> M = F.submodule([2, x])
>>> N = M.submodule([2*x, x**2])
>>> M.is_submodule(M)
True
>>> M.is_submodule(N)
True
>>> N.is_submodule(M)
False
```

`is_zero()`

Return True if `self` is a zero module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_zero()
False
>>> F.submodule([0, 0]).is_zero()
True
```

module_quotient(*other*, ****options**)

Returns the module quotient of *self* by submodule *other*.

That is, if *self* is the module M and *other* is N , then return the ideal $\{f \in R \mid fN \subset M\}$.

```
>>> from diofant import QQ
>>> from diofant.abc import x, y
```

```
>>> F = QQ.old_poly_ring(x, y).free_module(2)
>>> S = F.submodule([x*y, x*y])
>>> T = F.submodule([x, x])
>>> S.module_quotient(T)
<y>
```

Some implementations allow further options to be passed. Currently, the only one implemented is `relations=True`, which may only be passed if *other* is principal. In this case the function will return a pair (*res*, *rel*) where *res* is the ideal, and *rel* is a list of coefficient vectors, expressing the generators of the ideal, multiplied by the generator of *other* in terms of generators of *self*.

```
>>> S.module_quotient(T, relations=True)
(<y>, [[1]])
```

This means that the quotient ideal is generated by the single element y , and that $y(x, x) = 1(xy, xy)$, (x, x) and (xy, xy) being the generators of T and S , respectively.

multiply_ideal(*I*)

Multiply *self* by the ideal I .

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> I = QQ.old_poly_ring(x).ideal(x**2)
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([1, 1])
>>> I*M
<[x**2, x**2]>
```

quotient_module(*other*, ****opts**)

Return a quotient module.

This is the same as taking a submodule of a quotient of the containing module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S1 = F.submodule([x, 1])
>>> S2 = F.submodule([x**2, x])
>>> S1.quotient_module(S2)
<[x, 1] + <[x**2, x]>>
```

Or more coincisely, using the overloaded division operator:

```
>>> F.submodule([x, 1]) / [(x**2, x)]
<[x, 1] + <[x**2, x]>>
```

reduce_element(*x*)

Reduce the element x of our ring modulo the ideal *self*.

Here “reduce” has no specific meaning, it could return a unique normal form, simplify the expression a bit, or just do nothing.

submodule(*gens)

Generate a submodule.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = QQ.old_poly_ring(x).free_module(2).submodule([x, 1])
>>> M.submodule([x**2, x])
<[x**2, x]>
```

syzygy_module(opts)**

Compute the syzygy module of the generators of self.

Suppose M is generated by f_1, \dots, f_n over the ring R . Consider the homomorphism $\phi : R^n \rightarrow M$, given by sending $(r_1, \dots, r_n) \rightarrow r_1 f_1 + \dots + r_n f_n$. The syzygy module is defined to be the kernel of ϕ .

The syzygy module is zero iff the generators generate freely a free submodule:

```
>>> from diofant.abc import x, y
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2).submodule([1, 0], [1, 1]).syzygy_
    module().is_zero()
True
```

A slightly more interesting example:

```
>>> M = QQ.old_poly_ring(x, y).free_module(2).submodule([x, 2*x], [y, 2*y])
>>> S = QQ.old_poly_ring(x, y).free_module(2).submodule([y, -x])
>>> M.syzygy_module() == S
True
```

union(other)

Returns the module generated by the union of self and other.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(1)
>>> M = F.submodule([x**2 + x]) # <x(x+1)>
>>> N = F.submodule([x**2 - 1]) # <(x-1)(x+1)>
>>> M.union(N) == F.submodule([x+1])
True
```

class diofant.polys.agca.modules.FreeModuleElement(module, data)

Element of a free module. Data stored as a tuple.

Ideals are created very similarly to modules. For example, let's verify that the nodal cubic is indeed singular at the origin:

```
>>> I = lr.ideal(x, y)
>>> I == lr.ideal(x)
False

>>> I == lr.ideal(y)
False
```

We are using here the fact that a curve is non-singular at a point if and only if the maximal ideal of the local ring is principal, and that in this case at least one of x and y must be generators.

This is the detailed documentation of the class ideal. Please note that most of the methods regarding properties of ideals (primality etc.) are not yet implemented.

class diofant.polys.agca.ideals.**Ideal**(ring)

Abstract base class for ideals.

Do not instantiate - use explicit constructors in the ring class instead:

```
>>> from diofant import QQ
>>> from diofant.abc import x
>>> QQ.old_poly_ring(x).ideal(x+1)
<x + 1>
```

Attributes

- ring - the ring this ideal belongs to

Non-implemented methods:

- _contains_elem
- _contains_ideal
- _quotient
- _intersect
- _union
- _product
- is_whole_ring
- is_zero
- is_prime, is_maximal, is_primary, is_radical
- is_principal
- height, depth
- radical

Methods that likely should be overridden in subclasses:

- reduce_element

contains(elem)

Return True if elem is an element of this ideal.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).ideal(x+1, x-1).contains(3)
True
>>> QQ.old_poly_ring(x).ideal(x**2, x**3).contains(x)
False
```

depth()

Compute the depth of self.

height()

Compute the height of self.

intersect(J)

Compute the intersection of self with ideal J.

```
>>> from diofant.abc import x, y
>>> from diofant import QQ
>>> R = QQ.old_poly_ring(x, y)
```

```
>>> R.ideal(x).intersect(R.ideal(y))
<x*y>
```

is_maximal()

Return True if `self` is a maximal ideal.

is_primary()

Return True if `self` is a primary ideal.

is_prime()

Return True if `self` is a prime ideal.

is_principal()

Return True if `self` is a principal ideal.

is_radical()

Return True if `self` is a radical ideal.

is_whole_ring()

Return True if `self` is the whole ring.

is_zero()

Return True if `self` is the zero ideal.

product(J)

Compute the ideal product of `self` and `J`.

That is, compute the ideal generated by products xy , for x an element of `self` and $y \in J$.

```
>>> from diofant.abc import x, y
>>> from diofant import QQ
>>> QQ.old_poly_ring(x, y).ideal(x).product(QQ.old_poly_ring(x, y).ideal(y))
<x*y>
```

quotient(J, **opts)

Compute the ideal quotient of `self` by `J`.

That is, if `self` is the ideal I , compute the set $I : J = \{x \in R \mid xJ \subset I\}$.

```
>>> from diofant.abc import x, y
>>> from diofant import QQ
>>> R = QQ.old_poly_ring(x, y)
>>> R.ideal(x*y).quotient(R.ideal(x))
<y>
```

radical()

Compute the radical of `self`.

reduce_element(x)

Reduce the element `x` of our ring modulo the ideal `self`.

Here “reduce” has no specific meaning: it could return a unique normal form, simplify the expression a bit, or just do nothing.

saturate(J)

Compute the ideal saturation of `self` by `J`.

That is, if `self` is the ideal I , compute the set $I : J^\infty = \{x \in R \mid xJ^n \subset I \text{ for some } n\}$.

subset(other)

Returns True if `other` is a subset of `self`.

Here `other` may be an ideal.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> I = QQ.old_poly_ring(x).ideal(x+1)
>>> I.subset([x**2 - 1, x**2 + 2*x + 1])
True
>>> I.subset([x**2 + 1, x + 1])
False
>>> I.subset(QQ.old_poly_ring(x).ideal(x**2 - 1))
True
```

union(J)

Compute the ideal generated by the union of self and J.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).ideal(x**2 - 1).union(QQ.old_poly_ring(x).
↳ ideal((x+1)**2)) == QQ.old_poly_ring(x).ideal(x+1)
True
```

If M is an A -module and N is an A -submodule, we can define two elements x and y of M to be equivalent if $x - y \in N$. The set of equivalence classes is written M/N , and has a natural A -module structure. This is called the quotient module of M by N . If K is a submodule of M containing N , then K/N is in a natural way a submodule of M/N . Such a module is called a subquotient. Here is the documentation of quotient and subquotient modules:

class `diofant.polys.agca.modules.QuotientModule`(*ring, base, submodule*)

Class for quotient modules.

Do not instantiate this directly. For subquotients, see the `SubQuotientModule` class.

Attributes:

- `base` - the base module we are a quotient of
- `killed_module` - the submodule used to form the quotient
- `rank` of the base

convert(*elem, M=None*)

Convert `elem` into the internal representation.

This method is called implicitly whenever computations involve elements not in the internal representation.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> F.convert([1, 0])
[1, 0] + <[1, 2], [1, x]>
```

dtype

alias of [QuotientModuleElement](#) (page 758)

identity_hom()

Return the identity homomorphism on self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.identity_hom()
Matrix([
```



```
[1, 0], : QQ[x]**2/<[1, 2], [1, x]> -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

is_submodule(*other*)

Return True if *other* is a submodule of *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> S = Q.submodule([1, 0])
>>> Q.is_submodule(S)
True
>>> S.is_submodule(Q)
False
```

is_zero()

Return True if *self* is a zero module.

This happens if and only if the base module is the same as the submodule being killed.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> (F/[(1, 0)]).is_zero()
False
>>> (F/[(1, 0), (0, 1)]).is_zero()
True
```

quotient_hom()

Return the quotient homomorphism to *self*.

That is, return a homomorphism representing the natural map from *self*.base to *self*.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = QQ.old_poly_ring(x).free_module(2) / [(1, 2), (1, x)]
>>> M.quotient_hom()
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2/<[1, 2], [1, x]>
[0, 1]])
```

submodule(gens*, ***opts*)**

Generate a submodule.

This is the same as taking a quotient of a submodule of the base module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> Q = QQ.old_poly_ring(x).free_module(2) / [(x, x)]
>>> Q.submodule([x, 0])
<[x, 0] + <[x, x]>>
```

class diofant.polys.agca.modules.SubQuotientModule(*gens*, *container*, *opts*)**

Submodule of a quotient module.

Equivalently, quotient module of a submodule.

Do not instantiate this, instead use the submodule or quotient_module constructing methods:

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> S = F.submodule([1, 0], [1, x])
>>> Q = F/[(1, 0)]
>>> S/[(1, 0)] == Q.submodule([5, x])
True
```

Attributes:

- `base` - base module we are quotient of
- `killed_module` - submodule used to form the quotient

is_full_module()

Return True if `self` is the entire free module.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> F.submodule([x, 1]).is_full_module()
False
>>> F.submodule([1, 1], [1, 2]).is_full_module()
True
```

quotient_hom()

Return the quotient homomorphism to self.

That is, return the natural map from `self.base` to self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> M = (QQ.old_poly_ring(x).free_module(2) / [(1, x)].submodule([1, 0]))
>>> M.quotient_hom()
Matrix([
[1, 0], : <[1, 0], [1, x]> -> <[1, 0] + <[1, x]>, [1, x] + <[1, x]>>
[0, 1]])
```

class `diofant.polys.agca.modules.QuotientModuleElement`(*module, data*)

Element of a quotient module.

eq(*d1, d2*)

Equality comparison.

Module Homomorphisms and Syzygies

Let M and N be A -modules. A mapping $f : M \rightarrow N$ satisfying various obvious properties (see [Atiyah69] (page 1272)) is called an A -module homomorphism. In this case M is called the *domain* and N the *codomain*. The set $\{x \in M \mid f(x) = 0\}$ is called the *kernel* $\ker(f)$, whereas the set $\{f(x) \mid x \in M\}$ is called the *image* $\text{im}(f)$. The kernel is a submodule of M , the image is a submodule of N . The homomorphism f is injective if and only if $\ker(f) = 0$ and surjective if and only if $\text{im}(f) = N$. A bijective homomorphism is called an *isomorphism*. Equivalently, $\ker(f) = 0$ and $\text{im}(f) = N$. (A related notion, which currently has no special name in the AGCA module, is that of the *cokernel*, $\text{coker}(f) = N/\text{im}(f)$.)

Suppose now M is an A -module. M is called *finitely generated* if there exists a surjective homomorphism $A^n \rightarrow M$ for some n . If such a morphism f is chosen, the images of the standard basis of A^n are called the *generators* of M . The module $\ker(f)$ is called *syzygy module* with respect to the generators. A module is called *finitely presented* if it is finitely generated with a finitely generated syzygy module. The class of finitely presented modules is essentially the largest class we can hope to be able to meaningfully compute in.

It is an important theorem that, for all the rings we are considering, all submodules of finitely generated modules are finitely generated, and hence finitely generated and finitely presented modules are the same.

The notion of syzygies, while it may first seem rather abstract, is actually very computational. This is because there exist (fairly easy) algorithms for computing them, and more general questions (kernels, intersections, ...) are often reduced to syzygy computation.

Let us say a few words about the definition of homomorphisms in the AGCA module. Suppose first that $f : M \rightarrow N$ is an arbitrary morphism of A -modules. Then if K is a submodule of M , f naturally defines a new homomorphism $g : K \rightarrow N$ (via $g(x) = f(x)$), called the *restriction* of f to K . If now K contained in the kernel of f , then moreover f defines in a natural homomorphism $g : M/K \rightarrow N$ (same formula as above!), and we say that f *descends* to M/K . Similarly, if L is a submodule of N , there is a natural homomorphism $g : M \rightarrow N/L$, we say that g *factors* through f . Finally, if now L contains the image of f , then there is a natural homomorphism $g : M \rightarrow L$ (defined, again, by the same formula), and we say g is obtained from f by restriction of codomain. Observe also that each of these four operations is reversible, in the sense that given g , one can always (non-uniquely) find f such that g is obtained from f in the above way.

Note that all modules implemented in AGCA are obtained from free modules by taking a succession of submodules and quotients. Hence, in order to explain how to define a homomorphism between arbitrary modules, in light of the above, we need only explain how to define homomorphisms of free modules. But, essentially by the definition of free module, a homomorphism from a free module A^n to any module M is precisely the same as giving n elements of M (the images of the standard basis), and giving an element of a free module A^m is precisely the same as giving m elements of A . Hence a homomorphism of free modules $A^n \rightarrow A^m$ can be specified via a matrix, entirely analogously to the case of vector spaces.

The functions `restrict_domain` etc. of the class `Homomorphism` can be used to carry out the operations described above, and homomorphisms of free modules can in principle be instantiated by hand. Since these operations are so common, there is a convenience function `homomorphism` to define a homomorphism between arbitrary modules via the method outlined above. It is essentially the only way homomorphisms need ever be created by the user.

`diofant.polys.agca.homomorphisms.homomorphism(domain, codomain, matrix)`

Create a homomorphism object.

This function tries to build a homomorphism from `domain` to `codomain` via the matrix `matrix`.

Examples

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> R = QQ.old_poly_ring(x)
>>> T = R.free_module(2)
```

If `domain` is a free module generated by e_1, \dots, e_n , then `matrix` should be an n -element iterable (b_1, \dots, b_n) where the b_i are elements of `codomain`. The constructed homomor-

phism is the unique homomorphism sending e_i to b_i .

```
>>> F = R.free_module(2)
>>> h = homomorphism(F, T, [[1, x], [x**2, 0]])
>>> h
Matrix([
[1, x**2], : QQ[x]**2 -> QQ[x]**2
[x,      0]])
>>> h([1, 0])
[1, x]
>>> h([0, 1])
[x**2, 0]
>>> h([1, 1])
[x**2 + 1, x]
```

If domain is a submodule of a free module, then matrix determines a homomorphism from the containing free module to codomain, and the homomorphism returned is obtained by restriction to domain.

```
>>> S = F.submodule([1, 0], [0, x])
>>> homomorphism(S, T, [[1, x], [x**2, 0]])
Matrix([
[1, x**2], : <[1, 0], [0, x]> -> QQ[x]**2
[x,      0]])
```

If domain is a (sub)quotient N/K , then matrix determines a homomorphism from N to codomain. If the kernel contains K , this homomorphism descends to domain and is returned; otherwise an exception is raised.

```
>>> homomorphism(S/[(1, 0)], T, [0, [x**2, 0]])
Matrix([
[0, x**2], : <[1, 0] + <[1, 0]>, [0, x] + <[1, 0]>, [1, 0] + <[1, 0]>> -> QQ[x]**2
[0,      0]])
>>> homomorphism(S/[(0, x)], T, [0, [x**2, 0]])
Traceback (most recent call last):
...
ValueError: kernel <[1, 0], [0, 0]> must contain sm, got <[0,x]>
```

Finally, here is the detailed reference of the actual homomorphism class:

class diofant.polys.agca.homomorphisms.**ModuleHomomorphism**(domain, codomain)
Abstract base class for module homomorphisms. Do not instantiate.

Instead, use the homomorphism function:

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [0, 1]])
Matrix([
[1, 0], : QQ[x]**2 -> QQ[x]**2
[0, 1]])
```

Attributes:

- ring - the ring over which we are considering modules
- domain - the domain module

- `codomain` - the codomain module
- `_ker` - cached kernel
- `_img` - cached image

Non-implemented methods:

- `_kernel`
- `_image`
- `_restrict_domain`
- `_restrict_codomain`
- `_quotient_domain`
- `_quotient_codomain`
- `_apply`
- `_mul_scalar`
- `_compose`
- `_add`

`image()`

Compute the image of `self`.

That is, if `self` is the homomorphism $\phi : M \rightarrow N$, then compute $im(\phi) = \{\phi(x) | x \in M\}$. This is a submodule of N .

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).image() == F.submodule([1, 0])
True
```

`is_injective()`

Return True if `self` is injective.

That is, check if the elements of the domain are mapped to the same codomain element.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_injective()
False
>>> h.quotient_domain(h.kernel()).is_injective()
True
```

`is_isomorphism()`

Return True if `self` is an isomorphism.

That is, check if every element of the codomain has precisely one preimage. Equivalently, `self` is both injective and surjective.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h = h.restrict_codomain(h.image())
>>> h.is_isomorphism()
False
>>> h.quotient_domain(h.kernel()).is_isomorphism()
True
```

is_surjective()

Return True if `self` is surjective.

That is, check if every element of the codomain has at least one preimage.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_surjective()
False
>>> h.restrict_codomain(h.image()).is_surjective()
True
```

is_zero()

Return True if `self` is a zero morphism.

That is, check if every element of the domain is mapped to zero under `self`.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h.is_zero()
False
>>> h.restrict_domain(F.submodule()).is_zero()
True
>>> h.quotient_codomain(h.image()).is_zero()
True
```

kernel()

Compute the kernel of `self`.

That is, if `self` is the homomorphism $\phi : M \rightarrow N$, then compute $\ker(\phi) = \{x \in M \mid \phi(x) = 0\}$. This is a submodule of M .

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> homomorphism(F, F, [[1, 0], [x, 0]]).kernel()
<[x, -1]>
```

quotient_codomain(sm)

Return `self` with codomain replaced by `codomain/sm`.

Here `sm` must be a submodule of `self.codomain`.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_codomain(F.submodule([1, 1]))
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

This is the same as composing with the quotient map on the left:

```
>>> (F/[1, 1]).quotient_hom() * h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2/<[1, 1]>
[0, 0]])
```

quotient_domain(sm)

Return self with domain replaced by domain/sm.

Here sm must be a submodule of self.kernel().

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.quotient_domain(F.submodule([-x, 1]))
Matrix([
[1, x], : QQ[x]**2/<[-x, 1]> -> QQ[x]**2
[0, 0]])
```

restrict_codomain(sm)

Return self, with codomain restricted to to sm.

Here sm has to be a submodule of self.codomain containing the image.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_codomain(F.submodule([1, 0]))
Matrix([
[1, x], : QQ[x]**2 -> <[1, 0]>
[0, 0]])
```

restrict_domain(sm)

Return self, with the domain restricted to sm.

Here sm has to be a submodule of self.domain.

```
>>> from diofant import QQ
>>> from diofant.abc import x
```

```
>>> F = QQ.old_poly_ring(x).free_module(2)
>>> h = homomorphism(F, F, [[1, 0], [x, 0]])
>>> h
Matrix([
[1, x], : QQ[x]**2 -> QQ[x]**2
[0, 0]])
>>> h.restrict_domain(F.submodule([1, 0]))
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

This is the same as just composing on the right with the submodule inclusion:

```
>>> h * F.submodule([1, 0]).inclusion_hom()
Matrix([
[1, x], : <[1, 0]> -> QQ[x]**2
[0, 0]])
```

3.13.5 Internals of the Polynomial Manipulation Module

The implementation of the polynomials module is structured internally in “levels”. There are four levels, called L0, L1, L2 and L3. The levels three and four contain the user-facing functionality and were described in the previous section. This section focuses on levels zero and one.

Level zero provides core polynomial manipulation functionality with C-like, low-level interfaces. Level one wraps this low-level functionality into object oriented structures. These are *not* the classes seen by the user, but rather classes used internally throughout the polys module.

There is one additional complication in the implementation. This comes from the fact that all polynomial manipulations are relative to a *ground domain*. For example, when factoring a polynomial like $x^{10} - 1$, one has to decide what ring the coefficients are supposed to belong to, or less trivially, what coefficients are allowed to appear in the factorization. This choice of coefficients is called a ground domain. Typical choices include the integers \mathbb{Z} , the rational numbers \mathbb{Q} or various related rings and fields. But it is perfectly legitimate (although in this case uninteresting) to factorize over polynomial rings such as $k[Y]$, where k is some fixed field.

Thus the polynomial manipulation algorithms (both complicated ones like factoring, and simpler ones like addition or multiplication) have to rely on other code to manipulate the coefficients. In the polynomial manipulation module, such code is encapsulated in so-called “domains”. A domain is basically a factory object: it takes various representations of data, and converts them into objects with unified interface. Every object created by a domain has to implement the arithmetic operations $+$, $-$ and \times . Other operations are accessed through the domain, e.g. as in `ZZ.quo(ZZ(4), ZZ(2))`.

Note that there is some amount of *circularity*: the polynomial ring domains use the level one classes, the level one classes use the level zero functions, and level zero functions use domains. It is possible, in principle, but not in the current implementation, to work in rings

like $k[X][Y]$. This would create even more layers. For this reason, working in the isomorphic ring $k[X, Y]$ is preferred.

Domains

Here we document the various implemented ground domains. There are three types: abstract domains, concrete domains, and “implementation domains”. Abstract domains cannot be (usefully) instantiated at all, and just collect together functionality shared by many other domains. Concrete domains are those meant to be instantiated and used in the polynomial manipulation algorithms. In some cases, there are various possible ways to implement the data type the domain provides. For example, depending on what libraries are available on the system, the integers are implemented either using the python built-in integers, or using gmpy. Note that various aliases are created automatically depending on the libraries available. As such e.g. ZZ always refers to the most efficient implementation of the integer ring available.

Abstract Domains

```
class diofant.polys.domains.domain.Domain
    Represents an abstract domain.

    abs(a)
        Absolute value of a, implies __abs__.

    add(a, b)
        Sum of a and b, implies __add__.

    algebraic_field(*extension)
        Returns an algebraic field, i.e.  $K(\alpha, \dots)$ .

    almosteq(a, b, tolerance=None)
        Check if a and b are almost equal.

    characteristic()
        Return the characteristic of this domain.

    cofactors(a, b)
        Returns GCD and cofactors of a and b.

    convert(element, base=None)
        Convert element to self.dtype.

    convert_from(element, base)
        Convert element to self.dtype given the base domain.

    denom(a)
        Returns denominator of a.

    div(a, b)
        Division of a and b, implies something.

    evalf(a, prec=None, **options)
        Returns numerical approximation of a.

    exquo(a, b)
        Exact quotient of a and b, implies something.

    frac_field(*symbols, **kwargs)
        Returns a fraction field, i.e.  $K(X)$ .
```

from_AlgebraicField(*a*, *K0*)
Convert an algebraic number to dtype.

from_ComplexField(*a*, *K0*)
Convert a complex element to dtype.

from_ExpressionDomain(*a*, *K0*)
Convert a EX object to dtype.

from_FF_gmpy(*a*, *K0*)
Convert ModularInteger(mpz) to dtype.

from_FF_python(*a*, *K0*)
Convert ModularInteger(int) to dtype.

from_FractionField(*a*, *K0*)
Convert a rational function to dtype.

from_GlobalPolynomialRing(*a*, *K0*)
Convert a polynomial to dtype.

from_PolynomialRing(*a*, *K0*)
Convert a polynomial to dtype.

from_QQ_gmpy(*a*, *K0*)
Convert a GMPY mpq object to dtype.

from_QQ_python(*a*, *K0*)
Convert a Python Fraction object to dtype.

from_RealField(*a*, *K0*)
Convert a real element object to dtype.

from_ZZ_gmpy(*a*, *K0*)
Convert a GMPY mpz object to dtype.

from_ZZ_python(*a*, *K0*)
Convert a Python int object to dtype.

from_diofant(*a*)
Convert a Diofant object to dtype.

gcd(*a*, *b*)
Returns GCD of *a* and *b*.

gcdex(*a*, *b*)
Extended GCD of *a* and *b*.

get_exact()
Returns an exact domain associated with self.

get_field()
Returns a field associated with self.

get_ring()
Returns a ring associated with self.

half_gcdex(*a*, *b*)
Half extended GCD of *a* and *b*.

inject(**symbols*)
Inject generators into this domain.

invert(*a*, *b*)
Returns inversion of *a* mod *b*, implies something.

is_negative(a)
Returns True if a is negative.

is_nonnegative(a)
Returns True if a is non-negative.

is_nonpositive(a)
Returns True if a is non-positive.

is_one(a)
Returns True if a is one.

is_positive(a)
Returns True if a is positive.

is_zero(a)
Returns True if a is zero.

lcm(a, b)
Returns LCM of a and b.

log(a, b)
Returns b-base logarithm of a.

map(seq)
Rersively apply self to all elements of seq.

mul(a, b)
Product of a and b, implies `__mul__`.

n(a, prec=None, **options)
Returns numerical approximation of a.

neg(a)
Returns a negated, implies `__neg__`.

numer(a)
Returns numerator of a.

of_type(element)
Check if a is of type dtype.

old_frac_field(*symbols, **kwargs)
Returns a fraction field, i.e. $K(X)$.

old_poly_ring(*symbols, **kwargs)
Returns a polynomial ring, i.e. $K[X]$.

poly_ring(*symbols, **kwargs)
Returns a polynomial ring, i.e. $K[X]$.

pos(a)
Returns a positive, implies `__pos__`.

pow(a, b)
Raise a to power b, implies `__pow__`.

quo(a, b)
Quotient of a and b, implies something.

rem(a, b)
Remainder of a and b, implies `__mod__`.

revert(a)
Returns $a^{**(-1)}$ if possible.

sqrt(*a*)

Returns square root of *a*.

sub(*a*, *b*)

Difference of *a* and *b*, implies `__sub__`.

to_diofant(*a*)

Convert *a* to a Diofant object.

unify(*K1*, *symbols=None*)

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.polys.domains.field.Field`

Represents a field domain.

div(*a*, *b*)

Division of *a* and *b*, implies `__div__`.

exquo(*a*, *b*)

Exact quotient of *a* and *b*, implies `__div__`.

gcd(*a*, *b*)

Returns GCD of *a* and *b*.

This definition of GCD over fields allows to clear denominators in `primitive()`.

```
>>> from diofant.polys.domains import QQ
>>> from diofant import Rational, gcd, primitive
>>> from diofant.abc import x
```

```
>>> 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)
```

get_field()

Returns a field associated with `self`.

get_ring()

Returns a ring associated with `self`.

lcm(*a*, *b*)

Returns LCM of *a* and *b*.

```
>>> from diofant.polys.domains import QQ
>>> from diofant import Rational, lcm
```

```
>>> QQ.lcm(QQ(2, 3), QQ(4, 9))
4/3
>>> lcm(Rational(2, 3), Rational(4, 9))
4/3
```

quo(*a*, *b*)
Quotient of *a* and *b*, implies `__div__`.

rem(*a*, *b*)
Remainder of *a* and *b*, implies nothing.

revert(*a*)
Returns $a^{**}(-1)$ if possible.

class `diofant.polys.domains.ring.Ring`
Represents a ring domain.

denom(*a*)
Returns denominator of *a*.

div(*a*, *b*)
Division of *a* and *b*, implies `__divmod__`.

exquo(*a*, *b*)
Exact quotient of *a* and *b*, implies `__floordiv__`.

free_module(*rank*)
Generate a free module of rank *rank* over self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).free_module(2)
QQ[x]**2
```

get_ring()
Returns a ring associated with self.

ideal(**gens*)
Generate an ideal of self.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).ideal(x**2)
<x**2>
```

invert(*a*, *b*)
Returns inversion of *a* mod *b*.

numer(*a*)
Returns numerator of *a*.

quo(*a*, *b*)
Quotient of *a* and *b*, implies `__floordiv__`.

quotient_ring(*e*)
Form a quotient ring of self.
Here *e* can be an ideal or an iterable.

```
>>> from diofant.abc import x
>>> from diofant import QQ
>>> QQ.old_poly_ring(x).quotient_ring(QQ.old_poly_ring(x).ideal(x**2))
QQ[x]/<x**2>
>>> QQ.old_poly_ring(x).quotient_ring([x**2])
QQ[x]/<x**2>
```

The division operator has been overloaded for this:

```
>>> QQ.old_poly_ring(x)/[x**2]
QQ[x]/<x**2>
```

rem(*a*, *b*)

Remainder of *a* and *b*, implies `__mod__`.

revert(*a*)

Returns a^{*-1} if possible.

class `diofant.polys.domains.simplesdomain.SimpleDomain`

Base class for simple domains, e.g. ZZ, QQ.

inject(**gens*)

Inject generators into this domain.

class `diofant.polys.domains.compositedomain.CompositeDomain`

Base class for composite domains, e.g. ZZ[x], ZZ(X).

inject(**symbols*)

Inject generators into this domain.

Concrete Domains

class `diofant.polys.domains.FiniteField(mod, dom, symmetric=True)`

General class for finite fields.

characteristic()

Return the characteristic of this domain.

from_FF_gmpy(*a*, *K0=None*)

Convert ModularInteger(mpz) to dtype.

from_FF_python(*a*, *K0=None*)

Convert ModularInteger(int) to dtype.

from_QQ_gmpy(*a*, *K0=None*)

Convert GMPY's mpq to dtype.

from_QQ_python(*a*, *K0=None*)

Convert Python's Fraction to dtype.

from_RealField(*a*, *K0*)

Convert mpmath's mpf to dtype.

from_ZZ_gmpy(*a*, *K0=None*)

Convert GMPY's mpz to dtype.

from_ZZ_python(*a*, *K0=None*)

Convert Python's int to dtype.

from_diofant(*a*)

Convert Diofant's Integer to Diofant's Integer.

get_field()
Returns a field associated with `self`.

to_diofant(*a*)
Convert *a* to a Diofant object.

class diofant.polys.domains.IntegerRing
General class for integer rings.

algebraic_field(extension*)**
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

from_AlgebraicField(*a*, *K0*)
Convert a ANP object to dtype.

get_field()
Returns a field associated with `self`.

log(*a*, *b*)
Returns *b*-base logarithm of *a*.

class diofant.polys.domains.PolynomialRing(*domain_or_ring*, *symbols=None*, *order=None*)
A class for representing multivariate polynomial rings.

factorial(*a*)
Returns factorial of *a*.

from_AlgebraicField(*a*, *K0*)
Convert an algebraic number to dtype.

from_FractionField(*a*, *K0*)
Convert a rational function to dtype.

from_PolynomialRing(*a*, *K0*)
Convert a polynomial to dtype.

from_QQ_gmpy(*a*, *K0*)
Convert a GMPY *mpq* object to dtype.

from_QQ_python(*a*, *K0*)
Convert a Python *Fraction* object to dtype.

from_RealField(*a*, *K0*)
Convert a mpmath *mpf* object to dtype.

from_ZZ_gmpy(*a*, *K0*)
Convert a GMPY *mpz* object to dtype.

from_ZZ_python(*a*, *K0*)
Convert a Python *int* object to dtype.

from_diofant(*a*)
Convert Diofant's expression to dtype.

gcd(*a*, *b*)
Returns GCD of *a* and *b*.

gcdex(*a*, *b*)
Extended GCD of *a* and *b*.

get_field()
Returns a field associated with `self`.

is_negative(*a*)
Returns True if $LC(a)$ is negative.

is_nonnegative(*a*)
Returns True if $LC(a)$ is non-negative.

is_nonpositive(*a*)
Returns True if $LC(a)$ is non-positive.

is_positive(*a*)
Returns True if $LC(a)$ is positive.

lcm(*a*, *b*)
Returns LCM of *a* and *b*.

to_diofant(*a*)
Convert *a* to a Diofant object.

class diofant.polys.domains.**RationalField**
General class for rational fields.

algebraic_field(**extension*)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

from_AlgebraicField(*a*, *K0*)
Convert a ANP object to dtype.

class diofant.polys.domains.**AlgebraicField**(*dom*, **ext*)
A class for representing algebraic number fields.

algebraic_field(**extension*)
Returns an algebraic field, i.e. $\mathbb{Q}(\alpha, \dots)$.

denom(*a*)
Returns denominator of *a*.

dtype
alias of [ANP](#) (page 781)

from_QQ_gmpy(*a*, *K0*)
Convert a GMPY mpq object to dtype.

from_QQ_python(*a*, *K0*)
Convert a Python Fraction object to dtype.

from_RealField(*a*, *K0*)
Convert a mpmath mpf object to dtype.

from_ZZ_gmpy(*a*, *K0*)
Convert a GMPY mpz object to dtype.

from_ZZ_python(*a*, *K0*)
Convert a Python int object to dtype.

from_diofant(*a*)
Convert Diofant's expression to dtype.

get_ring()
Returns a ring associated with self.

is_negative(*a*)
Returns True if *a* is negative.

is_nonnegative(*a*)
Returns True if *a* is non-negative.

is_nonpositive(*a*)
Returns True if *a* is non-positive.

is_positive(*a*)
Returns True if *a* is positive.

numer(*a*)
Returns numerator of *a*.

to_diofant(*a*)
Convert *a* to a Diofant object.

class diofant.polys.domains.FractionField(*domain_or_field*, *symbols=None*, *order=None*)
A class for representing multivariate rational function fields.

denom(*a*)
Returns denominator of *a*.

factorial(*a*)
Returns factorial of *a*.

from_AlgebraicField(*a*, *K0*)
Convert an algebraic number to *dtype*.

from_FractionField(*a*, *K0*)
Convert a rational function to *dtype*.

from_PolynomialRing(*a*, *K0*)
Convert a polynomial to *dtype*.

from_QQ_gmpy(*a*, *K0*)
Convert a GMPY *mpq* object to *dtype*.

from_QQ_python(*a*, *K0*)
Convert a Python *Fraction* object to *dtype*.

from_RealField(*a*, *K0*)
Convert a mpmath *mpf* object to *dtype*.

from_ZZ_gmpy(*a*, *K0*)
Convert a GMPY *mpz* object to *dtype*.

from_ZZ_python(*a*, *K0*)
Convert a Python *int* object to *dtype*.

from_diofant(*a*)
Convert Diofant's expression to *dtype*.

get_ring()
Returns a field associated with *self*.

is_negative(*a*)
Returns True if $LC(a)$ is negative.

is_nonnegative(*a*)
Returns True if $LC(a)$ is non-negative.

is_nonpositive(*a*)
Returns True if $LC(a)$ is non-positive.

is_positive(*a*)
Returns True if $LC(a)$ is positive.

numer(*a*)
Returns numerator of *a*.

to_diofant(*a*)
Convert *a* to a Diofant object.

class diofant.polys.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.

from_diofant(*expr*)
 Convert Diofant's number to dtype.

gcd(*a, b*)
 Returns GCD of a and b.

get_exact()
 Returns an exact domain associated with self.

get_ring()
 Returns a ring associated with self.

lcm(*a, b*)
 Returns LCM of a and b.

to_diofant(*element*)
 Convert element to Diofant number.

to_rational(*element, limit=True*)
 Convert a real number to rational number.

class diofant.polys.domains.**ExpressionDomain**
 A class for arbitrary expressions.

class **Expression**(*ex*)
 An arbitrary expression.

ExpressionDomain.**denom**(*a*)
 Returns denominator of a.

ExpressionDomain.**dtype**
 alias of [Expression](#) (page 774)

ExpressionDomain.**from_ExpressionDomain**(*a, K0*)
 Convert a EX object to dtype.

ExpressionDomain.**from_FractionField**(*a, K0*)
 Convert a DMF object to dtype.

ExpressionDomain.**from_PolynomialRing**(*a, K0*)
 Convert a DMP object to dtype.

ExpressionDomain.**from_QQ_gmpy**(*a, K0*)
 Convert a GMPY mpq object to dtype.

ExpressionDomain.**from_QQ_python**(*a, K0*)
 Convert a Python Fraction object to dtype.

ExpressionDomain.**from_RealField**(*a, K0*)
 Convert a mpmath mpf object to dtype.

ExpressionDomain.**from_ZZ_gmpy**(*a, K0*)
 Convert a GMPY mpz object to dtype.

ExpressionDomain.**from_ZZ_python**(*a, K0*)
 Convert a Python int object to dtype.

ExpressionDomain.**from_diofant**(*a*)
 Convert Diofant's expression to dtype.

ExpressionDomain.get_field()
 Returns a field associated with self.

ExpressionDomain.get_ring()
 Returns a ring associated with self.

ExpressionDomain.is_negative(*a*)
 Returns True if *a* is negative.

ExpressionDomain.is_nonnegative(*a*)
 Returns True if *a* is non-negative.

ExpressionDomain.is_nonpositive(*a*)
 Returns True if *a* is non-positive.

ExpressionDomain.is_positive(*a*)
 Returns True if *a* is positive.

ExpressionDomain.numer(*a*)
 Returns numerator of *a*.

ExpressionDomain.to_diofant(*a*)
 Convert *a* to a Diofant object.

Implementation Domains

class diofant.polys.domains.PythonFiniteField(*mod*, *symmetric=True*)
 Finite field based on Python's integers.

class diofant.polys.domains.GMPYFiniteField(*mod*, *symmetric=True*)
 Finite field based on GMPY integers.

class diofant.polys.domains.PythonIntegerRing
 Integer ring based on Python's int type.

class diofant.polys.domains.GMPYIntegerRing
 Integer ring based on GMPY's mpz type.

class diofant.polys.domains.PythonRationalField
 Rational field based on Python rational number type.

class diofant.polys.domains.GMPYRationalField
 Rational field based on GMPY mpq class.

Level One

class diofant.polys.polyclasses.DMP(*rep*, *dom*, *lev=None*, *ring=None*)
 Dense Multivariate Polynomials over *K*.

LC()
 Returns the leading coefficient of self.

TC()
 Returns the trailing coefficient of self.

abs()
 Make all coefficients in self positive.

add(*other*)
 Add two multivariate polynomials self and other.

add_ground(*c*)
Add an element of the ground domain to *self*.

all_coeffs()
Returns all coefficients from *self*.

all_monoms()
Returns all monomials from *self*.

all_terms()
Returns all terms from a *self*.

cancel(*other, include=True*)
Cancel common factors in a rational function *self/other*.

clear_denoms()
Clear denominators, but keep the ground domain.

coeffs(*order=None*)
Returns all non-zero coefficients from *self* in lex order.

cofactors(*other*)
Returns GCD of *self* and *other* and their cofactors.

compose(*other*)
Computes functional composition of *self* and *other*.

content()
Returns GCD of polynomial coefficients.

convert(*dom*)
Convert the ground domain of *self*.

count_complex_roots(*inf=None, sup=None*)
Return the number of complex roots of *self* in [*inf, sup*].

count_real_roots(*inf=None, sup=None*)
Return the number of real roots of *self* in [*inf, sup*].

decompose()
Computes functional decomposition of *self*.

deflate()
Reduce degree of *self* by mapping x_i^m to y_i .

degree(*j=0*)
Returns the leading degree of *self* in x_j .

degree_list()
Returns a list of degrees of *self*.

diff(*m=1, j=0*)
Computes the *m*-th order derivative of *self* in x_j .

discriminant()
Computes discriminant of *self*.

div(*other*)
Polynomial division with remainder of *self* and *other*.

eject(*dom, front=False*)
Eject selected generators into the ground domain.

eval(*a, j=0*)
Evaluates *self* at the given point *a* in x_j .

exclude()

Remove useless generators from `self`.

Returns the removed generators and the new excluded `self`.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> DMP([[ZZ(1)], [ZZ(1)], [ZZ(2)]]], ZZ).exclude()
([2], DMP([[1], [1, 2]], ZZ, None))
```

exquo(*other*)

Computes polynomial exact quotient of `self` and `other`.

exquo_ground(*c*)

Exact quotient of `self` by a an element of the ground domain.

factor_list()

Returns a list of irreducible factors of `self`.

factor_list_include()

Returns a list of irreducible factors of `self`.

classmethod from_dict(*rep, lev, dom*)

Construct and instance of `cls` from a dict representation.

classmethod from_diofant_list(*rep, lev, dom*)

Create an instance of `cls` given a list of Diofant coefficients.

classmethod from_list(*rep, lev, dom*)

Create an instance of `cls` given a list of native coefficients.

gcd(*other*)

Returns polynomial GCD of `self` and `other`.

gcdex(*other*)

Extended Euclidean algorithm, if univariate.

gff_list()

Computes greatest factorial factorization of `self`.

half_gcdex(*other*)

Half extended Euclidean algorithm, if univariate.

homogeneous_order()

Returns the homogeneous order of `self`.

homogenize(*s*)

Return homogeneous polynomial of `self`

inject(*front=False*)

Inject ground domain generators into `self`.

integrate(*m=1, j=0*)

Computes the *m*-th order indefinite integral of `self` in `xj`.

intervals(*all=False, eps=None, inf=None, sup=None, fast=False, sqf=False*)

Compute isolating intervals for roots of `self`.

invert(*other*)

Invert `self` modulo `other`, if possible.

is_cyclotomic
Returns True if `self` is a cyclotomic polynomial.

is_ground
Returns True if `self` is an element of the ground domain.

is_homogeneous
Returns True if `self` is a homogeneous polynomial.

is_irreducible
Returns True if `self` has no factors over its domain.

is_linear
Returns True if `self` is linear in all its variables.

is_monic
Returns True if the leading coefficient of `self` is one.

is_monomial
Returns True if `self` is zero or has only one term.

is_one
Returns True if `self` is a unit polynomial.

is_primitive
Returns True if the GCD of the coefficients of `self` is one.

is_quadratic
Returns True if `self` is quadratic in all its variables.

is_sqf
Returns True if `self` is a square-free polynomial.

is_zero
Returns True if `self` is a zero polynomial.

l1_norm()
Returns l1 norm of `self`.

lcm(*other*)
Returns polynomial LCM of `self` and `other`.

lift()
Convert algebraic coefficients to rationals.

max_norm()
Returns maximum norm of `self`.

monic()
Divides all coefficients by `LC(self)`.

monoms(*order=None*)
Returns all non-zero monomials from `self` in lex order.

mul(*other*)
Multiply two multivariate polynomials `f` and `g`.

mul_ground(*c*)
Multiply `self` by a an element of the ground domain.

neg()
Negate all coefficients in `self`.

nth(N*)**
Returns the `n`-th coefficient of `self`.

pdiv(*other*)
Polynomial pseudo-division of *self* and *other*.

per(*rep*, *dom=None*, *kill=False*, *ring=None*)
Create a DMP out of the given representation.

permute(*P*)
Returns a polynomial in $K[x_{P(1)}, \dots, x_{P(n)}]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [[]], ZZ).permute([1, 0, 2])
DMP([[2], []], [[1, 0], []], ZZ, None)
```

```
>>> DMP([[ZZ(2)], [ZZ(1), ZZ(0)]], [[]], ZZ).permute([1, 2, 0])
DMP([[1], []], [[2, 0], []], ZZ, None)
```

pexquo(*other*)
Polynomial exact pseudo-quotient of *self* and *other*.

pow(*n*)
Raise *self* to a non-negative power *n*.

pquo(*other*)
Polynomial pseudo-quotient of *self* and *other*.

prem(*other*)
Polynomial pseudo-remainder of *self* and *other*.

primitive()
Returns content and a primitive form of *self*.

quo(*other*)
Computes polynomial quotient of *self* and *other*.

quo_ground(*c*)
Quotient of *self* by a an element of the ground domain.

refine_root(*s*, *t*, *eps=None*, *steps=None*, *fast=False*)
Refine an isolating interval to the given precision.
eps should be a rational number.

rem(*other*)
Computes polynomial remainder of *self* and *other*.

resultant(*other*, *includePRS=False*)
Computes resultant of *self* and *other* via PRS.

revert(*n*)
Compute $\text{self}^{*(-1)} \bmod x^{*n}$.

shift(*a*)
Efficiently compute Taylor shift $\text{self}(x + a)$.

slice(*m*, *n*, *j=0*)
Take a continuous subsequence of terms of *self*.

sqf_list(*all=False*)
Returns a list of square-free factors of *self*.

sqf_list_include(*all=False*)

Returns a list of square-free factors of *self*.

sqf_norm()

Computes square-free norm of *self*.

sqf_part()

Computes square-free part of *self*.

sqr()

Square a multivariate polynomial *self*.

sturm()

Computes the Sturm sequence of *self*.

sub(*other*)

Subtract two multivariate polynomials *self* and *other*.

sub_ground(*c*)

Subtract an element of the ground domain from *self*.

subresultants(*other*)

Computes subresultant PRS sequence of *self* and *other*.

terms(*order=None*)

Returns all non-zero terms from *self* in lex order.

terms_gcd()

Remove GCD of terms from the polynomial *self*.

to_dict(*zero=False*)

Convert *self* to a dict representation with native coefficients.

to_diofant_dict(*zero=False*)

Convert *self* to a dict representation with Diofant coefficients.

to_exact()

Make the ground domain exact.

to_field()

Make the ground domain a field.

to_ring()

Make the ground domain a ring.

to_tuple()

Convert *self* to a tuple representation with native coefficients.

This is needed for hashing.

total_degree()

Returns the total degree of *self*.

trunc(*p*)

Reduce *self* modulo a constant *p*.

unify(*other*)

Unify representations of two multivariate polynomials.

class `diofant.polys.polyclasses.DMF`(*rep, dom, lev=None, ring=None*)

Dense Multivariate Fractions over *K*.

add(*other*)

Add two multivariate fractions *self* and *other*.

cancel()

Remove common factors from *self.num* and *self.den*.

denom()
Returns the denominator of `self`.

exquo(*other*)
Computes quotient of fractions `self` and `other`.

frac_unify(*other*)
Unify representations of two multivariate fractions.

half_per(*rep, kill=False*)
Create a DMP out of the given representation.

invert(*check=True*)
Computes inverse of a fraction `self`.

is_one
Returns True if `self` is a unit fraction.

is_zero
Returns True if `self` is a zero fraction.

mul(*other*)
Multiply two multivariate fractions `self` and `other`.

neg()
Negate all coefficients in `self`.

numer()
Returns the numerator of `self`.

per(*num, den, cancel=True, kill=False, ring=None*)
Create a DMF out of the given representation.

poly_unify(*other*)
Unify a multivariate fraction and a polynomial.

pow(*n*)
Raise `self` to a non-negative power `n`.

quo(*other*)
Computes quotient of fractions `self` and `other`.

sub(*other*)
Subtract two multivariate fractions `self` and `other`.

class diofant.polys.polyclasses.ANP(*rep, mod, dom*)
Dense Algebraic Number Polynomials over a field.

LC()
Returns the leading coefficient of `self`.

TC()
Returns the trailing coefficient of `self`.

is_ground
Returns True if `self` is an element of the ground domain.

is_one
Returns True if `self` is a unit algebraic number.

is_zero
Returns True if `self` is a zero algebraic number.

pow(*n*)
Raise `self` to a non-negative power `n`.

to_dict()
Convert `self` to a dict representation with native coefficients.

to_diofant_dict()
Convert `self` to a dict representation with Diofant coefficients.

to_diofant_list()
Convert `self` to a list representation with Diofant coefficients.

to_list()
Convert `self` to a list representation with native coefficients.

to_tuple()
Convert `self` to a tuple representation with native coefficients.
This is needed for hashing.

unify(*other*)
Unify representations of two algebraic numbers.

Level Zero

Level zero contains the bulk code of the polynomial manipulation module.

Manipulation of dense, multivariate polynomials

These functions can be used to manipulate polynomials in $K[X_0, \dots, X_u]$. Functions for manipulating multivariate polynomials in the dense representation have the prefix `dmp_`. Functions which only apply to univariate polynomials (i.e. $u = 0$) have the prefix `dup_`. The ground domain K has to be passed explicitly. For many multivariate polynomial manipulation functions also the level u , i.e. the number of generators minus one, has to be passed. (Note that, in many cases, `dup_` versions of functions are available, which may be slightly more efficient.)

Basic manipulation:

`diofant.polys.densebasic.dmp_LC(f, K)`
Return leading coefficient of `f`.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> poly_LC([], ZZ)
0
>>> poly_LC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
1
```

`diofant.polys.densebasic.dmp_TC(f, K)`
Return trailing coefficient of `f`.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> poly_TC([], ZZ)
0
>>> poly_TC([ZZ(1), ZZ(2), ZZ(3)], ZZ)
3
```

`diofant.polys.densebasic.dmp_ground_LC(f, u, K)`
Return the ground leading coefficient.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_LC(f, 2, ZZ)
1
```

`diofant.polys.densebasic.dmp_ground_TC(f, u, K)`
Return the ground trailing coefficient.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[[1], [2, 3]]])
```

```
>>> dmp_ground_TC(f, 2, ZZ)
3
```

`diofant.polys.densebasic.dmp_true_LT(f, u, K)`
Return the leading term $c * x_1^{n_1} \dots x_k^{n_k}$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[4], [2, 0], [3, 0, 0]])
```

```
>>> dmp_true_LT(f, 1, ZZ)
((2, 0), 4)
```

`diofant.polys.densebasic.dmp_degree(f, u)`
Return the leading degree of f in x_{θ} in $K[X]$.

Note that the degree of 0 is negative infinity (the Diofant object -oo).

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_degree([[[]]], 2)
-oo
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree(f, 1)
1
```

`diofant.polys.densebasic.dmp_degree_in(f, j, u)`
Return the leading degree of f in x_j in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[2], [1, 2, 3]])
```

```
>>> dmp_degree_in(f, 0, 1)
1
>>> dmp_degree_in(f, 1, 1)
2
```

`diofant.polys.densebasic.dmp_degree_list(f, u)`
Return a list of degrees of f in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [1, 2, 3]])
>>> dmp_degree_list(f, 1)
(1, 2)
```

`diofant.polys.densebasic.dmp_strip(f, u)`
Remove leading zeros from f in $K[X]$.

Examples

```
>>> dmp_strip([], [0, 1, 2], [1]), 1)
[[0, 1, 2], [1]]
```

`diofant.polys.densebasic.dmp_validate(f, K=None)`
Return the number of levels in f and recursively strip it.

Examples

```
>>> dmp_validate([[1], [0, 1, 2], [1]])
([[1, 2], [1]], 1)
```

```
>>> dmp_validate([[1], 1])
Traceback (most recent call last):
...
ValueError: invalid data structure for a multivariate polynomial
```

`diofant.polys.densebasic.dmp_reverse(f)`
 Compute $x^n * f(1/x)$, i.e.: reverse f in $K[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([1, 2, 3, 0])
>>> dmp_reverse(f)
[3, 2, 1]
```

`diofant.polys.densebasic.dmp_copy(f, u)`
 Create a new copy of a polynomial f in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [1, 2]])
>>> dmp_copy(f, 1)
[[1], [1, 2]]
```

`diofant.polys.densebasic.dmp_to_tuple(f, u)`
 Convert f into a nested tuple of tuples.
 This is needed for hashing. This is similar to `dmp_copy()`.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [1, 2]])
>>> dmp_to_tuple(f, 1)
((1,), (1, 2))
```

`diofant.polys.densebasic.dmp_normal(f, u, K)`
 Normalize a multivariate polynomial in the given domain.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_normal([[[]], [0, 1.5, 2]], 1, ZZ)
[[1, 2]]
```

`diofant.polys.densebasic.dmp_convert(f, u, K0, K1)`
Convert the ground domain of *f* from *K0* to *K1*.

Examples

```
>>> from diofant.polys.rings import ring
>>> from diofant.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> dmp_convert([[R(1)], [R(2)]], 1, R.to_domain(), ZZ)
[[1], [2]]
>>> dmp_convert([[ZZ(1)], [ZZ(2)]], 1, ZZ, R.to_domain())
[[1], [2]]
```

`diofant.polys.densebasic.dmp_from_diofant(f, u, K)`
Convert the ground domain of *f* from Diofant to *K*.

Examples

```
>>> from diofant import Integer
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_from_diofant([[Integer(1)], [Integer(2)]], 1, ZZ) == [[ZZ(1)], [ZZ(2)]]
True
```

`diofant.polys.densebasic.dmp_nth(f, n, u, K)`
Return the *n*-th coefficient of *f* in $K[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [2], [3]])
>>> dmp_nth(f, 0, 1, ZZ)
[3]
>>> dmp_nth(f, 4, 1, ZZ)
[]
```

`diofant.polys.densebasic.dmp_ground_nth(f, N, u, K)`
Return the ground *n*-th coefficient of *f* in $K[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [2, 3]])
>>> dmp_ground_nth(f, (0, 1), 1, ZZ)
2
```

`diofant.polys.densebasic.dmp_zero_p(f, u)`
Return True if f is zero in $K[X]$.

Examples

```
>>> dmp_zero_p([[[[[]]]]], 4)
True
>>> dmp_zero_p([[[[1]]]]], 4)
False
```

`diofant.polys.densebasic.dmp_zero(u)`
Return a multivariate zero.

Examples

```
>>> dmp_zero(4)
[[[[]]]]
```

`diofant.polys.densebasic.dmp_one_p(f, u, K)`
Return True if f is one in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_one_p([[[ZZ(1)]]], 2, ZZ)
True
```

`diofant.polys.densebasic.dmp_one(u, K)`
Return a multivariate one over K .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_one(2, ZZ)
[[[1]]]
```

`diofant.polys.densebasic.dmp_ground_p(f, c, u)`
Return True if f is constant in $K[X]$.

Examples

```
>>> dmp_ground_p([[[3]]], 3, 2)
True
>>> dmp_ground_p([[[4]]], None, 2)
True
```

`diofant.polys.densebasic.dmp_ground(c, u)`
Return a multivariate constant.

Examples

```
>>> dmp_ground(3, 5)
[[[[[[3]]]]]]
>>> dmp_ground(1, -1)
1
```

`diofant.polys.densebasic.dmp_zeros(n, u, K)`
Return a list of multivariate zeros.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_zeros(3, 2, ZZ)
[[[]], [[]], [[]]]
>>> dmp_zeros(3, -1, ZZ)
[0, 0, 0]
```

`diofant.polys.densebasic.dmp_grounds(c, n, u)`
Return a list of multivariate constants.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_grounds(ZZ(4), 3, 2)
[[[4]], [[4]], [[4]]]
>>> dmp_grounds(ZZ(4), 3, -1)
[4, 4, 4]
```

`diofant.polys.densebasic.dmp_negative_p(f, u, K)`
Return True if $LC(f)$ is negative.

Examples

```
>>> from diofant.polys.domains import ZZ
```



```
>>> dmp_negative_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
False
>>> dmp_negative_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
True
```

`diofant.polys.densebasic.dmp_positive_p(f, u, K)`
Return True if LC(f) is positive.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_positive_p([[ZZ(1)], [-ZZ(1)]], 1, ZZ)
True
>>> dmp_positive_p([[-ZZ(1)], [ZZ(1)]], 1, ZZ)
False
```

`diofant.polys.densebasic.dmp_from_dict(f, u, K)`
Create a $K[X]$ polynomial from a dict.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_from_dict({(0, 0): ZZ(3), (0, 1): ZZ(2), (2, 1): ZZ(1)}, 1, ZZ)
[[1, 0], [], [2, 3]]
>>> dmp_from_dict({}, 0, ZZ)
[]
```

`diofant.polys.densebasic.dmp_to_dict(f, u, K=None, zero=False)`
Convert a $K[X]$ polynomial to a dict ```.

Examples

```
>>> dmp_to_dict([[1, 0], [], [2, 3]], 1) == {(0, 0): 3, (0, 1): 2, (2, 1): 1}
True
>>> dmp_to_dict([], 0)
{}
```

`diofant.polys.densebasic.dmp_swap(f, i, j, u, K)`
Transform $K[\dots x_i \dots x_j \dots]$ to $K[\dots x_j \dots x_i \dots]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[[2], [1, 0]], []])
```

```
>>> dmp_swap(f, 0, 1, 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_swap(f, 1, 2, 2, ZZ)
[[[1], [2, 0]], [[]]]
>>> dmp_swap(f, 0, 2, 2, ZZ)
[[[1, 0]], [[2, 0], []]]
```

`diofant.polys.densebasic.dmp_permute(f, P, u, K)`
Return a polynomial in $K[x_{P(1)}, \dots, x_{P(n)}]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[2], [1, 0]], [])
```

```
>>> dmp_permute(f, [1, 0, 2], 2, ZZ)
[[[2], []], [[1, 0], []]]
>>> dmp_permute(f, [1, 2, 0], 2, ZZ)
[[[1], []], [[2, 0], []]]
```

`diofant.polys.densebasic.dmp_nest(f, l, K)`
Return a multivariate value nested *l*-levels.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_nest([[ZZ(1)]], 2, ZZ)
[[[[1]]]]
```

`diofant.polys.densebasic.dmp_raise(f, l, u, K)`
Return a multivariate polynomial raised *l*-levels.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([], [1, 2])
```

```
>>> dmp_raise(f, 2, 1, ZZ)
[[[[[]]], [[1]], [[2]]]
```

`diofant.polys.densebasic.dmp_deflate(f, u, K)`
Map $x_i^{m_i}$ to y_i in a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
```

```
>>> dmp_deflate(f, 1, ZZ)
((2, 3), [[1, 2], [3, 4]])
```

`diofant.polys.densebasic.dmp_multi_deflate(polys, u, K)`
Map $x_i^{*m_i}$ to y_i in a set of polynomials in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1, 0, 0, 2], [], [3, 0, 0, 4]])
>>> g = ZZ.map([[1, 0, 2], [], [3, 0, 4]])
```

```
>>> dmp_multi_deflate((f, g), 1, ZZ)
((2, 1), ([[1, 0, 0, 2], [3, 0, 0, 4]], [[1, 0, 2], [3, 0, 4]]))
```

`diofant.polys.densebasic.dmp_inflate(f, M, u, K)`
Map y_i to $x_i^{*k_i}$ in a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1, 2], [3, 4]])
```

```
>>> dmp_inflate(f, (2, 3), 1, ZZ)
[[1, 0, 0, 2], [], [3, 0, 0, 4]]
```

`diofant.polys.densebasic.dmp_exclude(f, u, K)`
Exclude useless levels from f .
Return the levels excluded, the new excluded f , and the new u .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[[1]], [[1], [2]]])
```

```
>>> dmp_exclude(f, 2, ZZ)
([2], [[1], [1, 2]], 1)
```

`diofant.polys.densebasic.dmp_include(f, J, u, K)`
Include useless levels in f .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1], [1, 2]])
```

```
>>> dmp_include(f, [2], 1, ZZ)
[[[1]], [[1], [2]]]
```

`diofant.polys.densebasic.dmp_inject(f, u, K, front=False)`
 Convert f from $K[X][Y]$ to $K[X, Y]$.

Examples

```
>>> from diofant.polys.rings import ring
>>> from diofant.polys.domains import ZZ
```

```
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> dmp_inject([R(1), x + 2], 0, R.to_domain())
([[1]], [[1], [2]]], 2)
>>> dmp_inject([R(1), x + 2], 0, R.to_domain(), front=True)
([[1]], [[1], [2]]], 2)
```

`diofant.polys.densebasic.dmp_eject(f, u, K, front=False)`
 Convert f from $K[X, Y]$ to $K[X][Y]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dmp_eject([[1]], [[1], [2]]], 2, ZZ['x', 'y'])
[1, x + 2]
```

`diofant.polys.densebasic.dmp_terms_gcd(f, u, K)`
 Remove GCD of terms from f in $K[X]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1, 0], [1, 0, 0], [], []])
```

```
>>> dmp_terms_gcd(f, 1, ZZ)
((2, 1), [[1], [1, 0]])
```

`diofant.polys.densebasic.dmp_list_terms(f, u, K, order=None)`
 List all non-zero terms from f in the given order order.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> f = ZZ.map([[1, 1], [2, 3]])
```

```
>>> dmp_list_terms(f, 1, ZZ)
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
>>> dmp_list_terms(f, 1, ZZ, order='grevlex')
[((1, 1), 1), ((1, 0), 1), ((0, 1), 2), ((0, 0), 3)]
```

`diofant.polys.densebasic.dmp_apply_pairs`(*f*, *g*, *h*, *args*, *u*, *K*)
Apply *h* to pairs of coefficients of *f* and *g*.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> h = lambda x, y, z: 2*x + y - z
```

```
>>> dmp_apply_pairs([[1], [2, 3]], [[3], [2, 1]], h, (1,), 1, ZZ)
[[4], [5, 6]]
```

`diofant.polys.densebasic.dmp_slice`(*f*, *m*, *n*, *u*, *K*)
Take a continuous subsequence of terms of *f* in $K[X]$.

`diofant.polys.densebasic.dup_random`(*n*, *a*, *b*, *K*)
Return a polynomial of degree *n* with coefficients in [*a*,*b*].

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> dup_random(3, -10, 10, ZZ)
[-2, -8, 9, -4]
```

Arithmetic operations:

`diofant.polys.densearith.dmp_add_term`(*f*, *c*, *i*, *u*, *K*)
Add $c(x_{2..x_u})x_0^{**i}$ to *f* in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_term(x*y + 1, 2, 2)
2*x**2 + x*y + 1
```

`diofant.polys.densearith.dmp_sub_term`(*f*, *c*, *i*, *u*, *K*)
Subtract $c(x_{2..x_u})x_0^{**i}$ from *f* in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_term(2*x**2 + x*y + 1, 2, 2)
x*y + 1
```

`diofant.polys.densearith.dmp_mul_term(f, c, i, u, K)`
 Multiply f by $c(x_2..x_u)*x_0^{**i}$ in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_term(x**2*y + x, 3*y, 2)
3*x**4*y**2 + 3*x**3*y
```

`diofant.polys.densearith.dmp_add_ground(f, c, u, K)`
 Add an element of the ground domain to f .

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x + 8
```

`diofant.polys.densearith.dmp_sub_ground(f, c, u, K)`
 Subtract an element of the ground domain from f .

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_ground(x**3 + 2*x**2 + 3*x + 4, ZZ(4))
x**3 + 2*x**2 + 3*x
```

`diofant.polys.densearith.dmp_mul_ground(f, c, u, K)`
 Multiply f by a constant value in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul_ground(2*x + 2*y, ZZ(3))
6*x + 6*y
```

`diofant.polys.densearith.dmp_quo_ground(f, c, u, K)`
Quotient by a constant in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, ZZ(2))
x**2*y + x
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo_ground(2*x**2*y + 3*x, QQ(2))
x**2*y + 3/2*x
```

`diofant.polys.densearith.dmp_exquo_ground(f, c, u, K)`
Exact quotient by a constant in $K[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_exquo_ground(x**2*y + 2*x, QQ(2))
1/2*x**2*y + x
```

`diofant.polys.densearith.dup_lshift(f, n, K)`
Efficiently multiply f by x^{**n} in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_lshift(x**2 + 1, 2)
x**4 + x**2
```

`diofant.polys.densearith.dup_rshift(f, n, K)`
Efficiently divide f by x^{**n} in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_rshift(x**4 + x**2, 2)
x**2 + 1
>>> R.dup_rshift(x**4 + x**2 + 2, 2)
x**2 + 1
```

`diofant.polys.densearith.dmp_abs(f, u, K)`
Make all coefficients positive in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_abs(x**2*y - x)
x**2*y + x
```

`diofant.polys.densearith.dmp_neg(f, u, K)`
Negate a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_neg(x**2*y - x)
-x**2*y + x
```

`diofant.polys.densearith.dmp_add(f, g, u, K)`
Add dense polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add(x**2 + y, x**2*y + x)
x**2*y + x**2 + x + y
```

`diofant.polys.densearith.dmp_sub(f, g, u, K)`
Subtract dense polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub(x**2 + y, x**2*y + x)
-x**2*y + x**2 - x + y
```


`diofant.polys.densearith.dmp_add_mul(f, g, h, u, K)`
Returns $f + g \cdot h$ where f, g, h are in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_add_mul(x**2 + y, x, x + 2)
2*x**2 + 2*x + y
```

`diofant.polys.densearith.dmp_sub_mul(f, g, h, u, K)`
Returns $f - g \cdot h$ where f, g, h are in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sub_mul(x**2 + y, x, x + 2)
-2*x + y
```

`diofant.polys.densearith.dmp_mul(f, g, u, K)`
Multiply dense polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_mul(x*y + 1, x)
x**2*y + x
```

`diofant.polys.densearith.dmp_sqr(f, u, K)`
Square dense polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_sqr(x**2 + x*y + y**2)
x**4 + 2*x**3*y + 3*x**2*y**2 + 2*x*y**3 + y**4
```

`diofant.polys.densearith.dmp_pow(f, n, u, K)`
Raise f to the n -th power in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pow(x*y + 1, 3)
x**3*y**3 + 3*x**2*y**2 + 3*x*y + 1
```

`diofant.polys.densearith.dmp_pdiv(f, g, u, K)`
 Polynomial pseudo-division in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_pdiv(x**2 + x*y, 2*x + 2)
(2*x + 2*y - 2, -4*y + 4)
```

`diofant.polys.densearith.dmp_prem(f, g, u, K)`
 Polynomial pseudo-remainder in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_prem(x**2 + x*y, 2*x + 2)
-4*y + 4
```

`diofant.polys.densearith.dmp_pquo(f, g, u, K)`
 Polynomial exact pseudo-quotient in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

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

```
>>> R.dmp_pquo(f, g)
2*x
```

```
>>> R.dmp_pquo(f, h)
2*x + 2*y - 2
```

`diofant.polys.densearith.dmp_pexquo(f, g, u, K)`
 Polynomial pseudo-quotient in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

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

```
>>> R.dmp_pexquo(f, g)
2*x
```

```
>>> R.dmp_pexquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`diofant.polys.densearith.dmp_rr_div(f, g, u, K)`
Multivariate division with remainder over a ring.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_rr_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

`diofant.polys.densearith.dmp_ff_div(f, g, u, K)`
Polynomial division with remainder over a field.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_ff_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`diofant.polys.densearith.dmp_div(f, g, u, K)`
Polynomial division with remainder in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(0, x**2 + x*y)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_div(x**2 + x*y, 2*x + 2)
(1/2*x + 1/2*y - 1/2, -y + 1)
```

`diofant.polys.densearith.dmp_rem(f, g, u, K)`
Returns polynomial remainder in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
x**2 + x*y
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_rem(x**2 + x*y, 2*x + 2)
-y + 1
```

`diofant.polys.densearith.dmp_quo(f, g, u, K)`
Returns exact polynomial quotient in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
0
```

```
>>> R, x,y = ring("x,y", QQ)
>>> R.dmp_quo(x**2 + x*y, 2*x + 2)
1/2*x + 1/2*y - 1/2
```

`diofant.polys.densearith.dmp_exquo(f, g, u, K)`
Returns polynomial quotient in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

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

```
>>> R.dmp_exquo(f, g)
x
```

```
>>> R.dmp_exquo(f, h)
Traceback (most recent call last):
...
ExactQuotientFailed: [[2], [2]] does not divide [[1], [1, 0], []]
```

`diofant.polys.densearith.dmp_max_norm(f, u, K)`
Returns maximum norm of a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_max_norm(2*x*y - x - 3)
3
```

`diofant.polys.densearith.dmp_l1_norm(f, u, K)`
Returns l1 norm of a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_l1_norm(2*x*y - x - 3)
6
```

`diofant.polys.densearith.dmp_expand(polys, u, K)`
Multiply together several polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_expand([x**2 + y**2, x + 1])
x**3 + x**2 + x*y**2 + y**2
```

Further tools:

`diofant.polys.densetools.dmp_integrate(f, m, u, K)`
Computes the indefinite integral of f in x_{-0} in $K[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate(x + 2*y, 1)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate(x + 2*y, 2)
1/6*x**3 + x**2*y
```

`diofant.polys.densetools.dmp_integrate_in(f, m, j, u, K)`
 Computes the indefinite integral of f in x_j in $K[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> R.dmp_integrate_in(x + 2*y, 1, 0)
1/2*x**2 + 2*x*y
>>> R.dmp_integrate_in(x + 2*y, 1, 1)
x*y + y**2
```

`diofant.polys.densetools.dmp_diff(f, m, u, K)`
 m -th order derivative in x_u of a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff(f, 1)
y**2 + 2*y + 3
>>> R.dmp_diff(f, 2)
0
```

`diofant.polys.densetools.dmp_diff_in(f, m, j, u, K)`
 m -th order derivative in x_j of a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_in(f, 1, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_in(f, 1, 1)
2*x*y + 2*x + 4*y + 3
```

`diofant.polys.densetools.dmp_eval(f, a, u, K)`
 Evaluate a polynomial at $x_u = a$ in $K[X]$ using the Horner scheme.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_eval(2*x*y + 3*x + y + 2, 2)
5*y + 8
```

`diofant.polys.densetools.dmp_eval_in(f, a, j, u, K)`

Evaluate a polynomial at $x_j = a$ in $K[X]$ using the Horner scheme.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_in(f, 2, 0)
5*y + 8
>>> R.dmp_eval_in(f, 2, 1)
7*x + 4
```

`diofant.polys.densetools.dmp_eval_tail(f, A, u, K)`

Evaluate a polynomial at $x_j = a_j, \dots$ in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 2*x*y + 3*x + y + 2
```

```
>>> R.dmp_eval_tail(f, [2])
7*x + 4
>>> R.dmp_eval_tail(f, [2, 2])
18
```

`diofant.polys.densetools.dmp_diff_eval_in(f, m, a, j, u, K)`

Differentiate and evaluate a polynomial in x_j at a in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x*y**2 + 2*x*y + 3*x + 2*y**2 + 3*y + 1
```

```
>>> R.dmp_diff_eval_in(f, 1, 2, 0)
y**2 + 2*y + 3
>>> R.dmp_diff_eval_in(f, 1, 2, 1)
6*x + 11
```

`diofant.polys.densetools.dmp_trunc(f, p, u, K)`
Reduce a $K[X]$ polynomial modulo a polynomial p in $K[Y]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
>>> g = (y - 1).drop(x)
```

```
>>> R.dmp_trunc(f, g)
11*x**2 + 11*x + 5
```

`diofant.polys.densetools.dmp_ground_trunc(f, p, u, K)`
Reduce a $K[X]$ polynomial modulo a constant p in K .

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_trunc(f, ZZ(3))
-x**2 - x*y - y
```

`diofant.polys.densetools.dup_monic(f, K)`
Divide all coefficients by $LC(f)$ in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> R.dup_monic(3*x**2 + 6*x + 9)
x**2 + 2*x + 3
```

```
>>> R, x = ring("x", QQ)
>>> R.dup_monic(3*x**2 + 4*x + 2)
x**2 + 4/3*x + 2/3
```

`diofant.polys.densetools.dmp_ground_monic(f, u, K)`
Divide all coefficients by $LC(f)$ in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 3*x**2*y + 6*x**2 + 3*x*y + 9*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 2*x**2 + x*y + 3*y + 1
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 3*x**2*y + 8*x**2 + 5*x*y + 6*x + 2*y + 3
```

```
>>> R.dmp_ground_monic(f)
x**2*y + 8/3*x**2 + 5/3*x*y + 2*x + 2/3*y + 1
```

`diofant.polys.densetools.dup_content(f, K)`
Compute the GCD of coefficients of f in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_content(f)
2
```

`diofant.polys.densetools.dmp_ground_content(f, u, K)`
Compute the GCD of coefficients of f in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

`diofant.polys.densetools.dup_primitive(f, K)`
Compute content and the primitive form of f in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

`diofant.polys.densetools.dmp_ground_primitive(f, u, K)`
Compute content and the primitive form of f in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

`diofant.polys.densetools.dup_extract(f, g, K)`
Extract common content from a pair of polynomials in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_extract(6*x**2 + 12*x + 18, 4*x**2 + 8*x + 12)
(2, 3*x**2 + 6*x + 9, 2*x**2 + 4*x + 6)
```

`diofant.polys.densetools.dmp_ground_extract(f, g, u, K)`
Extract common content from a pair of polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_ground_extract(6*x*y + 12*x + 18, 4*x*y + 8*x + 12)
(2, 3*x*y + 6*x + 9, 2*x*y + 4*x + 6)
```

`diofant.polys.densetools.dup_real_imag(f, K)`
Return bivariate polynomials f_1 and f_2 , such that $f = f_1 + f_2 \cdot I$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)
```

`diofant.polys.densetools.dup_mirror(f, K)`
Evaluate efficiently the composition $f(-x)$ in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

`diofant.polys.densetools.dup_scale(f, a, K)`
Evaluate efficiently composition $f(a \cdot x)$ in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

`diofant.polys.densetools.dup_shift(f, a, K)`
 Evaluate efficiently Taylor shift $f(x + a)$ in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

`diofant.polys.densetools.dup_transform(f, p, q, K)`
 Evaluate functional transformation $q^{*n} * f(p/q)$ in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

`diofant.polys.densetools.dmp_compose(f, g, u, K)`
 Evaluate functional composition $f(g)$ in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

`diofant.polys.densetools.dup_decompose(f, K)`
 Computes functional decomposition of f in $K[x]$.

Given a univariate polynomial f with coefficients in a field of characteristic zero, returns list $[f_1, f_2, \dots, f_n]$, where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and f_2, \dots, f_n are monic and homogeneous polynomials of at least second degree.

Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

$$1. f \circ g = f(x + b) \circ (g - b)$$

$$2. x^{*n} \circ x^{*m} = x^{*m} \circ x^{*n}$$

3. $T_n \circ T_m = T_m \circ T_n$
 where T_n and T_m are Chebyshev polynomials.

References

[R396] (page 1271)

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```

`diofant.polys.densetools.dmp_lift(f, u, K)`
 Convert algebraic coefficients to integers in $K[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> from diofant import I
```

```
>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)
```

```
>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])
```

```
>>> R.dmp_lift(f)
x**8 + 2*x**6 + 9*x**4 - 8*x**2 + 16
```

`diofant.polys.densetools.dup_sign_variations(f, K)`
 Compute the number of sign variations of f in $K[x]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2
```

`diofant.polys.densetools.dmp_clear_denoms(f, u, K0, K1=None, convert=False)`
 Clear denominators, i.e. transform K_0 to K_1 .

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + QQ(1,3)*y + 1
```

```
>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)
```

`diofant.polys.densetools.dmp_revert(f, g, u, K)`
Compute $f^{**}(-1) \bmod x^{**n}$ using Newton iteration.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

Manipulation of dense, univariate polynomials with finite field coefficients

Functions in this module carry the prefix `gf_`, referring to the classical name “Galois Fields” for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

`diofant.polys.galoistools.gf_crt(U, M, K=None)`
Chinese Remainder Theorem.

Given a set of integer residues u_0, \dots, u_n and a set of co-prime integer moduli m_0, \dots, m_n , returns an integer u , such that $u = u_i \bmod m_i$ for $i = 0, \dots, n$.

As an example consider a set of residues $U = [49, 76, 65]$ and a set of moduli $M = [99, 97, 95]$. Then we have:

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.ntheory.modular import solve_congruence

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

See also:

[*diofant.ntheory.modular.crt*](#) (page 266) a higher level crt routine

diofant.ntheory.modular.solve_congruence (page 267)

`diofant.polys.galoistools.gf_crt1(M, K)`

First part of the Chinese Remainder Theorem.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_crt1([99, 97, 95], ZZ)
(912285, [9215, 9405, 9603], [62, 24, 12])
```

`diofant.polys.galoistools.gf_crt2(U, M, p, E, S, K)`

Second part of the Chinese Remainder Theorem.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> U = [49, 76, 65]
>>> M = [99, 97, 95]
>>> p = 912285
>>> E = [9215, 9405, 9603]
>>> S = [62, 24, 12]
```

```
>>> gf_crt2(U, M, p, E, S, ZZ)
639985
```

`diofant.polys.galoistools.gf_int(a, p)`

Coerce $a \bmod p$ to an integer in the range $[-p/2, p/2]$.

Examples

```
>>> gf_int(2, 7)
2
>>> gf_int(5, 7)
-2
```

`diofant.polys.galoistools.gf_degree(f)`

Return the leading degree of f .

Examples

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

`diofant.polys.galoistools.gf_LC(f, K)`

Return the leading coefficient of f .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_LC([3, 0, 1], ZZ)
3
```

`diofant.polys.galoistools.gf_TC(f, K)`
Return the trailing coefficient of *f*.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_TC([3, 0, 1], ZZ)
1
```

`diofant.polys.galoistools.gf_strip(f)`
Remove leading zeros from *f*.

Examples

```
>>> gf_strip([0, 0, 0, 3, 0, 1])
[3, 0, 1]
```

`diofant.polys.galoistools.gf_trunc(f, p)`
Reduce all coefficients modulo *p*.

Examples

```
>>> gf_trunc([7, -2, 3], 5)
[2, 3, 3]
```

`diofant.polys.galoistools.gf_normal(f, p, K)`
Normalize all coefficients in *K*.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]
```

`diofant.polys.galoistools.gf_from_dict(f, p, K)`
Create a GF(*p*) [*x*] polynomial from a dict.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4]
```

`diofant.polys.galoistools.gf_to_dict(f, p, symmetric=True)`
Convert a $\text{GF}(p)[x]$ polynomial to a dict.

Examples

```
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5) == {0: -1, 4: -2, 10: -1}
True
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False) == {0: 4, 4:
↪ 3, 10: 4}
True
```

`diofant.polys.galoistools.gf_from_int_poly(f, p)`
Create a $\text{GF}(p)[x]$ polynomial from $\mathbb{Z}[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3, 3]
```

`diofant.polys.galoistools.gf_to_int_poly(f, p, symmetric=True)`
Convert a $\text{GF}(p)[x]$ polynomial to $\mathbb{Z}[x]$.

Examples

```
>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

`diofant.polys.galoistools.gf_neg(f, p, K)`
Negate a polynomial in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

`diofant.polys.galoistools.gf_add_ground(f, a, p, K)`
Compute $f + a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

`diofant.polys.galoistools.gf_sub_ground(f, a, p, K)`
Compute $f - a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

`diofant.polys.galoistools.gf_mul_ground(f, a, p, K)`
Compute $f * a$ where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

`diofant.polys.galoistools.gf_quo_ground(f, a, p, K)`
Compute f/a where f in $\text{GF}(p)[x]$ and a in $\text{GF}(p)$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]
```

`diofant.polys.galoistools.gf_add(f, g, p, K)`
Add polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]
```

`diofant.polys.galoistools.gf_sub(f, g, p, K)`
Subtract polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]
```

`diofant.polys.galoistools.gf_mul(f, g, p, K)`
Multiply polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]
```

`diofant.polys.galoistools.gf_sqr(f, p, K)`
Square polynomials in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

`diofant.polys.galoistools.gf_add_mul(f, g, h, p, K)`
Returns $f + g*h$ where f, g, h in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

`diofant.polys.galoistools.gf_sub_mul(f, g, h, p, K)`
Compute $f - g*h$ where f, g, h in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```

`diofant.polys.galoistools.gf_expand(F, p, K)`
Expand results of `factor()` (page 685) in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_expand([(3, 2, 4), 1], ([2, 2], 2), ([3, 1], 3), 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

`diofant.polys.galoistools.gf_div(f, g, p, K)`
Division with remainder in $\text{GF}(p)[x]$.

Given univariate polynomials *f* and *g* with coefficients in a finite field with *p* elements, returns polynomials *q* and *r* (quotient and remainder) such that $f = q \cdot g + r$.

Consider polynomials $x^3 + x + 1$ and $x^2 + x$ in $\text{GF}(2)$:

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient $x + 1$ and remainder 1, thus:

```
>>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```

References

[R397] (page 1271), [R398] (page 1271)

`diofant.polys.galoistools.gf_rem(f, g, p, K)`
Compute polynomial remainder in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```

`diofant.polys.galoistools.gf_quo(f, g, p, K)`
Compute exact quotient in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

`diofant.polys.galoistools.gf_exquo(f, g, p, K)`
 Compute polynomial quotient in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

```
>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...
ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

`diofant.polys.galoistools.gf_lshift(f, n, K)`
 Efficiently multiply *f* by x^{**n} .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0, 0]
```

`diofant.polys.galoistools.gf_rshift(f, n, K)`
 Efficiently divide *f* by x^{**n} .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2], [3, 4, 0])
```

`diofant.polys.galoistools.gf_pow(f, n, p, K)`
 Compute f^{**n} in $\text{GF}(p)[x]$ using repeated squaring.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

`diofant.polys.galoistools.gf_pow_mod(f, n, g, p, K)`

Compute f^{**n} in $GF(p)[x]/(g)$ using repeated squaring.

Given polynomials f and g in $GF(p)[x]$ and a non-negative integer n , efficiently computes $f^{**n} \pmod{g}$ i.e. the remainder of f^{**n} from division by g , using the repeated squaring algorithm.

References

[R399] (page 1271)

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

`diofant.polys.galoistools.gf_gcd(f, g, p, K)`

Euclidean Algorithm in $GF(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]
```

`diofant.polys.galoistools.gf_lcm(f, g, p, K)`

Compute polynomial LCM in $GF(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 2, 0, 4]
```

`diofant.polys.galoistools.gf_cofactors(f, g, p, K)`

Compute polynomial GCD and cofactors in $GF(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
([1, 3], [3, 3], [2, 1])
```

`diofant.polys.galoistools.gf_gcdex(f, g, p, K)`
Extended Euclidean Algorithm in $GF(p)[x]$.

Given polynomials f and g in $GF(p)[x]$, computes polynomials s , t and h , such that $h = \gcd(f, g)$ and $s*f + t*g = h$. The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials $f = (x + 7)(x + 1)$, $g = (x + 7)(x^2 + 1)$ in $GF(11)[x]$. Application of Extended Euclidean Algorithm gives:

```
>>> from diofant.polys.domains import ZZ
>>> s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> (s, t, g)
([5, 6], [6], [1, 7])
```

As result we obtained polynomials $s = 5x + 6$ and $t = 6$, and additionally $\gcd(f, g) = x + 7$. This is correct because:

```
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> gf_add(S, T, 11, ZZ) == [1, 7]
True
```

References

[R400] (page 1271)

`diofant.polys.galoistools.gf_monic(f, p, K)`
Compute LC and a monic polynomial in $GF(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

`diofant.polys.galoistools.gf_diff(f, p, K)`
Differentiate polynomial in $GF(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

`diofant.polys.galoistools.gf_eval(f, a, p, K)`
Evaluate $f(a)$ in $\text{GF}(p)$ using Horner scheme.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

`diofant.polys.galoistools.gf_multi_eval(f, A, p, K)`
Evaluate $f(a)$ for a in $[a_1, \dots, a_n]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

`diofant.polys.galoistools.gf_compose(f, g, p, K)`
Compute polynomial composition $f(g)$ in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]
```

`diofant.polys.galoistools.gf_compose_mod(g, h, f, p, K)`
Compute polynomial composition $g(h)$ in $\text{GF}(p)[x]/(f)$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]), 5, ZZ)
[4]
```


`diofant.polys.galoistools.gf_trace_map(a, b, c, n, f, p, K)`

Compute polynomial trace map in $\overline{GF}(p)[x]/(f)$.

Given a polynomial f in $\overline{GF}(p)[x]$, polynomials a, b, c in the quotient ring $\overline{GF}(p)[x]/(f)$ such that $b = c^{**t} \pmod{f}$ for some positive power t of p , and a positive integer n , returns a mapping:

```
a -> a**t**n, a + a**t + a**t**2 + ... + a**t**n (mod f)
```

In factorization context, $b = x^{**p} \pmod{f}$ and $c = x \pmod{f}$. This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

References

[R401] (page 1271)

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])
```

`diofant.polys.galoistools.gf_random(n, p, K)`

Generate a random polynomial in $\overline{GF}(p)[x]$ of degree n .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 1, 2, 0, 4, 2]
```

`diofant.polys.galoistools.gf_irreducible(n, p, K)`

Generate random irreducible polynomial of degree n in $\overline{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

`diofant.polys.galoistools.gf_irreducible_p(f, p, K)`

Test irreducibility of a polynomial f in $\overline{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

`diofant.polys.galoistools.gf_sqf_p(f, p, K)`
Return True if f is square-free in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 2, 1, 4]), 5, ZZ)
False
```

`diofant.polys.galoistools.gf_sqf_part(f, p, K)`
Return square-free part of a $\text{GF}(p)[x]$ polynomial.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

`diofant.polys.galoistools.gf_sqf_list(f, p, K, all=False)`
Return the square-free decomposition of a $\text{GF}(p)[x]$ polynomial.

Given a polynomial f in $\text{GF}(p)[x]$, returns the leading coefficient of f and a square-free decomposition $f_1^{e_1} f_2^{e_2} \dots f_k^{e_k}$ such that all f_i are monic polynomials and (f_i, f_j) for $i \neq j$ are co-prime and $e_1 \dots e_k$ are given in increasing order. All trivial terms (i.e. $f_i = 1$) aren't included in the output.

Consider polynomial $f = x^{11} + 1$ over $\text{GF}(11)[x]$:

```
>>> from diofant.polys.domains import ZZ
>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)
```

Note that $f'(x) = 0$:

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon doesn't happen in characteristic zero. However we can still compute square-free decomposition of f using `gf_sqf()`:

```
>>> gf_sqf_list(f, 11, ZZ)
(1, [[1, 1], 11])
```

We obtained factorization $f = (x + 1)^{11}$. This is correct because:

```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

References

[R402] (page 1271)

`diofant.polys.galoistools.gf_Qmatrix(f, p, K)`
Calculate Berlekamp's Q matrix.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]
```

```
>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0],
 [0, 4, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 4]]
```

`diofant.polys.galoistools.gf_Qbasis(Q, p, K)`
Compute a basis of the kernel of Q .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)
[[1, 0, 0, 0], [0, 0, 1, 0]]
```

```
>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)
[[1, 0]]
```

`diofant.polys.galoistools.gf_berlekamp(f, p, K)`
Factor a square-free f in $\text{GF}(p)[x]$ for small p .

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 2], [1, 0, 3]]
```

`diofant.polys.galoistools.gf_zassenhaus(f, p, K)`
Factor a square-free *f* in $\text{GF}(p)[x]$ for medium *p*.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_zassenhaus(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`diofant.polys.galoistools.gf_shoup(f, p, K)`
Factor a square-free *f* in $\text{GF}(p)[x]$ for large *p*.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_shoup(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

`diofant.polys.galoistools.gf_factor_sqf(f, p, K, method=None)`
Factor a square-free polynomial *f* in $\text{GF}(p)[x]$.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> gf_factor_sqf(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [[1, 1], [1, 3]])
```

`diofant.polys.galoistools.gf_factor(f, p, K)`
Factor (non square-free) polynomials in $\text{GF}(p)[x]$.

Given a possibly non square-free polynomial *f* in $\text{GF}(p)[x]$, returns its complete factorization into irreducibles:

```
f_1(x)**e_1 f_2(x)**e_2 ... f_d(x)**e_d
```

where each f_i is a monic polynomial and $\text{gcd}(f_i, f_j) == 1$, for $i \neq j$. The result is given as a tuple consisting of the leading coefficient of *f* and a list of factors of *f* with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of *f* and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial $f = (7x + 1)(x + 2)^2$ in $\text{GF}(11)[x]$. We obtain its factorization into irreducibles as follows:

```
>>> from diofant.polys.domains import ZZ
>>> gf_factor(ZZ.map([5, 2, 7, 2]), 11, ZZ)
(5, [(1, 2), 1], [(1, 8), 2])
```

We arrived with factorization $f = 5 (x + 2) (x + 8)**2$. We didn't recover the exact form of the input polynomial because we requested to get monic factors of f and its leading coefficient separately.

Square-free factors of f can be factored into irreducibles over $GF(p)$ using three very different methods:

Berlekamp efficient for very small values of p (usually $p < 25$)

Cantor-Zassenhaus efficient on average input and with "typical" p

Shoup-Kaltofen-Gathen efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set `GF_FACTOR_METHOD` with one of `berlekamp`, `zassenhaus` or `shoup` values.

References

[R403] (page 1271)

`diofant.polys.galoistools.gf_value(f, a)`
Value of polynomial 'f' at 'a' in field R.

Examples

```
>>> gf_value([1, 7, 2, 4], 11)
2204
```

`diofant.polys.galoistools.gf_csolve(f, n)`
To solve $f(x) \text{ congruent } 0 \pmod{n}$.

n is divided into canonical factors and $f(x) \text{ cong } 0 \pmod{p**e}$ will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

References

[R404] (page 1271)

Examples

Solve $[1, 1, 7]$ congruent $0 \pmod{189}$:

```
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```

Manipulation of sparse, distributed polynomials and vectors

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a *sparse* representation.

Sparse polynomials are represented as dictionaries.

`diofant.polys.rings.ring`(*symbols*, *domain*, *order=lex*)

Construct a polynomial ring returning (ring, x_1, \dots, x_n).

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : *Domain* (page 765) or coercible

order : *Order* (page 863) or coercible, optional, defaults to lex

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex
```

```
>>> R, x, y, z = ring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'diofant.polys.rings.PolyElement'>
```

`diofant.polys.rings.xring`(*symbols*, *domain*, *order=lex*)

Construct a polynomial ring returning (ring, (x_1, \dots, x_n)).

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : *Domain* (page 765) or coercible

order : *Order* (page 863) or coercible, optional, defaults to lex

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex
```

```
>>> R, (x, y, z) = xring("x,y,z", ZZ, lex)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'diofant.polys.rings.PolyElement'>
```

`diofant.polys.rings.vring`(*symbols*, *domain*, *order=lex*)

Construct a polynomial ring and inject x_1, \dots, x_n into the global namespace.

Parameters **symbols** : str, Symbol/Expr or sequence of str, Symbol/Expr (non-empty)

domain : *Domain* (page 765) or coercible

order : *Order* (page 863) or coercible, optional, defaults to lex

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex
```

```
>>> vring("x,y,z", ZZ, lex)
Polynomial ring in x, y, z over ZZ with lex order
>>> x + y + z
x + y + z
>>> type(_)
<class 'diofant.polys.rings.PolyElement'>
```

`diofant.polys.rings.sring`(*exprs*, **symbols*, ***options*)

Construct a ring deriving generators and domain from options and input expressions.

Parameters **exprs** : *Expr* (page 59) or sequence of *Expr* (page 59) (sympifiable)

symbols : sequence of *Symbol* (page 81)/*Expr* (page 59)

options : keyword arguments understood by *Options* (page 863)

Examples

```
>>> from diofant.core import symbols
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex
```

```
>>> x, y, z = symbols("x,y,z")
>>> R, f = sring(x + 2*y + 3*z)
>>> R
Polynomial ring in x, y, z over ZZ with lex order
>>> f
x + 2*y + 3*z
>>> type(_)
<class 'diofant.polys.rings.PolyElement'>
```

class `diofant.polys.rings.PolyRing`

Multivariate distributed polynomial ring.

add(**objs*)

Add a sequence of polynomials or containers of polynomials.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.add([ x**2 + 2*i + 3 for i in range(4) ])
4*x**2 + 24
>>> _.factor_list()
(4, [(x**2 + 6, 1)])
```

drop(*gens)

Remove specified generators from this ring.

drop_to_ground(*gens)

Remove specified generators from the ring and inject them into its domain.

index(gen)

Compute index of gen in self.gens.

monomial_basis(i)

Return the ith-basis element.

mul(*objs)

Multiply a sequence of polynomials or containers of polynomials.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> R, x = ring("x", ZZ)
>>> R.mul([ x**2 + 2*i + 3 for i in range(4) ])
x**8 + 24*x**6 + 206*x**4 + 744*x**2 + 945
>>> _.factor_list()
(1, [(x**2 + 3, 1), (x**2 + 5, 1), (x**2 + 7, 1), (x**2 + 9, 1)])
```

class diofant.polys.rings.PolyElement

Element of multivariate distributed polynomial ring.

almosteq(other, tolerance=None)

Approximate equality test for polynomials.

cancel(g)

Cancel common factors in a rational function f/g.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> (2*x**2 - 2).cancel(x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

coeff(element)

Returns the coefficient that stands next to the given monomial.

Parameters element : PolyElement (with is_monomial = True) or 1

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y, z = ring("x,y,z", ZZ)
>>> f = 3*x**2*y - x*y*z + 7*z**3 + 23
```

```
>>> f.coeff(x**2*y)
3
>>> f.coeff(x*y)
0
>>> f.coeff(1)
23
```

`coeffs(order=None)`

Ordered list of polynomial coefficients.

Parameters `order`: *Order* (page 863) or coercible, optional

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.coeffs()
[2, 1]
>>> f.coeffs(grlex)
[1, 2]
```

`compose(x, a=None)`

Computes the functional composition.

`const()`

Returns the constant coefficient.

`content()`

Returns GCD of polynomial's coefficients.

`copy()`

Return a copy of polynomial self.

Polynomials are mutable; if one is interested in preserving a polynomial, and one plans to use inplace operations, one can copy the polynomial. This method makes a shallow copy.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```

>>> R, x, y = ring('x, y', ZZ)
>>> p = (x + y)**2
>>> p1 = p.copy()
>>> p2 = p
>>> p[R.zero_monom] = 3
>>> p
x**2 + 2*x*y + y**2 + 3
>>> p1
x**2 + 2*x*y + y**2
>>> p2
x**2 + 2*x*y + y**2 + 3

```

degree(*x=None*)

The leading degree in *x* or the main variable.

Note that the degree of 0 is negative infinity (the Diofant object -oo).

degrees()

A tuple containing leading degrees in all variables.

Note that the degree of 0 is negative infinity (the Diofant object -oo)

diff(*x*)

Computes partial derivative in *x*.

Examples

```

>>> from diofant.polys.domains import ZZ

```

```

>>> _, x, y = ring("x,y", ZZ)
>>> p = x + x**2*y**3
>>> p.diff(x)
2*x*y**3 + 1

```

div(*fv*)

Division algorithm, see [CLO] p64.

fv array of polynomials return *qv*, *r* such that $\text{self} = \sum(\text{fv}[i]*\text{qv}[i]) + r$

All polynomials are required not to be Laurent polynomials.

Examples

```

>>> from diofant.polys.domains import ZZ

```

```

>>> _, x, y = ring('x, y', ZZ)
>>> f = x**3
>>> f0 = x - y**2
>>> f1 = x - y
>>> qv, r = f.div((f0, f1))
>>> qv[0]
x**2 + x*y**2 + y**4
>>> qv[1]
0
>>> r
y**6

```

imul_num(*c*)

multiply inplace the polynomial self by an element in the coefficient ring, provided self is not one of the generators; else multiply not inplace

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p1 = p.imul_num(3)
>>> p1
3*x + 3*y**2
>>> p1 is p
True
>>> p = x
>>> p1 = p.imul_num(3)
>>> p1
3*x
>>> p1 is p
False
```

itercoeffs()

Iterator over coefficients of a polynomial.

itermonoms()

Iterator over monomials of a polynomial.

iterterms()

Iterator over terms of a polynomial.

leading_expv()

Leading monomial tuple according to the monomial ordering.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y, z = ring('x, y, z', ZZ)
>>> p = x**4 + x**3*y + x**2*z**2 + z**7
>>> p.leading_expv()
(4, 0, 0)
```

leading_monom()

Leading monomial as a polynomial element.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_monom()
x*y
```

leading_term()

Leading term as a polynomial element.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> (3*x*y + y**2).leading_term()
3*x*y
```

listcoeffs()

Unordered list of polynomial coefficients.

listmonoms()

Unordered list of polynomial monomials.

listterms()

Unordered list of polynomial terms.

monic()

Divides all coefficients by the leading coefficient.

monoms (*order=None*)

Ordered list of polynomial monomials.

Parameters **order** : *Order* (page 863) or coercible, optional

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.monoms()
[(2, 3), (1, 7)]
>>> f.monoms(grlex)
[(1, 7), (2, 3)]
```

primitive()

Returns content and a primitive polynomial.

square()

square of a polynomial

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> _, x, y = ring('x, y', ZZ)
>>> p = x + y**2
>>> p.square()
x**2 + 2*x*y**2 + y**4
```

strip_zero()

Eliminate monomials with zero coefficient.

tail_degree(x=None)

The tail degree in x or the main variable.

Note that the degree of 0 is negative infinity (the Diofant object -oo)

tail_degrees()

A tuple containing tail degrees in all variables.

Note that the degree of 0 is negative infinity (the Diofant object -oo)

terms(order=None)

Ordered list of polynomial terms.

Parameters order : *Order* (page 863) or coercible, optional

Examples

```
>>> from diofant.polys.domains import ZZ
>>> from diofant.polys.orderings import lex, grlex
```

```
>>> _, x, y = ring("x, y", ZZ, lex)
>>> f = x*y**7 + 2*x**2*y**3
```

```
>>> f.terms()
[((2, 3), 2), ((1, 7), 1)]
>>> f.terms(grlex)
[((1, 7), 1), ((2, 3), 2)]
```

In commutative algebra, one often studies not only polynomials, but also *modules* over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called f_1, f_2, \dots

diofant.polys.distributedmodules.sdm_monomial_mul(M, X)

Multiply tuple X representing a monomial of $K[X]$ into the tuple M representing a monomial of F .

Examples

Multiplying xy^3 into xf_1 yields $x^2y^3f_1$:

```
>>> sdm_monomial_mul((1, 1, 0), (1, 3))
(1, 2, 3)
```

diofant.polys.distributedmodules.sdm_monomial_deg(M)

Return the total degree of M.

Examples

For example, the total degree of x^2yf_5 is 3:

```
>>> sdm_monomial_deg((5, 2, 1))
3
```

`diofant.polys.distributedmodules.sdm_monomial_divides(A, B)`
Does there exist a (polynomial) monomial X such that $XA = B$?

Examples

Positive examples:

In the following examples, the monomial is given in terms of x, y and the generator(s), f_1, f_2 etc. The tuple form of that monomial is used in the call to `sdm_monomial_divides`. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

$A = f_1$ divides $B = f_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

$A = f_1$ divides $B = x^2yf_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$ divides $B = x^2yf_5$

```
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

$A = f_1$ does not divide $B = f_2$

```
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

$A = xf_1$ does not divide $B = f_1$

```
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

$A = xy^2f_5$ does not divide $B = yf_5$

```
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

`diofant.polys.distributedmodules.sdm_LC(f, K)`
Returns the leading coefficient of f.

`diofant.polys.distributedmodules.sdm_to_dict(f)`
Make a dictionary from a distributed polynomial.

`diofant.polys.distributedmodules.sdm_from_dict(d, O)`

Create an sdm from a dictionary.

Here O is the monomial order to use.

```
>>> from diofant.polys import QQ, lex
```

```
>>> dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
>>> sdm_from_dict(dic, lex)
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

`diofant.polys.distributedmodules.sdm_add(f, g, O, K)`

Add two module elements f, g .

Addition is done over the ground field K , monomials are ordered according to O .

Examples

All examples use lexicographic order.

$$(xyf_1) + (f_2) = f_2 + xyf_1$$

```
>>> from diofant.polys import lex, QQ
```

```
>>> sdm_add([(1, 1, 1), QQ(1)], [(2, 0, 0), QQ(1)], lex, QQ)
[((2, 0, 0), 1), ((1, 1, 1), 1)]
```

$$(xyf_1) + (-xyf_1) = 0'$$

```
>>> sdm_add([(1, 1, 1), QQ(1)], [(1, 1, 1), QQ(-1)], lex, QQ)
[]
```

$$(f_1) + (2f_1) = 3f_1$$

```
>>> sdm_add([(1, 0, 0), QQ(1)], [(1, 0, 0), QQ(2)], lex, QQ)
[((1, 0, 0), 3)]
```

$$(yf_1) + (xf_1) = xf_1 + yf_1$$

```
>>> sdm_add([(1, 0, 1), QQ(1)], [(1, 1, 0), QQ(1)], lex, QQ)
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

`diofant.polys.distributedmodules.sdm_LM(f)`

Returns the leading monomial of f .

Only valid if $f \neq 0$.

Examples

```
>>> from diofant.polys import QQ, lex
```

```
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
>>> sdm_LM(sdm_from_dict(dic, lex))
(4, 0, 1)
```

`diofant.polys.distributedmodules.sdm_LT(f)`

Returns the leading term of f .

Only valid if $f \neq 0$.

Examples

```
>>> from diofant.polys import QQ, lex
```

```
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

`diofant.polys.distributedmodules.sdm_mul_term(f, term, O, K)`

Multiply a distributed module element f by a (polynomial) term $term$.

Multiplication of coefficients is done over the ground field K , and monomials are ordered according to O .

Examples

$0f_1 = 0$

```
>>> from diofant.polys import lex, QQ
```

```
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((0, 0), QQ(0)), lex, QQ)
[]
```

$x0 = 0$

```
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

$(x)(f_1) = xf_1$

```
>>> sdm_mul_term([(1, 0, 0), QQ(1)], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

$(2xy)(3xf_1 + 4yf_2) = 8xy^2f_2 + 6x^2yf_1$

```
>>> f = [(2, 0, 1), QQ(4)], ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

`diofant.polys.distributedmodules.sdm_zero()`

Return the zero module element.

`diofant.polys.distributedmodules.sdm_deg(f)`

Degree of f .

This is the maximum of the degrees of all its monomials. Invalid if f is zero.

Examples

```
>>> sdm_deg([((1, 2, 3), 1), ((10, 0, 1), 1), ((2, 3, 4), 4)])
7
```

`diofant.polys.distributedmodules.sdm_from_vector`(*vec*, *O*, *K*, ***opts*)

Create an sdm from an iterable of expressions.

Coefficients are created in the ground field *K*, and terms are ordered according to monomial order *O*. Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

Examples

```
>>> from diofant.abc import x, y, z
>>> from diofant.polys import QQ, lex
```

```
>>> sdm_from_vector([x**2+y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

`diofant.polys.distributedmodules.sdm_to_vector`(*f*, *gens*, *K*, *n=None*)

Convert sdm *f* into a list of polynomial expressions.

The generators for the polynomial ring are specified via *gens*. The rank of the module is guessed, or passed via *n*. The ground field is assumed to be *K*.

Examples

```
>>> from diofant.abc import x, y, z
>>> from diofant.polys import QQ, lex
>>> f = [((1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0), QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

Polynomial factorization algorithms

Many variants of Euclid's algorithm:

Classical remainder sequence

Let *K* be a field, and consider the ring $K[X]$ of polynomials in a single indeterminate *X* with coefficients in *K*. Given two elements *f* and *g* of $K[X]$ with $g \neq 0$ there are unique polynomials *q* and *r* such that $f = qg + r$ and $\deg(r) < \deg(g)$ or $r = 0$. They are denoted by $\text{quo}(f, g)$ (*quotient*) and $\text{rem}(f, g)$ (*remainder*), so we have the *division identity*

$$f = \text{quo}(f, g)g + \text{rem}(f, g).$$

It follows that every ideal *I* of $K[X]$ is a principal ideal, generated by any element $\neq 0$ of minimum degree (assuming *I* non-zero). In fact, if *g* is such a polynomial and *f* is any element

of I , $\text{rem}(f, g)$ belongs to I as a linear combination of f and g , hence must be zero; therefore f is a multiple of g .

Using this result it is possible to find a **greatest common divisor** (gcd) of any polynomials f, g, \dots in $K[X]$. If I is the ideal formed by all linear combinations of the given polynomials with coefficients in $K[X]$, and d is its generator, then every common divisor of the polynomials also divides d . On the other hand, the given polynomials are multiples of the generator d ; hence d is a gcd of the polynomials, denoted $\text{gcd}(f, g, \dots)$.

An algorithm for the gcd of two polynomials f and g in $K[X]$ can now be obtained as follows. By the division identity, $r = \text{rem}(f, g)$ is in the ideal generated by f and g , as well as f is in the ideal generated by g and r . Hence the ideals generated by the pairs (f, g) and (g, r) are the same. Set $f_0 = f$, $f_1 = g$, and define recursively $f_i = \text{rem}(f_{i-2}, f_{i-1})$ for $i \geq 2$. The recursion ends after a finite number of steps with $f_{k+1} = 0$, since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs (f_{i-1}, f_i) generate the same ideal. In particular, the ideal generated by f and g is generated by f_k alone as $f_{k+1} = 0$. Hence $d = f_k$ is a gcd of f and g .

The sequence of polynomials f_0, f_1, \dots, f_k is called the *Euclidean polynomial remainder sequence* determined by (f, g) because of the analogy with the classical **Euclidean algorithm** for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for d in terms of f and g by using the full division identities to write recursively each f_i as a linear combination of f and g . This leads to an equation

$$d = uf + vg \quad (u, v \in K[X])$$

analogous to **Bézout's identity** in the case of integers.

`diofant.polys.euclidtools.dmp_half_gcdex(f, g, u, K)`
Half extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

`diofant.polys.euclidtools.dmp_gcdex(f, g, u, K)`
Extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

`diofant.polys.euclidtools.dmp_invert(f, g, u, K)`
Compute multiplicative inverse of f modulo g in $F[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x = ring("x", QQ)
```

`diofant.polys.euclidtools.dmp_euclidean_prs(f, g, u, K)`
Euclidean polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Simplified remainder sequences

Assume, as is usual, that the coefficient field K is the field of fractions of an integral domain A . In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If A is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in $K[X]$ is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial f in $K[X]$ can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation $f = cF$ where $c \in K$ is the *content* of f and F is a *primitive* polynomial, i.e., a polynomial in $A[X]$ with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials f and g with their primitive parts. This will only modify $\text{rem}(f, g)$ by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive $\text{gcd}(f, g)$.

This sequence is the *primitive polynomial remainder sequence*. It is an example of *general polynomial remainder sequences* where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

`diofant.polys.euclidtools.dmp_primitive_prs(f, g, u, K)`
Primitive polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called *subresultant polynomials* of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in

the subdomain A , so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in A .

To see how subresultants are associated with remainder sequences recall that all polynomials h in the sequence are linear combinations of the given polynomials f and g

$$h = uf + vg$$

with polynomials u and v in $K[X]$. Moreover, as is seen from the extended Euclidean algorithm, the degrees of u and v are relatively low, with limited growth from step to step.

Let $n = \deg(f)$, and $m = \deg(g)$, and assume $n \geq m$. If $\deg(h) = j < m$, the coefficients of the powers X^k ($k > j$) in the products uf and vg cancel each other. In particular, the products must have the same degree, say, l . Then $\deg(u) = l - n$ and $\deg(v) = l - m$ with a total of $2l - n - m + 2$ coefficients to be determined.

On the other hand, the equality $h = uf + vg$ implies that $l - j$ linear combinations of the coefficients are zero, those associated with the powers X^i ($j < i \leq l$), and one has a given non-zero value, namely the leading coefficient of h .

To satisfy these $l - j + 1$ linear equations the total number of coefficients to be determined cannot be lower than $l - j + 1$, in general. This leads to the inequality $l \geq n + m - j - 1$. Taking $l = n + m - j - 1$, we obtain $\deg(u) = m - j - 1$ and $\deg(v) = n - j - 1$.

In the case $j = 0$ the matrix of the resulting system of linear equations is the **Sylvester matrix** $S(f, g)$ associated to f and g , an $(n + m) \times (n + m)$ matrix with coefficients of f and g as entries. Its determinant is the **resultant** $\text{res}(f, g)$ of the pair (f, g) . It is non-zero if and only if f and g are relatively prime.

For any j in the interval from 0 to m the matrix of the linear system is an $(n + m - 2j) \times (n + m - 2j)$ submatrix of the Sylvester matrix. Its determinant $s_j(f, g)$ is called the j th *scalar subresultant* of f and g .

If $s_j(f, g)$ is not zero, the associated equation $h = uf + vg$ has a unique solution where $\deg(h) = j$ and the leading coefficient of h has any given value; the one with leading coefficient $s_j(f, g)$ is the j th *subresultant polynomial* or, briefly, *subresultant* of the pair (f, g) , and denoted $S_j(f, g)$. This choice guarantees that the remaining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if f and g are in $A[X]$, so is $S_j(f, g)$ as well. This construction of subresultants applies to any j between 0 and m regardless of the value of $s_j(f, g)$; if it is zero, then $\deg(S_j(f, g)) < j$.

The properties of subresultants are as follows. Let $n_0 = \deg(f)$, $n_1 = \deg(g)$, n_2, \dots, n_k be the decreasing sequence of degrees of polynomials in a remainder sequence. Let $0 \leq j \leq n_1$; then

- $s_j(f, g) \neq 0$ if and only if $j = n_i$ for some i .
- $S_j(f, g) \neq 0$ if and only if $j = n_i$ or $j = n_i - 1$ for some i .

Normally, $n_{i-1} - n_i = 1$ for $1 < i \leq k$. If $n_{i-1} - n_i > 1$ for some i (the *abnormal* case), then $S_{n_{i-1}-1}(f, g)$ and $S_{n_i}(f, g)$ are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the *subresultant remainder sequence* by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \leq i \leq k).$$

In the normal case, these are the same as the $S_{n_i}(f, g)$. He also derived expressions for the constants γ_i in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of f_1, \dots, f_{i-1} , working in the field K .

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients γ_i . Their algorithm deals with elements of the domain A exclusively (assuming $f, g \in A[X]$). However, in the abnormal case there was a problem, a division in A which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of f_i is $s_{n_i-2}(f, g)$ (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but $s_{n_k}(f, g)$ which is not needed after finding $f_{k+1} = 0$. Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

`diofant.polys.euclidtools.dmp_inner_subresultants(f, g, u, K)`
 Subresultant PRS algorithm in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729, -216,
↵16]]
```

```
>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

`diofant.polys.euclidtools.dmp_subresultants(f, g, u, K)`
 Computes subresultant PRS of two polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

`diofant.polys.euclidtools.dmp_prs_resultant(f, g, u, K)`
 Resultant algorithm in $K[X]$ using subresultant PRS.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

```
>>> res, prs = R.dmp_prs_resultant(f, g)
```

```
>>> res == b           # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True
```

`diofant.polys.euclidtools.dmp_zz_modular_resultant(f, g, p, u, K)`
 Compute resultant of f and g modulo a prime p .

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_modular_resultant(f, g, 5)
-2*y**2 + 1
```

`diofant.polys.euclidtools.dmp_zz_collins_resultant(f, g, u, K)`
 Collins's modular resultant algorithm in $\mathbb{Z}[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

`diofant.polys.euclidtools.dmp_qq_collins_resultant(f, g, u, K0)`
Collins's modular resultant algorithm in $Q[X]$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_qq_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

`diofant.polys.euclidtools.dmp_resultant(f, g, u, K, includePRS=False)`
Computes resultant of two polynomials in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

`diofant.polys.euclidtools.dmp_discriminant(f, u, K)`
Computes discriminant of a polynomial in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)
```

```
>>> R.dmp_discriminant(x**2*y + x*z + t)
-4*y*t + z**2
```

`diofant.polys.euclidtools.dmp_rr_prs_gcd(f, g, u, K)`
Computes polynomial GCD using subresultants over a ring.
Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

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

```
>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

`diofant.polys.euclidtools.dmp_ff_prs_gcd(f, g, u, K)`
Computes polynomial GCD using subresultants over a field.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

`diofant.polys.euclidtools.dmp_zz_heu_gcd(f, g, u, K)`
Heuristic polynomial GCD in $\mathbb{Z}[X]$.

Given univariate polynomials f and g in $\mathbb{Z}[X]$, returns their GCD and cofactors, i.e. polynomials h , cff and cfg such that:

```
h = gcd(f, g), cff = quo(f, h) and cfg = quo(g, h)
```

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.

The algorithm computes the polynomial GCD by evaluating polynomials f and g at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation proces reduces f and g variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.

References

[R405] (page 1271)

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

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

```
>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

`diofant.polys.euclidtools.dmp_qq_heu_gcd(f, g, u, K0)`

Heuristic polynomial GCD in $Q[X]$.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from diofant.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y
```

```
>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)
```

`diofant.polys.euclidtools.dmp_inner_gcd(f, g, u, K)`

Computes polynomial GCD and cofactors of f and g in $K[X]$.

Returns (h, cff, cfg) such that $a = \gcd(f, g)$, $cff = \text{quo}(f, h)$, and $cfg = \text{quo}(g, h)$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

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

```
>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)
```

`diofant.polys.euclidtools.dmp_gcd(f, g, u, K)`

Computes polynomial GCD of f and g in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

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

```
>>> R.dmp_gcd(f, g)
x + y
```

`diofant.polys.euclidtools.dmp_lcm(f, g, u, K)`
 Computes polynomial LCM of f and g in $K[X]$.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

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

```
>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

`diofant.polys.euclidtools.dmp_content(f, u, K)`
 Returns GCD of multivariate coefficients.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

`diofant.polys.euclidtools.dmp_primitive(f, u, K)`
 Returns multivariate content and a primitive polynomial.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

`diofant.polys.euclidtools.dmp_cancel(f, g, u, K, include=True)`
 Cancel common factors in a rational function f/g .

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:

`diofant.polys.factortools.dmp_trial_division(f, factors, u, K)`
Determine multiplicities of factors using trial division.

`diofant.polys.factortools.dmp_zz_mignotte_bound(f, u, K)`
Mignotte bound for multivariate polynomials in $K[X]$.

`diofant.polys.factortools.dup_zz_hensel_step(m, f, g, h, s, t, K)`
One step in Hensel lifting in $Z[x]$.

Given positive integer m and $Z[x]$ polynomials f , g , h , s and t such that:

```
f == g*h (mod m)
s*g + t*h == 1 (mod m)

lc(f) is not a zero divisor (mod m)
lc(h) == 1

deg(f) == deg(g) + deg(h)
deg(s) < deg(h)
deg(t) < deg(g)
```

returns polynomials G , H , S and T , such that:

```
f == G*H (mod m**2)
S*G + T**H == 1 (mod m**2)
```

References

[R406] (page 1271)

`diofant.polys.factortools.dup_zz_hensel_lift(p, f, f_list, l, K)`
Multifactor Hensel lifting in $Z[x]$.

Given a prime p , polynomial f over $Z[x]$ such that $lc(f)$ is a unit modulo p , monic pair-wise coprime polynomials f_i over $Z[x]$ satisfying:

```
f = lc(f) f_1 ... f_r (mod p)
```

and a positive integer l , returns a list of monic polynomials F_1, F_2, \dots, F_r satisfying:

```
f = lc(f) F_1 ... F_r (mod p**l)
F_i = f_i (mod p), i = 1..r
```

References

[R407] (page 1271)

`diofant.polys.factortools.dup_zz_zassenhaus(f, K)`
Factor primitive square-free polynomials in $Z[x]$.

`diofant.polys.factortools.dup_zz_irreducible_p(f, K)`
Test irreducibility using Eisenstein's criterion.

`diofant.polys.factortools.dup_cyclotomic_p(f, K, irreducible=False)`
Efficiently test if f is a cyclotomic polynomial.

Examples

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```

`diofant.polys.factortools.dup_zz_cyclotomic_poly(n, K)`
Efficiently generate n -th cyclotomic polynomial.

`diofant.polys.factortools.dup_zz_cyclotomic_factor(f, K)`
Efficiently factor polynomials $x^{*n} - 1$ and $x^{*n} + 1$ in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ returns a list of factors of f , provided that f is in the form $x^{*n} - 1$ or $x^{*n} + 1$ for $n \geq 1$. Otherwise returns `None`.

Factorization is performed using cyclotomic decomposition of f , which makes this method much faster than any other direct factorization approach (e.g. Zassenhaus's).

References

[R408] (page 1271)

`diofant.polys.factortools.dup_zz_factor_sqf(f, K)`
Factor square-free (non-primitive) polynomials in $Z[x]$.

`diofant.polys.factortools.dup_zz_factor(f, K)`
Factor (non square-free) polynomials in $Z[x]$.

Given a univariate polynomial f in $Z[x]$ computes its complete factorization f_1, \dots, f_n into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Zassenhaus algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial $f = 2 * x ** 4 - 2$:

```
>>> from diofant.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
>>> R.dup_zz_factor(2*x**4 - 2)
(2, [(x - 1, 1), (x + 1, 1), (x**2 + 1, 1)])
```

In result we got the following factorization:

```
f = 2 (x - 1) (x + 1) (x**2 + 1)
```

Note that this is a complete factorization over integers, however over Gaussian integers we can factor the last term.

By default, polynomials $x^{**n}-1$ and $x^{**n}+1$ are factored using cyclotomic decomposition to speedup computations. To disable this behaviour set `cyclotomic=False`.

References

[R409] (page 1271)

`diofant.polys.factortools.dmp_zz_wang_non_divisors(E, cs, ct, K)`
Wang/EEZ: Compute a set of valid divisors.

`diofant.polys.factortools.dmp_zz_wang_test_points(f, T, ct, A, u, K)`
Wang/EEZ: Test evaluation points for suitability.

`diofant.polys.factortools.dmp_zz_wang_lead_coeffs(f, T, cs, E, H, A, u, K)`
Wang/EEZ: Compute correct leading coefficients.

`diofant.polys.factortools.dmp_zz_diophantine(F, c, A, d, p, u, K)`
Wang/EEZ: Solve multivariate Diophantine equations.

`diofant.polys.factortools.dmp_zz_wang_hensel_lifting(f, H, LC, A, p, u, K)`
Wang/EEZ: Parallel Hensel lifting algorithm.

`diofant.polys.factortools.dmp_zz_wang(f, u, K, mod=None, seed=None)`
Factor primitive square-free polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x_1, \dots, x_n]$, which is primitive and square-free in x_1 , computes factorization of f into irreducibles over integers.

The procedure is based on Wang's Enhanced Extended Zassenhaus algorithm. The algorithm works by viewing f as a univariate polynomial in $Z[x_2, \dots, x_n][x_1]$, for which an evaluation mapping is computed:

```
x_2 -> a_2, ..., x_n -> a_n
```

where a_i , for $i = 2, \dots, n$, are carefully chosen integers. The mapping is used to transform f into a univariate polynomial in $Z[x_1]$, which can be factored efficiently using Zassenhaus algorithm. The last step is to lift univariate factors to obtain true multivariate factors. For this purpose a parallel Hensel lifting procedure is used.

The parameter `seed` is passed to `_randint` and can be used to seed `randint` (when an integer) or (for testing purposes) can be a sequence of numbers.

References

[R410] (page 1271), [R411] (page 1271)

`diofant.polys.factortools.dmp_zz_factor`(f, u, K)
Factor (non square-free) polynomials in $Z[X]$.

Given a multivariate polynomial f in $Z[x]$ computes its complete factorization f_1, \dots, f_n into irreducibles over integers:

```
f = content(f) f_1**k_1 ... f_n**k_n
```

The factorization is computed by reducing the input polynomial into a primitive square-free polynomial and factoring it using Enhanced Extended Zassenhaus (EEZ) algorithm. Trial division is used to recover the multiplicities of factors.

The result is returned as a tuple consisting of:

```
(content(f), [(f_1, k_1), ..., (f_n, k_n)])
```

Consider polynomial $f = 2 * (x ** 2 - y ** 2)$:

```
>>> from diofant.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
>>> R.dmp_zz_factor(2*x**2 - 2*y**2)
(2, [(x - y, 1), (x + y, 1)])
```

In result we got the following factorization:

```
f = 2 (x - y) (x + y)
```

References

[R412] (page 1271)

`diofant.polys.factortools.dmp_ext_factor`(f, u, K)
Factor multivariate polynomials over algebraic number fields.

`diofant.polys.factortools.dup_gf_factor`(f, K)
Factor univariate polynomials over finite fields.

`diofant.polys.factortools.dmp_factor_list`($f, u, K0$)
Factor polynomials into irreducibles in $K[X]$.

`diofant.polys.factortools.dmp_factor_list_include`(f, u, K)
Factor polynomials into irreducibles in $K[X]$.

`diofant.polys.factortools.dmp_irreducible_p`(f, u, K)
Returns True if f has no factors over its domain.

Groebner basis algorithms

Groebner bases can be used to answer many problems in computational commutative algebra. Their computation is rather complicated, and very performance-sensitive. We present here various low-level implementations of Groebner basis computation algorithms; please see the previous section of the manual for usage.

`diofant.polys.groebnertools.groebner(seq, ring, method=None)`

Computes Groebner basis for a set of polynomials in $K[X]$.

Wrapper around the (default) improved Buchberger and the other algorithms for computing Groebner bases. The choice of algorithm can be changed via `method` argument or `setup()` (page 864), where `method` can be either `buchberger` or `f5b`.

`diofant.polys.groebnertools.spoly(p1, p2, ring)`

Compute $\text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p1)*p1 - \text{LCM}(\text{LM}(p1), \text{LM}(p2))/\text{LM}(p2)*p2$ This is the S-poly provided `p1` and `p2` are monic

`diofant.polys.groebnertools.red_groebner(G, ring)`

Compute reduced Groebner basis, from BeckerWeispfenning93, p. 216

Selects a subset of generators, that already generate the ideal and computes a reduced Groebner basis for them.

`diofant.polys.groebnertools.is_groebner(G, ring)`

Check if `G` is a Groebner basis.

`diofant.polys.groebnertools.is_minimal(G, ring)`

Checks if `G` is a minimal Groebner basis.

`diofant.polys.groebnertools.is_reduced(G, ring)`

Checks if `G` is a reduced Groebner basis.

`diofant.polys.fglmtools.matrix_fglm(F, ring, O_to)`

Converts the reduced Groebner basis `F` of a zero-dimensional ideal w.r.t. `O_from` to a reduced Groebner basis w.r.t. `O_to`.

References

[R413] (page 1271)

Groebner basis algorithms for modules are also provided:

`diofant.polys.distributedmodules.sdm_spoly(f, g, O, K, phantom=None)`

Compute the generalized s-polynomial of `f` and `g`.

The ground field is assumed to be `K`, and monomials ordered according to `O`.

This is invalid if either of `f` or `g` is zero.

If the leading terms of `f` and `g` involve different basis elements of `F`, their s-poly is defined to be zero. Otherwise it is a certain linear combination of `f` and `g` in which the leading terms cancel. See [SCA, defn 2.3.6] for details.

If `phantom` is not `None`, it should be a pair of module elements on which to perform the same operation(s) as on `f` and `g`. The in this case both results are returned.

Examples

```
>>> from diofant.polys import QQ, lex
```

```
>>> f = [((2, 1, 1), QQ(1)), ((1, 0, 1), QQ(1))]
>>> g = [((2, 3, 0), QQ(1))]
>>> h = [((1, 2, 3), QQ(1))]
>>> sdm_spoly(f, h, lex, QQ)
[]
```

```
>>> sdm_spoly(f, g, lex, QQ)
[((1, 2, 1), 1)]
```

`diofant.polys.distributedmodules.sdm_ecart(f)`

Compute the ecart of f .

This is defined to be the difference of the total degree of f and the total degree of the leading monomial of f [SCA, defn 2.3.7].

Invalid if f is zero.

Examples

```
>>> sdm_ecart([((1, 2, 3), 1), ((1, 0, 1), 1)])
0
>>> sdm_ecart([((2, 2, 1), 1), ((1, 5, 1), 1)])
3
```

`diofant.polys.distributedmodules.sdm_nf_mora(f, G, O, K, phantom=None)`

Compute a weak normal form of f with respect to G and order O .

The ground field is assumed to be K , and monomials ordered according to O .

Weak normal forms are defined in [SCA, defn 2.3.3]. They are not unique. This function deterministically computes a weak normal form, depending on the order of G .

The most important property of a weak normal form is the following: if R is the ring associated with the monomial ordering (if the ordering is global, we just have $R = K[x_1, \dots, x_n]$, otherwise it is a certain localization thereof), I any ideal of R and G a standard basis for I , then for any $f \in R$, we have $f \in I$ if and only if $NF(f|G) = 0$.

This is the generalized Mora algorithm for computing weak normal forms with respect to arbitrary monomial orders [SCA, algorithm 2.3.9].

If `phantom` is not `None`, it should be a pair of “phantom” arguments on which to perform the same computations as on f , G , both results are then returned.

`diofant.polys.distributedmodules.sdm_groebner(G, NF, O, K, extended=False)`

Compute a minimal standard basis of \overline{G} with respect to order O .

The algorithm uses a normal form NF , for example `sdm_nf_mora`. The ground field is assumed to be K , and monomials ordered according to O .

Let N denote the submodule generated by elements of G . A standard basis for N is a subset S of N , such that $in(S) = in(N)$, where for any subset X of F , $in(X)$ denotes the submodule generated by the initial forms of elements of X . [SCA, defn 2.3.2]

A standard basis is called minimal if no subset of it is a standard basis.

One may show that standard bases are always generating sets.

Minimal standard bases are not unique. This algorithm computes a deterministic result, depending on the particular order of G .

If `extended=True`, also compute the transition matrix from the initial generators to the groebner basis. That is, return a list of coefficient vectors, expressing the elements of the groebner basis in terms of the elements of G .

This functions implements the “sugar” strategy, see

Giovini et al: “One sugar cube, please” OR Selection strategies in Buchberger algorithm.

Exceptions

These are exceptions defined by the polynomials module.

TODO sort and explain

```

class diofant.polys.polyerrors.BasePolynomialError
    Base class for polynomial related exceptions.
class diofant.polys.polyerrors.ExactQuotientFailed(f, g, dom=None)
class diofant.polys.polyerrors.OperationNotSupported(poly, func)
class diofant.polys.polyerrors.HeuristicGCDFailed
class diofant.polys.polyerrors.HomomorphismFailed
class diofant.polys.polyerrors.IsomorphismFailed
class diofant.polys.polyerrors.ExtraneousFactors
class diofant.polys.polyerrors.EvaluationFailed
class diofant.polys.polyerrors.RefinementFailed
class diofant.polys.polyerrors.CoercionFailed
class diofant.polys.polyerrors.NotInvertible
class diofant.polys.polyerrors.NotReversible
class diofant.polys.polyerrors.NotAlgebraic
class diofant.polys.polyerrors.DomainError
class diofant.polys.polyerrors.PolynomialError
class diofant.polys.polyerrors.UnificationFailed
class diofant.polys.polyerrors.GeneratorsNeeded
class diofant.polys.polyerrors.ComputationFailed(func, nargs, exc)
class diofant.polys.polyerrors.GeneratorsError
class diofant.polys.polyerrors.UnivariatePolynomialError
class diofant.polys.polyerrors.MultivariatePolynomialError
class diofant.polys.polyerrors.PolificationFailed(opt, origs, exprs, seq=False)
class diofant.polys.polyerrors.OptionError
class diofant.polys.polyerrors.FlagError

```

Reference

Modular GCD

`diofant.polys.modulargcd.modgcd_univariate(f, g)`

Computes the GCD of two polynomials in $\mathbb{Z}[x]$ using a modular algorithm.

The algorithm computes the GCD of two univariate integer polynomials f and g by computing the GCD in $\mathbb{Z}_p[x]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. Trial division is only made for candidates which are very likely the desired GCD.

Parameters f : PolyElement

univariate integer polynomial

g : PolyElement

univariate integer polynomial

Returns **h** : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

[R414] (page 1271)

Examples

```
>>> from diofant.polys import ring, ZZ
```

```
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**5 - 1
>>> g = x - 1
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(x - 1, x**4 + x**3 + x**2 + x + 1, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = 6*x**2 - 6
>>> g = 2*x**2 + 4*x + 2
```

```
>>> h, cff, cfg = modgcd_univariate(f, g)
>>> h, cff, cfg
(2*x + 2, 3*x - 3, x + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

`diofant.polys.modulargcd.modgcd_bivariate(f, g)`

Computes the GCD of two polynomials in $\mathbb{Z}[x, y]$ using a modular algorithm.

The algorithm computes the GCD of two bivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x, y]$ for suitable primes p and then reconstructing the coefficients

with the Chinese Remainder Theorem. To compute the bivariate GCD over \mathbb{Z}_p , the polynomials $f \bmod p$ and $g \bmod p$ are evaluated at $y = a$ for certain $a \in \mathbb{Z}_p$ and then their univariate GCD in $\mathbb{Z}_p[x]$ is computed. Interpolating those yields the bivariate GCD in $\mathbb{Z}_p[x, y]$. To verify the result in $\mathbb{Z}[x, y]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters **f** : PolyElement

bivariate integer polynomial

g : PolyElement

bivariate integer polynomial

Returns **h** : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

[R415] (page 1271)

Examples

```
>>> from diofant.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

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

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x**2*y - x**2 - 4*y + 4
>>> g = x + 2
```

```
>>> h, cff, cfg = modgcd_bivariate(f, g)
>>> h, cff, cfg
(x + 2, x*y - x - 2*y + 2, 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

`diofant.polys.modulargcd.modgcd_multivariate(f, g)`

Compute the GCD of two polynomials in $\mathbb{Z}[x_0, \dots, x_{k-1}]$ using a modular algorithm.

The algorithm computes the GCD of two multivariate integer polynomials f and g by calculating the GCD in $\mathbb{Z}_p[x_0, \dots, x_{k-1}]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem. To compute the multivariate GCD over \mathbb{Z}_p the recursive subroutine `_modgcd_multivariate_p` is used. To verify the result in $\mathbb{Z}[x_0, \dots, x_{k-1}]$, trial division is done, but only for candidates which are very likely the desired GCD.

Parameters **f** : PolyElement

multivariate integer polynomial

g : PolyElement

multivariate integer polynomial

Returns **h** : PolyElement

GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

See also:

[_modgcd_multivariate_p](#) (page 858)

References

[R416] (page 1271), [R417] (page 1271)

Examples

```
>>> from diofant.polys import ring, ZZ
```

```
>>> R, x, y = ring("x, y", ZZ)
```

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

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(x + y, x - y, x + y)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y, z = ring("x, y, z", ZZ)
```

```
>>> f = x*z**2 - y*z**2
>>> g = x**2*z + z
```

```
>>> h, cff, cfg = modgcd_multivariate(f, g)
>>> h, cff, cfg
(z, x*z - y*z, x**2 + 1)
```

```
>>> cff * h == f
True
>>> cfg * h == g
True
```

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 $\tilde{m}_\alpha(z)$ of the minimal polynomial m_α in $\mathbb{Z}[z]$ and the primitive associates of f and g in $\mathbb{Z}[x_1, \dots, x_{n-1}][z]/(\tilde{m}_\alpha)[x_0]$. 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 $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0]$ for suitable primes p and then reconstructing the coefficients with the Chinese Remainder Theorem and Rational Reconstruction. The GCD over $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0]$ 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 $\mathbb{Z}_p[z]/(\tilde{m}_\alpha(z))[x_0]$ 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 $\mathbb{Z}_p(x_1, \dots, x_{n-1})[z]/(\tilde{m}_\alpha(z))[x_0]$, are verified by a fraction free trial division.

Apart from the above GCD computation some GCDs in $\mathbb{Q}(\alpha)[x_1, \dots, x_{n-1}]$ have to be calculated, because treating the polynomials as univariate ones can result in a spurious content of the GCD. For this `func_field_modgcd` is called recursively.

Parameters *f*, *g* : PolyElement

polynomials in $\mathbb{Q}(\alpha)[x_0, \dots, x_{n-1}]$

Returns *h* : PolyElement

monic GCD of the polynomials f and g

cff : PolyElement

cofactor of f , i.e. $\frac{f}{h}$

cfg : PolyElement

cofactor of g , i.e. $\frac{g}{h}$

References

[R418] (page 1271)

Examples

```
>>> from diofant.polys import AlgebraicField, QQ, ring
>>> from diofant import sqrt
```

```
>>> A = AlgebraicField(QQ, sqrt(2))
>>> R, x = ring('x', A)
```

```
>>> f = x**2 - 2
>>> g = x + sqrt(2)
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> R, x, y = ring('x, y', A)
```

```
>>> f = x**2 + 2*sqrt(2)*x*y + 2*y**2
>>> g = x + sqrt(2)*y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == x + sqrt(2)*y
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

```
>>> f = x + sqrt(2)*y
>>> g = x + y
```

```
>>> h, cff, cfg = func_field_modgcd(f, g)
```

```
>>> h == R.one
True
>>> cff * h == f
True
>>> cfg * h == g
True
```

`diofant.polys.modulargcd._modgcd_multivariate_p(f, g, p, 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 integer polynomial with coefficients in \mathbb{Z}_p

g : PolyElement

multivariate integer polynomial with coefficients in \mathbb{Z}_p

p : Integer

prime number, modulus of f and g

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

[R419] (page 1271), [R420] (page 1271)

Manipulation of power series

Functions in this module carry the prefix `rs_`, standing for “ring series”. They manipulate finite power series in the sparse representation provided by `polys.ring.ring`.

`diofant.polys.ring_series.rs_trunc(p1, x, prec)`

truncate the series in the x variable with precision `prec`, that is modulo $0(x^{**}prec)$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = x**10 + x**5 + x + 1
>>> rs_trunc(p, x, 12)
x**10 + x**5 + x + 1
```

```
>>> rs_trunc(p, x, 10)
x**5 + x + 1
```

`diofant.polys.ring_series.rs_mul(p1, p2, x, prec)`
product of series modulo $0(x^{**prec})$

`x` is the series variable or its position in the generators.

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p1 = x**2 + 2*x + 1
>>> p2 = x + 1
>>> rs_mul(p1, p2, x, 3)
3*x**2 + 3*x + 1
```

`diofant.polys.ring_series.rs_square(p1, x, prec)`
square modulo $0(x^{**prec})$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = x**2 + 2*x + 1
>>> rs_square(p, x, 3)
6*x**2 + 4*x + 1
```

`diofant.polys.ring_series.rs_pow(p1, n, x, prec)`
return $p1^{**n}$ modulo $0(x^{**prec})$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = x + 1
>>> rs_pow(p, 4, x, 3)
6*x**2 + 4*x + 1
```

`diofant.polys.ring_series.rs_series_inversion(p, x, prec)`
multivariate series inversion $1/p$ modulo $0(x^{**prec})$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x, y = ring('x, y', QQ)
>>> rs_series_inversion(1 + x*y**2, x, 4)
-x**3*y**6 + x**2*y**4 - x*y**2 + 1
>>> rs_series_inversion(1 + x*y**2, y, 4)
-x*y**2 + 1
```

`diofant.polys.ring_series.rs_series_from_list(p, c, x, prec, concur=1)`
 series sum $c[n]*p^{**n}$ modulo $O(x^{**prec})$

reduce the number of multiplication summing concurrently $ax = [1, p, p^{**2}, \dots, p^{**(J-1)}]$ $s = \sum(c[i]*ax[i] \text{ for } i \text{ in range}(r, (r + 1)*J)) * p^{**((K - 1)*J)}$ with $K \geq (n + 1)/J$

See also:

[*diofant.polys.rings.PolyElement.compose*](#) (page 829)

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = x**2 + x + 1
>>> c = [1, 2, 3]
>>> rs_series_from_list(p, c, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> rs_trunc(1 + 2*p + 3*p**2, x, 4)
6*x**3 + 11*x**2 + 8*x + 6
>>> pc = R.from_list(list(reversed(c)))
>>> rs_trunc(pc.compose(x, p), x, 4)
6*x**3 + 11*x**2 + 8*x + 6
```

`diofant.polys.ring_series.rs_integrate(self, x)`
 integrate p with respect to x

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x, y = ring('x, y', QQ)
>>> p = x + x**2*y**3
>>> rs_integrate(p, x)
1/3*x**3*y**3 + 1/2*x**2
```

`diofant.polys.ring_series.rs_log(p, x, prec)`
 logarithm of p modulo $O(x^{**prec})$

Notes

truncation of integral $\int dx p^{*-1}d p/dx$ is used.

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> rs_log(1 + x, x, 8)
1/7*x**7 - 1/6*x**6 + 1/5*x**5 - 1/4*x**4 + 1/3*x**3 - 1/2*x**2 + x
```

`diofant.polys.ring_series.rs_exp(p, x, prec)`
 exponentiation of a series modulo $O(x^{**prec})$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> rs_exp(x**2, x, 7)
1/6*x**6 + 1/2*x**4 + x**2 + 1
```

`diofant.polys.ring_series.rs_newton(p, x, prec)`
 compute the truncated Newton sum of the polynomial p

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = x**2 - 2
>>> rs_newton(p, x, 5)
8*x**4 + 4*x**2 + 2
```

`diofant.polys.ring_series.rs_hadamard_exp(p1, inverse=False)`
 return $\sum f_i/i! * x^{**i}$ from $\sum f_i * x^{**i}$, where x is the first variable.
 If `inverse=True` return $\sum f_i * i! * x^{**i}$

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> p = 1 + x + x**2 + x**3
>>> rs_hadamard_exp(p)
1/6*x**3 + 1/2*x**2 + x + 1
```

`diofant.polys.ring_series.rs_compose_add(p1, p2)`
compute the composed sum prod(p2(x -beta) for beta root of p1)

References

[R421] (page 1271)

Examples

```
>>> from diofant.polys.domains import QQ
>>> from diofant.polys.rings import ring
```

```
>>> R, x = ring('x', QQ)
>>> f = x**2 - 2
>>> g = x**2 - 3
>>> rs_compose_add(f, g)
x**4 - 10*x**2 + 1
```

Undocumented

Many parts of the polys module are still undocumented, and even where there is documentation it is scarce. Please contribute!

class `diofant.polys.polyoptions.Order`
order option to polynomial manipulation functions.

class `diofant.polys.polyoptions.Options(gens, args, flags=None, strict=False)`
Options manager for polynomial manipulation module.

Examples

```
>>> from diofant.polys.domains import ZZ
```

```
>>> from diofant.abc import x, y, z
```

```
>>> Options((x, y, z), {'domain': 'ZZ'}) == {'auto': False, 'domain': ZZ, 'gens':
↪ (x, y, z)}
True
```

```
>>> build_options((x, y, z), {'domain': 'ZZ'}) == {'auto': False, 'domain': ZZ,
↪ 'gens': (x, y, z)}
True
```

Options

- Expand — boolean option
- Gens — option
- Wrt — option
- Sort — option
- Order — option
- Field — boolean option
- Greedy — boolean option
- Domain — option
- Split — boolean option
- Gaussian — boolean option
- Extension — option
- Modulus — option
- Symmetric — boolean option
- Strict — boolean option

Flags

- Auto — boolean flag
- Frac — boolean flag
- Formal — boolean flag
- Polys — boolean flag
- Include — boolean flag
- All — boolean flag
- Gen — flag

`diofant.polys.polyconfig.setup(key, value=None)`
Assign a value to (or reset) a configuration item.

3.13.6 Literature

The following is a non-comprehensive list of publications that were used as a theoretical foundation for implementing polynomials manipulation module.

3.14 Printing System

See the [Printing](#) (page 9) section in Tutorial for introduction into printing.

This guide documents the printing system in Diofant and how it works internally.

3.14.1 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

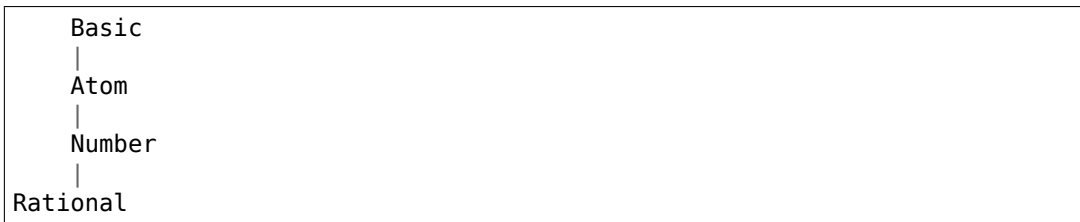
This was the original way of doing printing in diofant. Every class had its own latex, mathml, str and repr methods, but it turned out that it is hard to produce a high quality printer, if all the methods are spread out that far. Therefore all printing code was combined into the different printers, which 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:



then, for `expr=Rational(...)`, in order to dispatch, we will try calling printer methods as shown in the figure below:



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_F00(self, e):  
    ...
```

this should return how FOO instance e is printed

Also, if BAR is a subclass of FOO, `_print_F00(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 import Function, Symbol  
>>> from diofant.printing.str import StrPrinter
```

```
>>> class CustomStrPrinter(StrPrinter):  
...     def _print_Derivative(self, expr):  
...         return str(expr.args[0].func) + "*" * len(expr.args[1:])  
>>> def mystr(e):  
...     return CustomStrPrinter().doprint(e)  
>>> t = Symbol('t')  
>>> x = Function('x')(t)  
>>> print(mystr(x.diff(t, 2)))  
x''
```

printmethod = 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.

3.14.2 PrettyPrinter Class

The pretty printing subsystem is implemented in `diofant.printing.pretty.pretty` by the `PrettyPrinter` class deriving from `Printer`. It relies on the modules `diofant.printing.pretty.stringPict`, and `diofant.printing.pretty.pretty_symbology` for rendering nice-looking formulas.

The module `stringPict` provides a base class `stringPict` and a derived class `prettyForm` that ease the creation and manipulation of formulas that span across multiple lines.

The module `pretty_symbology` provides primitives to construct 2D shapes (`hline`, `vline`, etc) together with a technique to use unicode automatically when possible.

class `diofant.printing.pretty.pretty.PrettyPrinter`(*settings=None*)

`Printer`, which converts an expression into 2D ASCII-art figure.

`diofant.printing.pretty.pretty.pager_print`(*expr, **settings*)

Prints *expr* using the pager, in pretty form.

This invokes a pager command using `pydoc`. Lines are not wrapped automatically. This routine is meant to be used with a pager that allows sideways scrolling, like `less -S`.

Parameters are the same as for `pretty_print`. If you wish to wrap lines, pass `num_columns=None` to auto-detect the width of the terminal.

`diofant.printing.pretty.pretty.pprint`(*expr, **settings*)

`diofant.printing.pretty.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.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.

use_unicode : bool or None, optional

use unicode characters, such as the Greek letter pi instead of the string pi.

full_prec : bool or string, optional
use full precision. Default to “auto”

order : bool or string, optional
set to ‘none’ for long expressions if slow; default is None

3.14.3 CCodePrinter

This class implements C code printing (i.e. it converts Python expressions to strings of C code).

Usage:

```
>>> from diofant.printing import print_ccode
>>> from diofant.functions import sin, cos, Abs
>>> from diofant.abc import x
>>> print_ccode(sin(x)**2 + cos(x)**2)
pow(sin(x), 2) + pow(cos(x), 2)
>>> print_ccode(2*x + cos(x), assign_to="result")
result = 2*x + cos(x);
>>> print_ccode(Abs(x**2))
fabs(pow(x, 2))
```

`diofant.printing.ccode.known_functions = {'log': 'log', 'exp': 'exp', 'gamma': 'tgamma', 'atanh':`

`class diofant.printing.ccode.CCodePrinter(settings={})`
A printer to convert python expressions to strings of c code

`printmethod = '_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)]. See below for examples.

dereference : iterable, optional

An iterable of symbols that should be dereferenced in the printed code expression. These would be values passed by address to the function. For example, if `dereference=[a]`, the resulting code would print `(*a)` instead of `a`.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of `(symbols_to_declare, not_supported_functions, code_text)`. [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting `contract=False` will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

Examples

```
>>> from diofant import ccode, symbols, Rational, sin, ceiling, Abs, Function
>>> x, tau = symbols("x, tau")
>>> ccode((2*tau)**Rational(7, 2))
'8*sqrt(2)*pow(tau, 7.0L/2.0L)'
>>> ccode(sin(x), assign_to="s")
's = sin(x);'
```

Simple custom printing can be defined for certain types by passing a dictionary of `{"type": "function"}` to the `user_functions` kwarg. Alternatively, the dictionary value can be a list of tuples i.e. `[(argument_test, cfunction_string)]`.

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "Abs": [(lambda x: not x.is_integer, "fabs"),
...             (lambda x: x.is_integer, "ABS")],
...     "func": "f"
... }
>>> func = Function('func')
>>> ccode(func(Abs(x) + ceiling(x)), user_functions=custom_functions)
'f(fabs(x) + CEIL(x))'
```

Piecewise expressions are converted into conditionals. If an `assign_to` variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by `(expr, True)` then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from diofant import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(ccode(expr, tau))
if (x > 0) {
tau = x + 1;
}
else {
tau = x;
}
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from diofant import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> ccode(e.rhs, assign_to=e.lhs, contract=False)
'Dy[i] = (y[i + 1] - y[i])/(t[i + 1] - t[i]);'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a `Matrix`:

```
>>> from diofant import Matrix, MatrixSymbol
>>> 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);
```

`diofant.printing.ccode.print_ccode(expr, **settings)`
Prints C representation of the given expression.

3.14.4 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 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 keys are FunctionClass instances and values are their string representations. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)]. See below for examples.

human : bool, optional

If True, the result is a single string that may contain some constant declarations for the number symbols. If False, the same information is returned in a tuple of (symbols_to_declare, not_supported_functions, code_text). [default=True].

contract: bool, optional

If True, Indexed instances are assumed to obey tensor contraction rules and the corresponding nested loops over indices are generated. Setting contract=False will not generate loops, instead the user is responsible to provide values for the indices in the code. [default=True].

source_format : optional

The source format can be either 'fixed' or 'free'. [default='fixed']

standard : integer, optional

The Fortran standard to be followed. This is specified as an integer. Acceptable standards are 66, 77, 90, 95, 2003, and 2008. Default is 77. Note that currently the only distinction internally is between standards before 95, and those 95 and after. This may change later as more features are added.

Examples

```
>>> from diofant import fcode, symbols, Rational, sin, ceiling, floor
>>> x, tau = symbols("x, tau")
>>> fcode((2*tau)**Rational(7, 2))
'      8*sqrt(2.0d0)*tau**(7.0d0/2.0d0)'
>>> fcode(sin(x), assign_to="s")
'      s = sin(x)'
```

Custom printing can be defined for certain types by passing a dictionary of "type" : "function" to the user_functions kwarg. Alternatively, the dictionary value can be a list of tuples i.e. [(argument_test, cfunction_string)].

```
>>> custom_functions = {
...     "ceiling": "CEIL",
...     "floor": [(lambda x: not x.is_integer, "FLOOR1"),
...              (lambda x: x.is_integer, "FLOOR2")]
... }
>>> fcode(floor(x) + ceiling(x), user_functions=custom_functions)
'      CEIL(x) + FLOOR1(x)'
```

Piecewise expressions are converted into conditionals. If an assign_to variable is provided an if statement is created, otherwise the ternary operator is used. Note that if the Piecewise lacks a default term, represented by (expr, True) then an error will be thrown. This is to prevent generating an expression that may not evaluate to anything.

```
>>> from diofant import Piecewise
>>> expr = Piecewise((x + 1, x > 0), (x, True))
>>> print(fcode(expr, tau))
      if (x > 0) then
          tau = x + 1
      else
          tau = x
      end if
```

Support for loops is provided through Indexed types. With `contract=True` these expressions will be turned into loops, whereas `contract=False` will just print the assignment expression that should be looped over:

```
>>> from diofant import Eq, IndexedBase, Idx
>>> len_y = 5
>>> y = IndexedBase('y', shape=(len_y,))
>>> t = IndexedBase('t', shape=(len_y,))
>>> Dy = IndexedBase('Dy', shape=(len_y-1,))
>>> i = Idx('i', len_y-1)
>>> e=Eq(Dy[i], (y[i+1]-y[i])/(t[i+1]-t[i]))
>>> fcode(e.rhs, assign_to=e.lhs, contract=False)
'      Dy(i) = (y(i + 1) - y(i))/(t(i + 1) - t(i))'
```

Matrices are also supported, but a `MatrixSymbol` of the same dimensions must be provided to `assign_to`. Note that any expression that can be generated normally can also exist inside a Matrix:

```
>>> from diofant import Matrix, MatrixSymbol
>>> mat = Matrix([x**2, Piecewise((x + 1, x > 0), (x, True)), sin(x)])
>>> A = MatrixSymbol('A', 3, 1)
>>> print(fcode(mat, A))
      A(1, 1) = x**2
          if (x > 0) then
      A(2, 1) = x + 1
          else
      A(2, 1) = x
          end if
      A(3, 1) = sin(x)
```

`diofant.printing.fcode.print_fcode(expr, **settings)`
Prints the Fortran representation of the given expression.

See `fcode` for the meaning of the optional arguments.

class `diofant.printing.fcode.FCodePrinter(settings={})`
A printer to convert diofant expressions to strings of Fortran code

printmethod = `'_fcode'`

indent_code(code)
Accepts a string of code or a list of code lines

Two basic examples:

```
>>> from diofant import *
>>> x = symbols("x")
>>> fcode(sqrt(1-x**2))
'      sqrt(-x**2 + 1) '
>>> fcode((3 + 4*I)/(1 - conjugate(x)))
'      (cplx(3,4))/(-conjug(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 can not be translated in pure Fortran and (iii) a string of Fortran code. A few examples:

```
>>> fcode(1 - gamma(x)**2, human=False) == (set(), {gamma(x)}, '      -gamma(x)**2 + 1
↳ ')
True
>>> fcode(1 - sin(x)**2, human=False) == (set(), set(), '      -sin(x)**2 + 1')
True
>>> fcode(x - pi**2, human=False) == ({(pi, '3.14159265358979d0')}, set(), '      x -
↳ pi**2')
True
```

3.14.5 Mathematica code printing

```
diofant.printing.mathematica.known_functions = {'log': [(lambda x: log(x)), 'Log']}, 'exp':
```

```
class diofant.printing.mathematica.MCodePrinter(settings={})
```

A printer to convert python expressions to strings of the Wolfram's Mathematica code

```
printmethod = '_mcode'
```

```
doprint(expr)
```

Returns printer's representation for `expr` (as a string)

```
diofant.printing.mathematica.mathematica_code(expr, **settings)
```

Converts an `expr` to a string of the Wolfram Mathematica code

Examples

```
>>> from diofant import mathematica_code as mcode, symbols, sin
>>> x = symbols('x')
>>> mcode(sin(x).series(x).remove0())
'(1/120)*x^5 - 1/6*x^3 + x'
```

3.14.6 LambdaPrinter

This classes implements printing to strings that can be used by the `diofant.utilities.lambdify.lambdify()` (page 1163) function.

class `diofant.printing.lambdarepr.LambdaPrinter(settings=None)`

This printer converts expressions into strings that can be used by `lambdify`.

printmethod = `'_diofantstr'`

`diofant.printing.lambdarepr.LambdaPrinter(expr, **settings)`

Returns a string usable for `lambdifying`.

3.14.7 LatexPrinter

This class implements LaTeX printing. See `diofant.printing.latex`.

`diofant.printing.latex.accepted_latex_functions = ['arcsin', 'arccos', 'arctan', 'sin', 'cos', 'tan', ...]`
`list() -> new empty list`
`list(iterable) -> new list initialized from iterable's items`

class `diofant.printing.latex.LatexPrinter(settings=None)`

printmethod = `'_latex'`

`diofant.printing.latex.LatexPrinter(expr, **settings)`

Convert the given expression to LaTeX representation.

```
>>> from diofant import latex, pi, sin, asin, Integral, Matrix, Rational
>>> from diofant.abc import x, y, 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^{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'))
$\left [ 2 / x, \quad y\right ]$
```

`diofant.printing.latex.print_latex(expr, **settings)`
Prints LaTeX representation of the given expression.

3.14.8 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

[R456] (page 1273)

`printmethod = ‘_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*

diofant.printing.mathml.print_mathml(*expr*, **settings**)**

Prints a pretty representation of the MathML code for *expr*

Examples

```
>>> from diofant.printing.mathml import print_mathml
>>> from diofant.abc import x
>>> print_mathml(x+1)
<apply>
  <plus/>
  <ci>x</ci>
  <cn>1</cn>
</apply>
```

3.14.9 PythonPrinter

This class implements Python printing. Usage:

```
>>> from diofant import print_python, sin
>>> from diofant.abc import x

>>> print_python(5*x**3 + sin(x))
x = Symbol('x')
e = 5*x**3 + sin(x)
```

3.14.10 ReprPrinter

This printer generates executable code. This code satisfies the identity `eval(srepr(expr)) == expr`.

class diofant.printing.repr.ReprPrinter(*settings=None*)

printmethod = `'_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

3.14.11 StrPrinter

This module generates readable representations of Diofant expressions.

class diofant.printing.str.**StrPrinter**(*settings=None*)

printmethod = `'_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

```
>>> from diofant import symbols, Eq, sstr
>>> a, b = symbols('a b')
>>> sstr(Eq(a + b, 0))
'Eq(a + b, 0)'
```

diofant.printing.str.sstrrepr(*expr, **settings*)

return *expr* in mixed str/repr form

i.e. strings are returned in repr form with quotes, and everything else is returned in str form.

This function could be useful for hooking into `sys.displayhook`

3.14.12 Tree Printing

The functions in this module create a representation of an expression as a tree.

diofant.printing.tree.pprint_nodes(*subtrees*)

Prettyprints systems of nodes.

Examples

```
>>> from diofant.printing.tree import pprint_nodes
>>> print(pprint_nodes(["a", "b1\nb2", "c"]))
+-a
+-b1
| b2
+-c
```

diofant.printing.tree.print_node(*node*)

Returns information about the “node”.

This includes class name, string representation and assumptions.

diofant.printing.tree.tree(*node*)

Returns a tree representation of “node” as a string.

It uses `print_node()` together with `pprint_nodes()` on `node.args` recursively.

See also: `print_tree()`

diofant.printing.tree.print_tree(*node*)

Prints a tree representation of “node”.

Examples

```
>>> from diofant.printing import print_tree
>>> from diofant import Symbol
>>> x = Symbol('x', odd=True)
>>> y = Symbol('y', even=True)
>>> print_tree(y**x)
Pow: y**x
commutative: True
+-Symbol: y
| algebraic: True
| commutative: True
| complex: True
| even: True
| extended_real: True
| finite: True
| hermitian: True
| infinite: False
| integer: True
| irrational: False
| noninteger: False
| odd: False
| rational: True
| real: True
| transcendental: False
+-Symbol: x
algebraic: True
commutative: True
complex: True
even: False
extended_real: True
finite: True
hermitian: True
imaginary: False
infinite: False
integer: True
irrational: False
noninteger: False
nonzero: True
odd: True
rational: True
real: True
transcendental: False
zero: False
```

See also: `tree()`

3.14.13 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.

```
>>> from diofant.printing.conventions import split_super_sub
>>> split_super_sub('a_x^1')
('a', ['1'], ['x'])
>>> split_super_sub('var_sub1__sup_sub2')
('var', ['sup'], ['sub1', 'sub2'])
```

CodePrinter

This class is a base class for other classes that implement code-printing functionality, and additionally lists a number of functions that cannot be easily translated to C or Fortran.

class diofant.printing.codeprinter.**CodePrinter**(*settings=None*)

The base class for code-printing subclasses.

printmethod = `'_diofantstr'`

exception diofant.printing.codeprinter.**AssignmentError**

Raised if an assignment variable for a loop is missing.

Precedence

diofant.printing.precedence.**PRECEDENCE** = {'Xor': 10, 'And': 30, 'Mul': 50, 'Mod': 50, 'Atom': 10}

Default precedence values for some basic types.

diofant.printing.precedence.**PRECEDENCE_VALUES** = {'factorial': 60, 'HadamardProduct': 50, 'Neg

A dictionary assigning precedence values to certain classes. These values are treated like they were inherited, so not every single class has to be named here.

diofant.printing.precedence.**PRECEDENCE_FUNCTIONS** = {'Integer': <function precedence_Integer>

Sometimes it's not enough to assign a fixed precedence value to a class. Then a function can be inserted in this dictionary that takes an instance of this class as argument and returns the appropriate precedence value.

diofant.printing.precedence.**precedence**(*item*)

Returns the precedence of a given object.

3.14.14 Pretty-Printing Implementation Helpers

diofant.printing.pretty.pretty_symbology.**U**(*name*)

unicode character by name or None if not found

diofant.printing.pretty.pretty_symbology.**pretty_use_unicode**(*flag=None*)

Set whether pretty-printer should use unicode by default

diofant.printing.pretty.pretty_symbology.**pretty_try_use_unicode**()

See if unicode output is available and leverage it if possible

The following two functions return the Unicode version of the inputted Greek letter.

diofant.printing.pretty.pretty_symbology.**g**(*l*)

diofant.printing.pretty.pretty_symbology.**G**(*l*)

diofant.printing.pretty.pretty_symbology.**greek_letters** = ['alpha', 'beta', 'gamma', 'delta', 'ep

list() -> new empty list list(iterable) -> new list initialized from iterable's items

diofant.printing.pretty.pretty_symbology.**digit_2txt** = {'8': 'EIGHT', '5': 'FIVE', '3': 'THREE',

diofant.printing.pretty.pretty_symbology.**symb_2txt** = {'(': 'LEFT PARENTHESIS', '-': 'MINUS',

The following functions return the Unicode subscript/superscript version of the character.

```
diofant.printing.pretty.pretty_symbology.sub = {'(': '(', '5': '₅', 'chi': 'χ', '-': '-', 't': 'ₜ', 'h': 'ₕ', 'b': 'ₑ'}
```

```
diofant.printing.pretty.pretty_symbology.sup = {'i': 'ⁱ', '5': '⁵', 'n': 'ⁿ', '-': '⁻', '7': '⁷', '6': '⁶', '+': '⁺'}
```

The following functions return Unicode vertical objects.

```
diofant.printing.pretty.pretty_symbology.xobj(symp, length)
```

Construct spatial object of given length.

return: [] of equal-length strings

```
diofant.printing.pretty.pretty_symbology.vobj(symp, height)
```

Construct vertical object of a given height

see: xobj

```
diofant.printing.pretty.pretty_symbology.hobj(symp, width)
```

Construct horizontal object of a given width

see: xobj

The following constants are for rendering roots and fractions.

```
diofant.printing.pretty.pretty_symbology.root = {2: '√', 3: '∛', 4: '∜'}
```

```
diofant.printing.pretty.pretty_symbology.VF(txt)
```

```
diofant.printing.pretty.pretty_symbology.frac = {(1, 2): '½', (1, 3): '⅓', (5, 6): '⅚', (7, 8): '⅞'}
```

The following constants/functions are for rendering atoms and symbols.

```
diofant.printing.pretty.pretty_symbology.xsym(sym)
```

get symbology for a 'character'

```
diofant.printing.pretty.pretty_symbology.atoms_table = {'Reals': '[]', 'ImaginaryUnit': 'i', 'Emp': 'E'}
```

```
diofant.printing.pretty.pretty_symbology.pretty_atom(atom_name, de-  
fault=None)
```

return pretty representation of an atom

```
diofant.printing.pretty.pretty_symbology.pretty_symbol(symp_name)
```

return pretty representation of a symbol

```
diofant.printing.pretty.pretty_symbology.annotated(letter)
```

Return a stylised drawing of the letter *letter*, together with information on how to put annotations (super- and subscripts to the left and to the right) on it.

See `pretty.py` functions `_print_meijerg`, `_print_hyper` on how to use this information.

Prettyprinter by Jurjen Bos. (I hate spammers: mail me at `pietjepuk314` at the reverse of `ku.oc.oohay`). All objects have a method that create a "stringPict", that can be used in the `str` method for pretty printing.

Updates by Jason Gedge (email <my last name> at cs mun ca)

- `terminal_string()` method
- minor fixes and changes (mostly to `prettyForm`)

TODO:

- Allow left/center/right alignment options for above/below and top/center/bottom alignment options for left/right

```
class diofant.printing.pretty.stringpict.stringPict(s, baseline=0)
```

An ASCII picture. The pictures are represented as a list of equal length strings.

above(*args)

Put pictures above this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of bottom picture.

below(*args)

Put pictures under this picture. Returns string, baseline arguments for stringPict. Baseline is baseline of top picture

Examples

```
>>> from diofant.printing.pretty.pretty_symbology import pretty_use_unicode
>>> from diofant.printing.pretty.stringpict import stringPict
>>> f = pretty_use_unicode(flag=False)
>>> print(stringPict("x+3").below(
...     stringPict.LINE, '3')[0])
x+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

```
>>> from diofant.printing.pretty.pretty_symbology import pretty_use_unicode
>>> from diofant.printing.pretty.stringpict import stringPict
>>> f = pretty_use_unicode(flag=False)
>>> print(stringPict("10").right(" + ", stringPict("1\r-\r2", 1))[0])
 1
10 + -
 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.pretty.stringpict.prettyForm(*s*, *baseline*=0, *binding*=0, *unicode*=None)

Extension of the `stringPict` class that knows about basic math applications, optimizing double minus signs.

“Binding” is interpreted as follows:

ATOM	this is an atom: never needs to be parenthesized
FUNC	this is a function application: parenthesize if added (?)
DIV	this is a division: make wider division if divided
POW	this is a power: only parenthesize if exponent
MUL	this is a multiplication: parenthesize if powered
ADD	this is an addition: parenthesize if multiplied or powered
NEG	this is a negative number: optimize if added, parenthesize if multiplied or powered
OPEN	this is an open object: parenthesize if added, multiplied, or powered (example: Piecewise)

3.14.15 dotprint

diofant.printing.dot.dotprint(*expr*, *styles*=[(<class 'diofant.core.basic.Basic'>, {'shape': 'ellipse', 'color': 'blue'}), (<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

```
>>> from diofant.printing.dot import dotprint
>>> from diofant.abc import x
>>> print(dotprint(x + 2))
digraph{

# Graph style
"bgcolor"="transparent"
"ordering"="out"
"rankdir"="TD"

#####
# Nodes #
#####

"Add(Symbol('x'), Integer(2))_" ["color"="black", "label"="Add", "shape"=
↪"ellipse"];
"Integer(2)_(0,)" ["color"="black", "label"="2", "shape"="ellipse"];
"Symbol('x')_(1,)" ["color"="black", "label"="x", "shape"="ellipse"];

#####
# Edges #
#####

"Add(Symbol('x'), Integer(2))_" -> "Integer(2)_(0,)";
"Add(Symbol('x'), Integer(2))_" -> "Symbol('x')_(1,)";
}
```

3.15 Interactive

Helper module for setting up interactive Diofant sessions.

`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.

Examples

```

>>> from diofant.interactive import init_printing
>>> from diofant import Symbol, sqrt
>>> from diofant.abc import x, y, theta
>>> sqrt(5)
sqrt(5)
>>> init_printing(pretty_print=True, no_global=True)
>>> sqrt(5)
 $\sqrt{5}$ 
>>> theta
 $\theta$ 
>>> init_printing(pretty_print=True, use_unicode=False, no_global=True)
>>> 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**
 Add missing Symbol definitions automatically.

class diofant.interactive.session.**IntegerWrapper**
 Wraps all integers in a call to Integer()

3.16 Plotting

3.16.1 Introduction

The plotting module allows you to make 2-dimensional and 3-dimensional plots. Presently the plots are rendered using matplotlib as a backend.

The plotting module has the following functions:

- plot: Plots 2D line plots.
- plot_parametric: Plots 2D parametric plots.
- plot_implicit: Plots 2D implicit and region plots.
- plot3d: Plots 3D plots of functions in two variables.
- plot3d_parametric_line: Plots 3D line plots, defined by a parameter.
- plot3d_parametric_surface: Plots 3D parametric surface plots.

The above functions are only for convenience and ease of use. It is possible to plot any plot by passing the corresponding Series class to Plot as argument.

3.16.2 Plot Class

class diofant.plotting.plot.**Plot**(*args, **kwargs)
 The central class of the plotting module.

For interactive work the function plot is better suited.

This class permits the plotting of diofant expressions using numerous backends (matplotlib, Google charts api, etc).

The figure can contain an arbitrary number of plots of diofant expressions, lists of coordinates of points, etc. Plot has a private attribute `_series` that contains all data series to be plotted (expressions for lines or surfaces, lists of points, etc (all subclasses of `BaseSeries`)). Those data series are instances of classes not imported by `from diofant import *`.

The customization of the figure is on two levels. Global options that concern the figure as a whole (eg title, xlabel, scale, etc) and per-data series options (eg name) and aesthetics (eg. color, point shape, line type, etc.).

The difference between options and aesthetics is that an aesthetic can be a function of the coordinates (or parameters in a parametric plot). The supported values for an aesthetic are: - None (the backend uses default values) - a constant - a function of one variable (the first coordinate or parameter) - a function of two variables (the first and second coordinate or parameters) - a function of three variables (only in nonparametric 3D plots) Their implementation depends on the backend so they may not work in some backends.

If the plot is parametric and the arity of the aesthetic function permits it the aesthetic is calculated over parameters and not over coordinates. If the arity does not permit calculation over parameters the calculation is done over coordinates.

Only cartesian coordinates are supported for the moment, but you can use the parametric plots to plot in polar, spherical and cylindrical coordinates.

The arguments for the constructor `Plot` must be subclasses of `BaseSeries`.

Any global option can be specified as a keyword argument.

The global options for a figure are:

- title : str
- xlabel : str
- ylabel : str
- legend : bool
- xscale : {'linear', 'log'}
- yscale : {'linear', 'log'}
- axis : bool
- axis_center : tuple of two floats or {'center', 'auto'}
- xlim : tuple of two floats
- ylim : tuple of two floats
- aspect_ratio : tuple of two floats or {'auto'}
- autoscale : bool
- margin : float in [0, 1]

The per data series options and aesthetics are: There are none in the base series. See below for options for subclasses.

Some data series support additional aesthetics or options:

`ListSeries`, `LineOver1DRangeSeries`, `Parametric2DLineSeries`, `Parametric3DLineSeries` support the following:

Aesthetics:

- `line_color` : function which returns a float.

options:

- `label` : str
- `steps` : bool
- `integers_only` : bool

`SurfaceOver2DRangeSeries`, `ParametricSurfaceSeries` support the following:

aesthetics:

- `surface_color` : function which returns a float.

append(*arg*)

Adds an element from a plot's series to an existing plot.

See also:

[*extend*](#) (page 888)

Examples

Consider two `Plot` objects, `p1` and `p2`. To add the second plot's first series object to the first, use the `append` method, like so:

```
>>> from diofant import symbols
>>> from diofant.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.append(p2[0])
>>> print(str(p1))
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

extend(*arg*)

Adds all series from another plot.

Examples

Consider two `Plot` objects, `p1` and `p2`. To add the second plot to the first, use the `extend` method, like so:

```
>>> from diofant import symbols
>>> from diofant.plotting import plot
>>> x = symbols('x')
>>> p1 = plot(x*x)
>>> p2 = plot(x)
>>> p1.extend(p2)
>>> print(str(p1))
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
[1]: cartesian line: x for x over (-10.0, 10.0)
```

3.16.3 Plotting Function Reference

`diofant.plotting.plot.plot(*args, **kwargs)`

Plots a function of a single variable and returns an instance of the `Plot` class (also, see the description of the `show` keyword argument below).

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

Single Plot.

```
plot(expr, range, **kwargs)
```

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots with same range.

```
plot(expr1, expr2, ..., range, **kwargs)
```

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots with different ranges.

```
plot((expr1, range), (expr2, range), ..., **kwargs)
```

Range has to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Parameters `“expr”` : Expr

Expression representing the function of single variable

“range”: tuple

(x, 0, 5), A 3-tuple denoting the range of the free variable.

“show”: Boolean, optional

The default value is set to `True`. Set `show` to `False` and the function will not display the plot. The returned instance of the `Plot` class can then be used to save or display the plot by calling the `save()` and `show()` methods respectively.

Arguments for “LineOver1DRangeSeries” class:

“adaptive”: Boolean, optional

The default value is set to `True`. Set `adaptive` to `False` and specify `nb_of_points` if uniform sampling is required.

“depth”: int, optional

Recursion depth of the adaptive algorithm. A depth of value `n` samples a maximum of 2^n points.

“nb_of_points”: int, optional

Used when the `adaptive` is set to `False`. The function is uniformly sampled at `nb_of_points` number of points.

Aesthetics options:

“line_color”: float, optional

Specifies the color for the plot. See `Plot` to see how to set color for the plots.

If there are multiple plots, then the same series series are applied to all the plots. If you want to set these options separately, you can index

the “Plot” object returned and set it.

Arguments for “Plot” class:

“title” : str, optional

Title of the plot. It is set to the latex representation of the expression, if the plot has only one expression.

“xlabel” : str, optional

Label for the x-axis.

“ylabel” : str, optional

Label for the y-axis.

“xscale”: {‘linear’, ‘log’}, optional

Sets the scaling of the x-axis.

“yscale”: {‘linear’, ‘log’}, optional

Sets the scaling if the y-axis.

“axis_center”: tuple of two floats

denoting the coordinates of the center or {‘center’, ‘auto’}

“xlim” : tuple of two floats

denoting the x-axis limits.

“ylim” : tuple of two floats

denoting the y-axis limits.

See also:

Plot (page 886), *diofant.plotting.plot.LineOver1DRangeSeries* (page 899)

Examples

```
>>> from diofant import symbols
>>> from diofant.plotting import plot
>>> x = symbols('x')
```

Single Plot

```
>>> print(str(plot(x**2, (x, -5, 5))))
Plot object containing:
[0]: cartesian line: x**2 for x over (-5.0, 5.0)
```

Multiple plots with single range.

```
>>> print(str(plot(x, x**2, x**3, (x, -5, 5))))
Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> print(str(plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))))
Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

No adaptive sampling.

```
>>> print(str(plot(x**2, adaptive=False, nb_of_points=400)))
Plot object containing:
[0]: cartesian line: x**2 for x over (-10.0, 10.0)
```

`diofant.plotting.plot.plot_parametric(*args, **kwargs)`

Plots a 2D parametric plot.

The plotting uses an adaptive algorithm which samples recursively to accurately plot the plot. The adaptive algorithm uses a random point near the midpoint of two points that has to be further sampled. Hence the same plots can appear slightly different.

Single plot.

```
plot_parametric(expr_x, expr_y, range, **kwargs)
```

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots with same range.

```
plot_parametric((expr1_x, expr1_y), (expr2_x, expr2_y), range, **kwargs)
```

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots with different ranges.

```
plot_parametric((expr_x, expr_y, range), ..., **kwargs)
```

Range has to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Parameters “`expr_x`”: Expr

Expression representing the function along x.

“`expr_y`”: Expr

Expression representing the function along y.

“`range`”: tuple

(u, 0, 5), A 3-tuple denoting the range of the parameter variable.

Arguments for “Parametric2DLineSeries” class:

“`adaptive`”: Boolean, optional

The default value is set to True. Set adaptive to False and specify `nb_of_points` if uniform sampling is required.

“`depth`”: int, optional

Recursion depth of the adaptive algorithm. A depth of value n samples a maximum of 2^n points.

“`nb_of_points`”: int, optional

Used when the adaptive is set to False. The function is uniformly sampled at `nb_of_points` number of points.

Aesthetics:**“line_color“: function which returns a float**

Specifies the color for the plot. See `diofant.plotting.Plot` for more details.

If there are multiple plots, then the same Series arguments are applied to

all the plots. If you want to set these options separately, you can index

the returned “Plot“ object and set it.

Arguments for “Plot“ class:**“xlabel“ : str**

Label for the x-axis.

“ylabel“ : str

Label for the y-axis.

“xscale“: {‘linear’, ‘log’}

Sets the scaling of the x-axis.

“yscale“: {‘linear’, ‘log’}

Sets the scaling if the y-axis.

“axis_center“: tuple of two floats

denoting the coordinates of the center or {‘center’, ‘auto’}

“xlim“ : tuple of two floats

denoting the x-axis limits.

“ylim“ : tuple of two floats

denoting the y-axis limits.

See also:

[Plot](#) (page 886), [Parametric2DLineSeries](#) (page 899)

Examples

```
>>> from diofant import symbols, cos, sin
>>> from diofant.plotting import plot_parametric
>>> u = symbols('u')
```

Single Parametric plot

```
>>> print(str(plot_parametric(cos(u), sin(u), (u, -5, 5))))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
```

Multiple parametric plot with single range.

```
>>> print(str(plot_parametric((cos(u), sin(u)), (u, cos(u)))))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-10.0, 10.0)
[1]: parametric cartesian line: (u, cos(u)) for u over (-10.0, 10.0)
```

Multiple parametric plots.

```
>>> print(str(plot_parametric((cos(u), sin(u), (u, -5, 5)),
... (cos(u), u, (u, -5, 5))))))
Plot object containing:
[0]: parametric cartesian line: (cos(u), sin(u)) for u over (-5.0, 5.0)
[1]: parametric cartesian line: (cos(u), u) for u over (-5.0, 5.0)
```

`diofant.plotting.plot.plot3d(*args, **kwargs)`

Plots a 3D surface plot.

Single plot.

`plot3d(expr, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plot with the same range.

`plot3d(expr1, expr2, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plots with different ranges.

`plot3d((expr1, range_x, range_y), (expr2, range_x, range_y), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Parameters “`expr`”: Expr

Expression representing the function along x.

“range_x”: tuple

(x, 0, 5), A 3-tuple denoting the range of the x variable.

“range_y”: tuple

(y, 0, 5), A 3-tuple denoting the range of the y variable.

Arguments for “SurfaceOver2DRangeSeries” class:

“nb_of_points_x”: int

The x range is sampled uniformly at nb_of_points_x of points.

“nb_of_points_y”: int

The y range is sampled uniformly at nb_of_points_y of points.

Aesthetics:

“surface_color”: Function which returns a float

Specifies the color for the surface of the plot. See `diofant.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied to

all the plots. If you want to set these options separately, you can index

the returned “Plot” object and set it.

Arguments for “Plot” class:

“title” : str
Title of the plot.

See also:

Plot (page 886), *SurfaceOver2DRangeSeries* (page 900)

Examples

```
>>> from diofant import symbols
>>> from diofant.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```
>>> print(str(plot3d(x*y, (x, -5, 5), (y, -5, 5))))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```
>>> print(str(plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with different ranges.

```
>>> print(str(plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)),
... (x*y, (x, -3, 3), (y, -3, 3))))
Plot object containing:
[0]: cartesian surface: x**2 + y**2 for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: x*y for x over (-3.0, 3.0) and y over (-3.0, 3.0)
```

`diofant.plotting.plot.plot3d_parametric_line(*args, **kwargs)`

Plots a 3D parametric line plot.

Single plot.

`plot3d_parametric_line(expr_x,expr_y,expr_z,range,**kwargs)`

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots.

`plot3d_parametric_line((expr_x,expr_y,expr_z,range),...,**kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Parameters “expr_x” : Expr

Expression representing the function along x.

“expr_y” : Expr

Expression representing the function along y.

“expr_z” : Expr

Expression representing the function along z .

“range“: tuple

$(u, 0, 5)$, A 3-tuple denoting the range of the parameter variable.

Arguments for “Parametric3DLineSeries“ class.

“nb_of_points“: int

The range is uniformly sampled at `nb_of_points` number of points.

Aesthetics:

“line_color“: function which returns a float.

Specifies the color for the plot. See `diofant.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied to

all the plots. If you want to set these options separately, you can index

the returned “Plot“ object and set it.

Arguments for “Plot“ class.

“title“ : str

Title of the plot.

See also:

[Plot](#) (page 886), [Parametric3DLineSeries](#) (page 899)

Examples

```
>>> from diofant import symbols, cos, sin
>>> from diofant.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

Single plot.

```
>>> print(str(plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
```

Multiple plots.

```
>>> print(str(plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)),
... (sin(u), u**2, u, (u, -5, 5))))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0, 5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.0)
```

`diofant.plotting.plot.plot3d_parametric_surface(*args, **kwargs)`

Plots a 3D parametric surface plot.

Single plot.

```
plot3d_parametric_surface(expr_x, expr_y, expr_z, range_u, range_v, **kwargs)
```

If the ranges is not specified, then a default range of $(-10, 10)$ is used.

Multiple plots.

```
plot3d_parametric_surface((expr_x, expr_y, expr_z, range_u, range_v), ..., **kwargs)
```

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Parameters “`expr_x`”: Expr

Expression representing the function along x.

“`expr_y`”: Expr

Expression representing the function along y.

“`expr_z`”: Expr

Expression representing the function along z.

“`range_u`”: tuple

(u, 0, 5), A 3-tuple denoting the range of the u variable.

“`range_v`”: tuple

(v, 0, 5), A 3-tuple denoting the range of the v variable.

Arguments for “ParametricSurfaceSeries” class:

“`nb_of_points_u`”: int

The u range is sampled uniformly at `nb_of_points_v` of points

“`nb_of_points_y`”: int

The v range is sampled uniformly at `nb_of_points_y` of points

Aesthetics:

“`surface_color`”: Function which returns a float

Specifies the color for the surface of the plot. See `diofant.plotting.Plot` for more details.

If there are multiple plots, then the same series arguments are applied for

all the plots. If you want to set these options separately, you can index

the returned “Plot” object and set it.

Arguments for “Plot” class:

“`title`” : str

Title of the plot.

See also:

[Plot](#) (page 886), [ParametricSurfaceSeries](#) (page 900)

Examples

```
>>> from diofant import symbols, cos, sin
>>> from diofant.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```
>>> print(str(plot3d_parametric_surface(cos(u + v), sin(u - v), u - v,
...   (u, -5, 5), (v, -5, 5))))
Plot object containing:
[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u over (-5.0, 5.0) and v over (-5.0, 5.0)
```

`diofant.plotting.plot_implicit.plot_implicit(expr, x_var=None, y_var=None, **kwargs)`

A plot function to plot implicit equations / inequalities.

Parameters “`expr`” : Expr

The equation / inequality that is to be plotted.

“`x_var`” : symbol or tuple, optional

symbol to plot on x-axis or tuple giving symbol and range as (symbol, xmin, xmax)

“`y_var`” : symbol or tuple, optional

symbol to plot on y-axis or tuple giving symbol and range as (symbol, ymin, ymax)

If neither “`x_var`” nor “`y_var`” are given then the free symbols in the expression will be assigned in the order they are sorted.

The following keyword arguments can also be used:

“`adaptive`” : Boolean, optional

The default value is set to True. It has to be set to False if you want to use a mesh grid.

“`depth`” : integer

The depth of recursion for adaptive mesh grid. Default value is 0. Takes value in the range (0, 4).

“`points`” : integer

The number of points if adaptive mesh grid is not used. Default value is 200.

“`title`” : str

The title for the plot.

“`xlabel`” : str

The label for the x-axis

“`ylabel`” : string

The label for the y-axis

Aesthetics options:

“`line_color`” : float or str

Specifies the color for the plot. See Plot to see how to set color for the plots.

plot_implicit, by default, uses interval arithmetic to plot functions. If

the expression cannot be plotted using interval arithmetic, it defaults to

a generating a contour using a mesh grid of fixed number of points. By

setting adaptive to False, you can force plot_implicit to use the mesh grid. The mesh grid method can be effective when adaptive plotting using

interval arithmetic, fails to plot with small line width.

Examples

Plot expressions:

```
>>> from diofant import plot_implicit, cos, sin, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression

```
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols

```
>>> p2 = plot_implicit(Eq(x**2 + y**2, 3),
...                    (x, -3, 3), (y, -3, 3))
```

With depth of recursion as argument.

```
>>> p3 = plot_implicit(Eq(x**2 + y**2, 5),
...                    (x, -4, 4), (y, -4, 4), depth = 2)
```

Using mesh grid and not using adaptive meshing.

```
>>> p4 = plot_implicit(Eq(x**2 + y**2, 5),
...                    (x, -5, 5), (y, -2, 2), adaptive=False)
```

Using mesh grid with number of points as input.

```
>>> p5 = plot_implicit(Eq(x**2 + y**2, 5),
...                    (x, -5, 5), (y, -2, 2),
...                    adaptive=False, points=400)
```

Plotting regions.

```
>>> p6 = plot_implicit(y > x**2)
```

Plotting Using boolean conjunctions.

```
>>> p7 = plot_implicit(And(y > x, y > -x))
```

When plotting an expression with a single variable ($y - 1$, for example), specify the x or the y variable explicitly:

```
>>> p8 = plot_implicit(y - 1, y_var=y)
>>> p9 = plot_implicit(x - 1, x_var=x)
```

3.16.4 Series Classes

class diofant.plotting.plot.BaseSeries

Base class for the data objects containing stuff to be plotted.

The backend should check if it supports the data series that it's given. (eg TextBackend supports only LineOver1DRange). It's the backend responsibility to know how to use the class of data series that it's given.

Some data series classes are grouped (using a class attribute like is_2Dline) according to the api they present (based only on convention). The backend is not obliged to use that api (eg. The LineOver1DRange belongs to the is_2Dline group and presents the get_points method, but the TextBackend does not use the get_points method).

class diofant.plotting.plot.Line2DBaseSeries

A base class for 2D lines.

- adding the label, steps and only_integers options
- making is_2Dline true
- defining get_segments and get_color_array

class diofant.plotting.plot.LineOver1DRangeSeries(*expr*, *var_start_end*, ***kwargs*)

Representation for a line consisting of a Diofant expression over a range.

get_segments()

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

- [1] **Adaptive polygonal approximation of parametric curves**, Luiz Henrique de Figueiredo.

class diofant.plotting.plot.Parametric2DLineSeries(*expr_x*, *expr_y*, *var_start_end*, ***kwargs*)

Representation for a line consisting of two parametric diofant expressions over a range.

get_segments()

Adaptively gets segments for plotting.

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

- [1] **Adaptive polygonal approximation of parametric curves**, Luiz Henrique de Figueiredo.

class diofant.plotting.plot.Line3DBaseSeries

A base class for 3D lines.

Most of the stuff is derived from Line2DBaseSeries.

class diofant.plotting.plot.**Parametric3DLineSeries**(*expr_x, expr_y, expr_z, var_start_end, **kwargs*)

Representation for a 3D line consisting of two parametric diofant expressions and a range.

class diofant.plotting.plot.**SurfaceBaseSeries**

A base class for 3D surfaces.

class diofant.plotting.plot.**SurfaceOver2DRangeSeries**(*expr, var_start_end_x, var_start_end_y, **kwargs*)

Representation for a 3D surface consisting of a diofant expression and 2D range.

class diofant.plotting.plot.**ParametricSurfaceSeries**(*expr_x, expr_y, expr_z, var_start_end_u, var_start_end_v, **kwargs*)

Representation for a 3D surface consisting of three parametric diofant expressions and a range.

class diofant.plotting.plot_implicit.**ImplicitSeries**(*expr, var_start_end_x, var_start_end_y, has_equality, use_interval_math, depth, nb_of_points, line_color*)

Representation for Implicit plot

3.17 Term rewriting

Term rewriting is a very general class of functionalities which are used to convert expressions of one type in terms of expressions of different kind. For example expanding, combining and converting expressions apply to term rewriting, and also simplification routines can be included here. Currently Diofant has several functions and basic built-in methods for performing various types of rewriting.

3.17.1 Expanding

The simplest rewrite rule is expanding expressions into a `_sparse_` form. Expanding has several flavors and include expanding complex valued expressions, arithmetic expand of products and powers but also expanding functions in terms of more general functions is possible. Below are listed all currently available expand rules.

Expanding of arithmetic expressions involving products and powers:

```
>>> from diofant import *
>>> x, y, z = symbols('x,y,z')
>>> ((x + y)*(x - y)).expand(basic=True)
x**2 - y**2
>>> ((x + y + z)**2).expand(basic=True)
x**2 + 2*x*y + 2*x*z + y**2 + 2*y*z + z**2
```

Arithmetic expand is done by default in `expand()` so the keyword `basic` can be omitted. However you can set `basic=False` to avoid this type of expand if you use rules described below. This give complete control on what is done with the expression.

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

```
>>> (x + I*y).expand(complex=True)
re(x) + I*re(y) + I*im(x) - im(y)
```

```
>>> sin(x + I*y).expand(complex=True)
sin(re(x) - im(y))*cosh(re(y) + im(x)) + I*cos(re(x) - im(y))*sinh(re(y) + im(x))
```

Note also that the same behavior can be obtained by using `as_real_imag()` method. However it will return a tuple containing the real part in the first place and the imaginary part in the other. This can be also done in a two step process by using `collect` function:

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

```
>>> collect((x + I*y).expand(complex=True), I, evaluate=False) == {1: re(x) - im(y),
↳ I: re(y) + im(x)}
True
```

There is also possibility for expanding expressions in terms of expressions of different kind. This is very general type of expanding and usually you would use `rewrite()` to do specific type of rewrite:

```
>>> GoldenRatio.expand(func=True)
1/2 + sqrt(5)/2
```

3.17.2 Common Subexpression Detection and Collection

Before evaluating a large expression, it is often useful to identify common subexpressions, collect them and evaluate them at once. This is implemented in the `cse` function. Examples:

```
>>> from diofant import cse, sqrt, sin, pprint
>>> from diofant.abc import x

>>> pprint(cse(sqrt(sin(x))), use_unicode=True)
(
  [ ], [  $\sqrt{\sin(x)}$  ]
)

>>> pprint(cse(sqrt(sin(x)+5)*sqrt(sin(x)+4)), use_unicode=True)
(
  [ (x_0$, sin(x)) ], [  $\sqrt{x_0$ + 4} \cdot \sqrt{x_0$ + 5}$  ]
)

>>> pprint(cse(sqrt(sin(x+1) + 5 + cos(y))*sqrt(sin(x+1) + 4 + cos(y))),
...       use_unicode=True)
(
  [ (x_0$, sin(x + 1) + cos(y)) ], [  $\sqrt{x_0$ + 4} \cdot \sqrt{x_0$ + 5}$  ]
)

>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y))), use_unicode=True)
(
  [ (x_0$, -y), (x_1$, (x + x_0$)·(x_0$ + z)) ], [  $\sqrt{x_1$ + x_1$}$  ]
)
```

Optimizations to be performed before and after common subexpressions elimination can be passed in the “optimizations” optional argument. A set of predefined basic optimizations can

be applied by passing `optimizations='basic'`:

```
>>> pprint(cse((x-y)*(z-y) + sqrt((x-y)*(z-y)), optimizations='basic'),
...         use_unicode=True)
(
  [(x$_0$, -(x - y)·(y - z))], [  $\sqrt{x$_0$ + x$_0$}$  ])
```

However, these optimizations can be very slow for large expressions. Moreover, if speed is a concern, one can pass the option `order='none'`. Order of terms will then be dependent on hashing algorithm implementation, but speed will be greatly improved.

More information:

3.18 Series Expansions

The series module implements series expansions as a function and many related functions.

`class diofant.series.limits.Limit`

Represents a 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` : {"+", "-", "real"}, optional

For `dir="+"` (default) it calculates the limit from the right ($z \rightarrow z_0 + 0$) and for `dir="-"` the limit from the left ($z \rightarrow z_0 - 0$). If `dir="real"`, the limit is the bidirectional real limit. For infinite `z0` (`oo` or `-oo`), the `dir` argument is determined from the direction of the infinity (i.e., `dir="-"` for `oo`).

Examples

```
>>> from diofant import Limit, sin
>>> from diofant.abc import x
>>> Limit(sin(x)/x, x, 0)
Limit(sin(x)/x, x, 0)
>>> Limit(1/x, x, 0, dir="-")
Limit(1/x, x, 0, dir='-')
```

`doit(**hints)`

Evaluates limit.

Notes

First we handle some trivial cases (i.e. constant), then try Gruntz algorithm (see the [gruntz](#) (page 905) module).

`diofant.series.limits.limit(expr, z, z0, dir='+')`
 Compute the directional limit of `expr` at the point `z0`.

See also:

[Limit](#) (page 902)

Examples

```
>>> from diofant import limit, sin, oo
>>> from diofant.abc import x
>>> limit(sin(x)/x, x, 0)
1
>>> limit(1/x, x, 0, dir="+")
oo
>>> limit(1/x, x, 0, dir="-")
-oo
>>> limit(1/x, x, oo)
0
```

`diofant.series.series.series(expr, x=None, x0=0, n=6, dir='+')`
 Series expansion of `expr` in `x` around point `x0`.

See also:

[diofant.core.expr.Expr.series](#) (page 79)

`diofant.series.order.O`
 alias of [Order](#) (page 903)

class `diofant.series.order.Order`
 Represents the limiting behavior of function.

The formal definition [\[R457\]](#) (page 1273) 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

args : sequence of Symbol's or pairs (Symbol, Expr), optional

If only symbols are provided, i.e. no limit point are passed, then the limit point is assumed to be zero. If no symbols are passed then all symbols in the expression are used.

References

[\[R457\]](#) (page 1273)

Examples

```
>>> from diofant import O, oo, cos, pi
>>> from diofant.abc import x, y
```

The order of a function can be intuitively thought of representing all terms of powers greater than the one specified. For example, $O(x^3)$ corresponds to any terms proportional to x^3, x^4, \dots and any higher power. For a polynomial, this leaves terms proportional to x^2, x and constants.

```
>>> 1 + x + x**2 + x**3 + x**4 + O(x**3)
1 + x + x**2 + O(x**3)
```

$O(f(x))$ is automatically transformed to $O(f(x).as_leading_term(x))$:

```
>>> O(x + x**2)
O(x)
>>> O(cos(x))
O(1)
```

Some arithmetic operations:

```
>>> O(x)*x
O(x**2)
>>> O(x) - O(x)
O(x)
```

The Big O symbol is a set, so we support membership test:

```
>>> x in O(x)
True
>>> O(1) in O(1, x)
True
>>> O(1, x) in O(1)
False
>>> O(x) in O(1, x)
True
>>> O(x**2) in O(x)
True
```

Limit points other than zero and multivariate Big O are also supported:

```
>>> O(x) == O(x, (x, 0))
True
>>> O(x + x**2, (x, oo))
O(x**2, (x, oo))
>>> O(cos(x), (x, pi/2))
O(x - pi/2, (x, pi/2))
```

```
>>> O(1 + x*y)
O(1, x, y)
>>> O(1 + x*y, (x, 0), (y, 0))
O(1, x, y)
>>> O(1 + x*y, (x, oo), (y, oo))
O(x*y, (x, oo), (y, oo))
```

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.series.residues.residue(expr, x, x0)`

Finds the residue of `expr` at the point `x=x0`.

The residue is defined [R458] (page 1273) 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 [R459] (page 1273)

References

[R458] (page 1273), [R459] (page 1273)

Examples

```
>>> from diofant import residue, sin
>>> from diofant.abc import x
>>> residue(1/x, x, 0)
1
>>> residue(1/x**2, x, 0)
0
>>> residue(2/sin(x), x, 0)
2
```

3.18.1 The Gruntz Algorithm

This section explains the basics of the algorithm [R460] (page 1273) used for computing limits. Most of the time the `limit()` (page 902) 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 [\[R460\]](#) (page 1273) for proof of the termination (pp. 52-60).

References

`diofant.series.gruntz.compare(a, b, x)`

Determine order relation between two functons.

Returns {1, 0, -1}

Respectively, if $a(x) \succ b(x)$, $a(x) \asymp b(x)$ or $b(x) \succ a(x)$.

Examples

```
>>> from diofant import Symbol, exp
```

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

```
>>> compare(x, x**2, x)
0
>>> compare(1/x, x**m, x)
0
>>> compare(exp(x), exp(x**2), x)
-1
>>> compare(exp(x), x**5, x)
1
```

`diofant.series.gruntz.limitinf(e, x)`

Compute limit of the expression at the infinity.

Examples

```
>>> from diofant import Symbol, exp, log
```

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

```
>>> limitinf(exp(x)*(exp(1/x - exp(-x)) - exp(1/x)), x)
-1
>>> limitinf(x/log(x**(log(x**(log(2)/log(x))))), x)
oo
```

`diofant.series.gruntz.mrv(e, x)`
Calculate the MRV set of expression.

Examples

```
>>> from diofant import Symbol, exp, log
```

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

```
>>> mrv(log(x - log(x))/log(x), x)
{x}
>>> mrv(exp(x + exp(-x)), x)
{E**(-x), E**(x + E**(-x))}
```

`diofant.series.gruntz.mrv_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 (c_0, e_0) of Expr.

Examples

```
>>> from diofant import Symbol, exp
```

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

```
>>> mrv_leadterm(1/exp(-x + exp(-x)) - exp(x), x)
(-1, 0)
```

`diofant.series.gruntz.mrv_max(f, g, x)`
Computes the maximum of two MRV sets.

`diofant.series.gruntz.rewrite(e, x, w)`
Rewrites expression in terms of the most rapidly varying 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 most rapidly varying in x subexpression.

Returns tuple

A pair: rewritten (in w) expression and $\log(w)$.

Examples

```
>>> from diofant import Symbol, exp
```

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

```
>>> rewrite(exp(x), x, m)
(1/m, -x)
>>> rewrite(exp(x)*log(log(exp(x))), x, m)
(log(x)/m, -x)
```

`diofant.series.gruntz.sign(e, x)`

Determine a sign of an expression at 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 $x \rightarrow \infty$.

The result of this function is currently undefined if e changes sign arbitrarily often at infinity (e.g. $\sin(x)$).

3.19 Sets

3.19.1 Set

class `diofant.sets.sets.Set`

The base class for any kind of set.

This is not meant to be used directly as a container of items. It does not behave like the builtin `set`; see [FiniteSet](#) (page 915) for that.

Real intervals are represented by the [Interval](#) (page 913) class and unions of sets by the [Union](#) (page 916) class. The empty set is represented by the [EmptySet](#) (page 918) class and available as a singleton as `S.EmptySet`.

boundary

The boundary or frontier of a set

A point x is on the boundary of a set S if

1. x is in the closure of S . I.e. Every neighborhood of x contains a point in S .
2. x is not in the interior of S . I.e. There does not exist an open set centered on x contained entirely within S .

There are the points on the outer rim of S . If S is open then these points need not actually be contained within S .

For example, the boundary of an interval is its start and end points. This is true regardless of whether or not the interval is open.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).boundary
{0, 1}
>>> Interval(0, 1, True, False).boundary
{0, 1}
```

complement(*universe*)

The complement of 'self' w.r.t the given the universe.

Examples

```
>>> from diofant import Interval, S
>>> Interval(0, 1).complement(S.Reals)
(-oo, 0) U (1, oo)
```

```
>>> Interval(0, 1).complement(S.UniversalSet)
UniversalSet() \ [0, 1]
```

contains(*other*)

Returns True if 'other' is contained in 'self' as an element.

As a shortcut it is possible to use the 'in' operator:

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).contains(0.5)
true
>>> 0.5 in Interval(0, 1)
True
```

inf

The infimum of 'self'

Examples

```
>>> from diofant import Interval, Union
>>> Interval(0, 1).inf
0
>>> Union(Interval(0, 1), Interval(2, 3)).inf
0
```

intersect(*other*)

Returns the intersection of 'self' and 'other'.

```
>>> from diofant import Interval
```

```
>>> Interval(1, 3).intersect(Interval(1, 2))
[1, 2]
```

intersection(*other*)

Alias for *intersect*() (page 909)

is_disjoint(*other*)

Returns True if 'self' and 'other' are disjoint

References

[R461] (page 1273)

Examples

```
>>> from diofant import Interval
>>> Interval(0, 2).is_disjoint(Interval(1, 2))
False
>>> Interval(0, 2).is_disjoint(Interval(3, 4))
True
```

is_proper_subset(*other*)

Returns True if 'self' is a proper subset of 'other'.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 0.5).is_proper_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_proper_subset(Interval(0, 1))
False
```

is_proper_superset(*other*)

Returns True if 'self' is a proper superset of 'other'.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).is_proper_superset(Interval(0, 0.5))
True
>>> Interval(0, 1).is_proper_superset(Interval(0, 1))
False
```

is_subset(*other*)

Returns True if 'self' is a subset of 'other'.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 0.5).is_subset(Interval(0, 1))
True
>>> Interval(0, 1).is_subset(Interval(0, 1, left_open=True))
False
```

is_superset(*other*)

Returns True if 'self' is a superset of 'other'.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 0.5).is_superset(Interval(0, 1))
False
>>> Interval(0, 1).is_superset(Interval(0, 1, left_open=True))
True
```

isdisjoint(*other*)

Alias for *is_disjoint()* (page 910)

issubset(*other*)

Alias for *is_subset()* (page 910)

issuperset(*other*)

Alias for *is_superset()* (page 911)

measure

The (Lebesgue) measure of 'self'

Examples

```
>>> from diofant import Interval, Union
>>> Interval(0, 1).measure
1
>>> Union(Interval(0, 1), Interval(2, 3)).measure
2
```

powerset()

Find the Power set of 'self'.

References

[R462] (page 1273)

Examples

```
>>> from diofant import FiniteSet, EmptySet
>>> A = EmptySet()
>>> A.powerset()
```

```
{EmptySet()}
>>> A = FiniteSet(1, 2)
>>> a, b, c = FiniteSet(1), FiniteSet(2), FiniteSet(1, 2)
>>> A.powerset() == FiniteSet(a, b, c, EmptySet())
True
```

sup

The supremum of ‘self’

Examples

```
>>> from diofant import Interval, Union
>>> Interval(0, 1).sup
1
>>> Union(Interval(0, 1), Interval(2, 3)).sup
3
```

union(*other*)

Returns the union of ‘self’ and ‘other’.

Examples

As a shortcut it is possible to use the ‘+’ operator:

```
>>> from diofant import Interval, FiniteSet
>>> Interval(0, 1).union(Interval(2, 3))
[0, 1] U [2, 3]
>>> Interval(0, 1) + Interval(2, 3)
[0, 1] U [2, 3]
>>> Interval(1, 2, True, True) + FiniteSet(2, 3)
(1, 2] U {3}
```

Similarly it is possible to use the ‘-’ operator for set differences:

```
>>> Interval(0, 2) - Interval(0, 1)
(1, 2]
>>> Interval(1, 3) - FiniteSet(2)
[1, 2) U (2, 3]
```

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.

$$f(x)|x \in self$$

See also:

[diofant.sets.fancysets.ImageSet](#) (page 921)

Examples

```
>>> from diofant import Interval, Symbol, imageset, sin, Lambda
>>> x = Symbol('x')
```

```
>>> 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(-2, 1))
ImageSet(Lambda(x, sin(x)), [-2, 1])
```

Elementary Sets

3.19.2 Interval

`class diofant.sets.sets.Interval`

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.

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

[R463] (page 1273)

Examples

```
>>> from diofant import Symbol, Interval
>>> 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', extended_real=True)
>>> Interval(0, a)
[0, a]
```

classmethod `Lopen(a, b)`

Return an interval not including the left boundary.

classmethod `Ropen(a, b)`

Return an interval not including the right boundary.

as_relational(x)

Rewrite an interval in terms of inequalities and logic operators.

end

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).end
1
```

is_left_unbounded

Return True if the left endpoint is negative infinity.

is_right_unbounded

Return True if the right endpoint is positive infinity.

left

The left end point of 'self'.

This property takes the same value as the 'inf' property.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).start
0
```

left_open

True if 'self' is left-open.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1, left_open=True).left_open
true
>>> Interval(0, 1, left_open=False).left_open
false
```

classmethod `open(a, b)`

Return an interval including neither boundary.

right

The right end point of 'self'.

This property takes the same value as the 'sup' property.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).end
1
```

right_open

True if 'self' is right-open.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1, right_open=True).right_open
true
>>> Interval(0, 1, right_open=False).right_open
false
```

start

The left end point of 'self'.

This property takes the same value as the 'inf' property.

Examples

```
>>> from diofant import Interval
>>> Interval(0, 1).start
0
```

3.19.3 FiniteSet

class diofant.sets.sets.FiniteSet

Represents a finite set of discrete numbers

References

[R464] (page 1273)

Examples

```
>>> from diofant import FiniteSet
>>> FiniteSet(1, 2, 3, 4)
{1, 2, 3, 4}
```

```
>>> 3 in FiniteSet(1, 2, 3, 4)
True
```

as_relational(*symbol*)

Rewrite a `FiniteSet` in terms of equalities and logic operators.

Compound Sets

3.19.4 Union

class `diofant.sets.sets.Union`

Represents a union of sets as a *Set* (page 908).

See also:

Intersection (page 916)

References

[R465] (page 1273)

Examples

```
>>> from diofant import Union, Interval
>>> Union(Interval(1, 2), Interval(3, 4))
[1, 2] U [3, 4]
```

The `Union` constructor will always try to merge overlapping intervals, if possible. For example:

```
>>> Union(Interval(1, 2), Interval(2, 3))
[1, 3]
```

as_relational(*symbol*)

Rewrite a `Union` in terms of equalities and logic operators.

static reduce(*args*)

Simplify a *Union* (page 916) 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

3.19.5 Intersection

class `diofant.sets.sets.Intersection`

Represents an intersection of sets as a *Set* (page 908).

See also:

Union (page 916)

References

[R466] (page 1273)

Examples

```
>>> from diofant import Intersection, Interval
>>> Intersection(Interval(1, 3), Interval(2, 4))
[2, 3]
```

We often use the `.intersect` method

```
>>> Interval(1,3).intersect(Interval(2,4))
[2, 3]
```

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

3.19.6 ProductSet

class `diofant.sets.sets.ProductSet`

Represents a Cartesian Product of Sets.

Returns a Cartesian product given several sets as either an iterable or individual arguments.

Can use ‘*’ operator on any sets for convenient shorthand.

Notes

- Passes most operations down to the argument sets
- Flattens Products of ProductSets

References

[R467] (page 1273)

Examples

```
>>> from diofant import Interval, FiniteSet, ProductSet, Symbol
>>> 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)}
True
```

3.19.7 Complement

class diofant.sets.sets.Complement

Represents relative complement of a set with another set.

$$A - B = \{x \in A | x \notin B\}$$

See also:

[Intersection](#) (page 916), [Union](#) (page 916)

References

[R468] (page 1273)

Examples

```
>>> from diofant import Complement, FiniteSet
>>> Complement(FiniteSet(0, 1, 2), FiniteSet(1))
{0, 2}
```

static reduce(*A*, *B*)
Simplify a *Complement* (page 918).

Singleton Sets

3.19.8 EmptySet

class diofant.sets.sets.EmptySet

Represents the empty set.

The empty set is available as a singleton as `S.EmptySet`.

See also:

[UniversalSet](#) (page 919)

References

[R469] (page 1273)

Examples

```
>>> from diofant import S, Interval
>>> S.EmptySet
EmptySet()
```

```
>>> Interval(1, 2).intersect(S.EmptySet)
EmptySet()
```

3.19.9 UniversalSet

class `diofant.sets.sets.UniversalSet`

Represents the set of all things.

The universal set is available as a singleton as `S.UniversalSet`

See also:

[EmptySet](#) (page 918)

References

[R470] (page 1273)

Examples

```
>>> from diofant import S, Interval
>>> S.UniversalSet
UniversalSet()
```

```
>>> Interval(1, 2).intersect(S.UniversalSet)
[1, 2]
```

Special Sets

3.19.10 Naturals

class `diofant.sets.fancysets.Naturals`

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`.

See also:

[Naturals0](#) (page 920) non-negative integers

Integers (page 920) also includes negative integers

Examples

```
>>> from diofant import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Naturals)
>>> next(iterable)
1
>>> next(iterable)
2
>>> next(iterable)
3
>>> pprint(S.Naturals.intersect(Interval(0, 10)))
{1, 2, ..., 10}
```

3.19.11 Naturals0

class diofant.sets.fancysets.Naturals0

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 919) positive integers

Integers (page 920) also includes the negative integers

3.19.12 Integers

class diofant.sets.fancysets.Integers

The set of all integers.

Represents all integers: positive, negative and zero. This set is also available as the Singleton, S.Integers.

See also:

Naturals0 (page 920) non-negative integers

Integers (page 920) positive and negative integers and zero

Examples

```
>>> from diofant import S, Interval, pprint
>>> 5 in S.Naturals
True
>>> iterable = iter(S.Integers)
>>> next(iterable)
0
>>> next(iterable)
1
```

```
>>> next(iterable)
-1
>>> next(iterable)
2
```

```
>>> pprint(S.Integers.intersect(Interval(-4, 4)))
{-4, -3, ..., 4}
```

3.19.13 Rationals

class diofant.sets.fancysets.**Rationals**

The set of all rationals.

3.19.14 ImageSet

class diofant.sets.fancysets.**ImageSet**

Image of a set under a mathematical function.

Examples

```
>>> from diofant import Symbol, S, ImageSet, FiniteSet, Lambda
```

```
>>> x = Symbol('x')
>>> N = S.Naturals
>>> squares = ImageSet(Lambda(x, x**2), N) # {x**2 for x in N}
>>> 4 in squares
True
>>> 5 in squares
False
```

```
>>> FiniteSet(0, 1, 2, 3, 4, 5, 6, 7, 9, 10).intersect(squares)
{1, 4, 9}
```

```
>>> square_iterable = iter(squares)
>>> for i in range(4):
...     next(square_iterable)
1
4
9
16
```

3.19.15 Range

class diofant.sets.fancysets.**Range**

Represents a range of integers.

Examples

```
>>> from diofant import Range
>>> list(Range(5)) # 0 to 5
[0, 1, 2, 3, 4]
>>> list(Range(10, 15)) # 10 to 15
[10, 11, 12, 13, 14]
>>> list(Range(10, 20, 2)) # 10 to 20 in steps of 2
[10, 12, 14, 16, 18]
>>> list(Range(20, 10, -2)) # 20 to 10 backward in steps of 2
[20, 18, 16, 14, 12]
```

3.20 Simplify

3.20.1 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:

```
>>> from diofant import simplify, cos, sin
>>> from diofant.abc import x, y
>>> a = (x + x**2)/(x*sin(y)**2 + x*cos(y)**2)
>>> a
(x**2 + x)/(x*sin(y)**2 + x*cos(y)**2)
>>> simplify(a)
x + 1
```

Note that we could have obtained the same result by using specific simplification functions:

```
>>> from diofant import trigsimp, cancel
>>> trigsimp(a)
(x**2 + x)/x
>>> cancel(_)
x + 1
```

In some cases, applying `simplify()` (page 922) 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 142), which returns the total number of operations in the expression.

For example, if `ratio=1`, `simplify` output can't be longer than input.

```
>>> from diofant import sqrt, simplify, count_ops, oo
>>> root = 1/(sqrt(2)+3)
```

Since `simplify(root)` would result in a slightly longer expression, `root` is returned unchanged instead:

```
>>> simplify(root, ratio=1) == root
True
```

If `ratio=oo`, `simplify` will be applied anyway:

```
>>> count_ops(simplify(root, ratio=oo)) > count_ops(root)
True
```

Note that the shortest expression is not 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:

```
>>> from diofant import symbols, log
>>> a, b = symbols('a b', positive=True)
>>> g = log(a) + log(b) + log(a)*log(1/b)
>>> h = simplify(g)
>>> h
log(a*b**(-log(a) + 1))
>>> count_ops(g)
8
>>> count_ops(h)
5
```

So you can see that `h` is simpler than `g` using the `count_ops` metric. However, we may not like how `simplify` (in this case, using `logcombine`) has created the `b**(-log(1/a) + 1)` term. A simple way to reduce this would be to give more weight to powers as operations in `count_ops`. We can do this by using the `visual=True` option:

```
>>> print(count_ops(g, visual=True))
2*ADD + DIV + 4*LOG + MUL
>>> print(count_ops(h, visual=True))
2*LOG + MUL + POW + SUB
```

```
>>> from diofant import Symbol, S
>>> def my_measure(expr):
...     POW = Symbol('POW')
...     # Discourage powers by giving POW a weight of 10
...     count = count_ops(expr, visual=True).subs(POW, 10)
...     # Every other operation gets a weight of 1 (the default)
```

```

...     count = count.replace(Symbol, type(S.One))
...     return count
>>> my_measure(g)
8
>>> my_measure(h)
14
>>> 15./8 > 1.7 # 1.7 is the default ratio
True
>>> simplify(g, measure=my_measure)
-log(a)*log(b) + log(a) + log(b)

```

Note that because `simplify()` internally tries many different simplification strategies and then compares them using the measure function, we get a completely different result that is still different from the input expression by doing this.

3.20.2 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 x, y, z, alpha
>>> from diofant import separatevars, sin
>>> separatevars((x*y)**y)
(x*y)**y
>>> separatevars((x*y)**y, force=True)
x**y*y**y

```

```

>>> e = 2*x**2*z*sin(y)+2*z*x**2
>>> separatevars(e)
2*x**2*z*(sin(y) + 1)

```



```
>>> separatevars(e, symbols=(x, y), dict=True) == {'coeff': 2*z, x: x**2, y:
↪ sin(y) + 1}
True
>>> separatevars(e, [x, y, alpha], dict=True) == {'coeff': 2*z, alpha: 1, x: x**2,
↪ y: sin(y) + 1}
True
```

If the expression is not really separable, or is only partially separable, `separatevars` will do the best it can to separate it by using factoring.

```
>>> separatevars(x + x*y - 3*x**2)
-x*(3*x - y - 1)
```

If the expression is not separable then `expr` is returned unchanged or (if `dict=True`) then `None` is returned.

```
>>> eq = 2*x + y*sin(x)
>>> separatevars(eq) == eq
True
>>> separatevars(2*x + y*sin(x), symbols=(x, y), dict=True) == None
True
```

3.20.3 nthroot

`diofant.simplify.simplify.nthroot(expr, n, max_len=4, prec=15)`
compute a real *n*th-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

```
>>> from diofant.simplify.simplify import nthroot
>>> from diofant import sqrt
>>> nthroot(90 + 34*sqrt(7), 3)
sqrt(7) + 3
```

3.20.4 besssimp

`diofant.simplify.simplify.besssimp(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 import besselj, besseli, besselsimp, polar_lift, I, Rational
>>> from diofant.abc import z, 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
```

3.20.5 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.

The algorithm performs three basic steps:

1. Rewrite all functions in terms of gamma, if possible.
2. Rewrite all occurrences of gamma in terms of products of gamma and rising factorial with integer, absolute constant exponent.
3. Perform simplification of nested fractions, powers and if the resulting expression is a quotient of polynomials, reduce their total degree.

If $f(k)$ is hypergeometric then as result we arrive with a quotient of polynomials of minimal degree. Otherwise None is returned.

For more information on the implemented algorithm refer to:

- 1.W. Koepf, Algorithms for m-fold Hypergeometric Summation, Journal of Symbolic Computation (1995) 20, 399-417

3.20.6 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.

For more information see `hypersimp()`.

3.20.7 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).

See also:

[diofant.core.function.nfloat](#) (page 146)

Examples

```
>>> from diofant import nsimplify, sqrt, GoldenRatio, exp, I, pi
>>> nsimplify(4/(1+sqrt(5)), [GoldenRatio])
-2 + 2*GoldenRatio
>>> nsimplify((1/(exp(3*pi*I/5)+1)))
1/2 - I*sqrt(sqrt(5)/10 + 1/4)
>>> nsimplify(I**I, [pi])
E**(-pi/2)
>>> nsimplify(pi, tolerance=0.01)
22/7
```

3.20.8 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.

```
>>> from diofant import posify, Symbol, log, solve
>>> from diofant.abc import x
>>> 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)
[-2, 2]
>>> eq_x, reps = posify([eq, x]); eq_x
[_x**2 - 4, _x]
>>> solve(*eq_x)
[2]
```

3.20.9 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.

See also:

[posify \(page 927\)](#) replace all symbols with symbols having positive assumptions

Examples

```
>>> from diofant import Symbol, symbols, log, logcombine, I
>>> from diofant.abc import a, x, y, z
>>> logcombine(a*log(x) + log(y) - log(z))
a*log(x) + log(y) - log(z)
>>> logcombine(a*log(x) + log(y) - log(z), force=True)
log(x**a*y/z)
>>> x,y,z = symbols('x,y,z', positive=True)
>>> a = Symbol('a', extended_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)
```

3.20.10 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

```
>>> from diofant import radsimp, sqrt, Symbol, denom, pprint, I
>>> from diofant import factor_terms, fraction, signsimp
>>> from diofant.simplify.radsimp import collect_sqrt
>>> from diofant.abc import a, b, c
```

```
>>> radsimp(1/(I + 1))
(1 - I)/2
>>> radsimp(1/(2 + sqrt(2)))
(-sqrt(2) + 2)/2
>>> x,y = map(Symbol, 'xy')
>>> e = ((2 + 2*sqrt(2))*x + (2 + sqrt(8))*y)/(2 + sqrt(2))
>>> radsimp(e)
sqrt(2)*(x + y)
```

No simplification beyond removal of the gcd is done. One might want to polish the result a little, however, by collecting square root terms:

```
>>> r2 = sqrt(2)
>>> r5 = sqrt(5)
>>> ans = radsimp(1/(y*r2 + x*r2 + a*r5 + b*r5))
>>> pprint(ans, use_unicode=False)
```

$$\frac{\sqrt{5} * a + \sqrt{5} * b - \sqrt{2} * x - \sqrt{2} * y}{5 * a^2 + 10 * a * b + 5 * b^2 - 2 * x^2 - 4 * x * y - 2 * y^2}$$

```
>>> n, d = fraction(ans)
>>> pprint(factor_terms(signsimp(collect_sqrt(n))/d, radical=True), use_
↳ unicode=False)
```

$$\frac{\sqrt{5} * (a + b) - \sqrt{2} * (x + y)}{5 * a^2 + 10 * a * b + 5 * b^2 - 2 * x^2 - 4 * x * y - 2 * y^2}$$

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

```
>>> from diofant import sqrt
>>> from diofant.simplify.radsimp import rad_rationalize
>>> 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 930), so user is expected to provide an expression in an appropriate form (for example, by using `expand()` (page 137) prior to calling this function). This makes `collect()` (page 930) 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 145) 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.

See also:

[collect_const](#) (page 934), [collect_sqrt](#) (page 933), [rcollect](#) (page 933)

Examples

```
>>> from diofant import collect, expand, factor
>>> from diofant.abc import a, b, c, x, y
```

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:

```
>>> from diofant import sin, log
>>> collect(a*sin(2*x) + b*sin(2*x), sin(2*x))
(a + b)*sin(2*x)
```

```
>>> collect(a*x*log(x) + b*(x*log(x)), x*log(x))
x*(a + b)*log(x)
```

You can use wildcards in the pattern:

```
>>> from diofant import Wild
>>> 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 930) function apply to the exponential function, so you can get:

```
>>> from diofant import exp
>>> 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:

```
>>> from diofant import Derivative as D, Function
>>> f = Function('f')(x)
```

```
>>> collect(a*D(f,x) + b*D(f,x), D(f,x))
(a + b)*Derivative(f(x), x)
```

```
>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), f)
(a + b)*Derivative(f(x), x, x)
```

```
>>> collect(a*D(D(f,x),x) + b*D(D(f,x),x), D(f,x), exact=True)
a*Derivative(f(x), x, x) + b*Derivative(f(x), x, x)
```

```
>>> collect(a*D(f,x) + b*D(f,x) + a*f + b*f, f)
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*D(D(f,x),x)**2 + b*D(D(f,x),x)**2, D(f,x))
(a + b)*Derivative(f(x), x, x)**2
```

Finally, you can apply a function to each of the collected coefficients. For example you can factorize symbolic coefficients of polynomial:


```
>>> f = expand((x + a + 1)**3)
```

```
>>> collect(f, x, factor)
x**3 + 3*x**2*(a + 1) + 3*x*(a + 1)**2 + (a + 1)**3
```

`diofant.simplify.radsimp.rcollect(expr, *vars)`

Recursively collect sums in an expression.

See also:

[collect](#) (page 930), [collect_const](#) (page 934), [collect_sqrt](#) (page 933)

Examples

```
>>> from diofant.simplify import rcollect
>>> from diofant.abc import x, y
```

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

```
>>> rcollect(expr, y)
(x + y*(x**2 + x + 1))/(x + y)
```

collect_sqrt

`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.

See also:

[collect](#) (page 930), [collect_const](#) (page 934), [rcollect](#) (page 933)

Examples

```
>>> from diofant import sqrt
>>> from diofant.simplify.radsimp import collect_sqrt
>>> from diofant.abc import a, b
```

```
>>> r2, r3, r5 = [sqrt(i) for i in [2, 3, 5]]
>>> collect_sqrt(a*r2 + b*r2)
sqrt(2)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r3)
sqrt(2)*(a + b) + sqrt(3)*(a + b)
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5)
sqrt(3)*a + sqrt(5)*b + sqrt(2)*(a + b)
```

If evaluate is False then the arguments will be sorted and returned as a list and a count of the number of sqrt-containing terms will be returned:

```
>>> collect_sqrt(a*r2 + b*r2 + a*r3 + b*r5, evaluate=False)
((sqrt(3)*a, sqrt(5)*b, sqrt(2)*(a + b)), 3)
>>> collect_sqrt(a*sqrt(2) + b, evaluate=False)
((b, sqrt(2)*a), 1)
>>> collect_sqrt(a + b, evaluate=False)
((a + b,), 0)
```

collect_const

`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.

See also:

[collect](#) (page 930), [collect_sqrt](#) (page 933), [rcollect](#) (page 933)

Examples

```
>>> from diofant import sqrt
>>> from diofant.abc import a, s, x, y, z
>>> from diofant.simplify.radsimp import collect_const
>>> collect_const(sqrt(3) + sqrt(3)*(1 + sqrt(2)))
sqrt(3)*(sqrt(2) + 2)
>>> collect_const(sqrt(3)*s + sqrt(7)*s + sqrt(3) + sqrt(7))
(sqrt(3) + sqrt(7))*(s + 1)
>>> s = sqrt(2) + 2
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7))
(sqrt(2) + 3)*(sqrt(3) + sqrt(7))
>>> collect_const(sqrt(3)*s + sqrt(3) + sqrt(7)*s + sqrt(7), sqrt(3))
sqrt(7) + sqrt(3)*(sqrt(2) + 3) + sqrt(7)*(sqrt(2) + 2)
```

The collection is sign-sensitive, giving higher precedence to the unsigned values:

```
>>> collect_const(x - y - z)
x - (y + z)
>>> collect_const(-y - z)
-(y + z)
>>> collect_const(2*x - 2*y - 2*z, 2)
2*(x - y - z)
>>> collect_const(2*x - 2*y - 2*z, -2)
2*x - 2*(y + z)
```

fraction

`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 `numer(expr)` or `denom(expr)` functions respectively.

```
>>> from diofant import fraction, Rational, Symbol
>>> from diofant.abc import x, y
```

```
>>> fraction(x/y)
(x, y)
>>> fraction(x)
(x, 1)
```

```
>>> fraction(1/y**2)
(1, y**2)
```

```
>>> fraction(x*y/2)
(x*y, 2)
>>> fraction(Rational(1, 2))
(1, 2)
```

This function will also work fine with assumptions:

```
>>> k = Symbol('k', negative=True)
>>> fraction(x * y**k)
(x, y**(-k))
```

If we know nothing about sign of some exponent and 'exact' flag is unset, then structure this exponent's structure will be analyzed and pretty fraction will be returned:

```
>>> from diofant import exp
>>> fraction(2*x**(-y))
(2, x**y)
```

```
>>> fraction(exp(-x))
(1, E**x)
```

```
>>> fraction(exp(-x), exact=True)
(E**(-x), 1)
```

3.20.11 Ratsimp

ratsimp

`diofant.simplify.ratsimp.ratsimp(expr)`

Put an expression over a common denominator, cancel and reduce.

Examples

```
>>> from diofant import ratsimp
>>> from diofant.abc import x, y
```

```
>>> ratsimp(1/x + 1/y)
(x + y)/(x*y)
```

3.20.12 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. (see the *fu* docstring).

Examples

```
>>> from diofant import trigsimp, sin, cos, log
>>> from diofant.abc import x, y
>>> e = 2*sin(x)**2 + 2*cos(x)**2
>>> trigsimp(e)
2
```

Simplification occurs wherever trigonometric functions are located.

```
>>> trigsimp(log(e))
log(2)
```

Using `method = "groebner"` (or `"combined"`) might lead to greater simplification.

The old `trigsimp` routine can be accessed as with method 'old'.

```
>>> from diofant import coth, tanh
>>> t = 3*tanh(x)**7 - 2/coth(x)**7
>>> trigsimp(t, method='old') == t
True
>>> trigsimp(t)
tanh(x)**7
```

3.20.13 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:

$x^a * y^a \Rightarrow (x*y)^a$ <p style="text-align: center;">as well as things like $2^{2x} \Rightarrow 4^x$</p>

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

```
>>> from diofant import powsimp, exp, log, symbols
>>> from diofant.abc import x, y, z, n
>>> powsimp(x**y*x**z*y**z, combine='all')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='exp')
x**(y + z)*y**z
>>> powsimp(x**y*x**z*y**z, combine='base', force=True)
x**y*(x*y)**z
```

```
>>> powsimp(x**z*x**y*n**z*n**y, combine='all', force=True)
(n*x)**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='exp')
n**(y + z)*x**(y + z)
>>> powsimp(x**z*x**y*n**z*n**y, combine='base', force=True)
(n*x)**y*(n*x)**z
```

```
>>> x, y = symbols('x y', positive=True)
>>> powsimp(log(exp(x)*exp(y)))
log(E**x*E**y)
>>> powsimp(log(exp(x)*exp(y)), deep=True)
x + y
```

Radicals with Mul bases will be combined if combine='exp'

```
>>> from diofant import sqrt, Mul
>>> x, y = symbols('x y')
```

Two radicals are automatically joined through Mul:

```
>>> a=sqrt(x*sqrt(y))
>>> a*a**3 == a**4
True
```

But if an integer power of that radical has been autoexpanded then Mul does not join the resulting factors:

```
>>> a**4 # auto expands to a Mul, no longer a Pow
x**2*y
>>> _*a # so Mul doesn't combine them
x**2*y*sqrt(x*sqrt(y))
>>> powsimp(_) # but powsimp will
(x*sqrt(y))**(5/2)
>>> powsimp(x*y*a) # but won't when doing so would violate assumptions
x*y*sqrt(x*sqrt(y))
```

powdenest

`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, $(bb1**be1 * bb2**be2...)**e$, 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))) -> a**3*b**6`.

Examples

```
>>> from diofant.abc import a, b, x, y, z
>>> from diofant import Symbol, exp, log, sqrt, symbols, powdenest
```

```
>>> powdenest((x**(2*a/3))**(3*x))
(x**(2*a/3))**(3*x)
>>> powdenest(exp(3*x*log(2)))
2**(3*x)
```

Assumptions may prevent expansion:

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

```
>>> p = symbols('p', positive=True)
>>> powdenest(sqrt(p**2))
p
```

No other expansion is done.

```
>>> i, j = symbols('i,j', integer=True)
>>> powdenest((x**x)**(i + j)) # -X-> (x**x)**i*(x**x)**j
x**(x*(i + j))
```

But `exp()` will be denested by moving all non-log terms outside of the function; this may result in the collapsing of the `exp` to a power with a different base:

```
>>> powdenest(exp(3*y*log(x)))
x**(3*y)
>>> powdenest(exp(y*(log(a) + log(b))))
(a*b)**y
>>> powdenest(exp(3*(log(a) + log(b))))
a**3*b**3
```

If assumptions allow, symbols can also be moved to the outermost exponent:

```
>>> i = Symbol('i', integer=True)
>>> powdenest(((x**(2*i))**(3*y))**x)
((x**(2*i))**(3*y))**x
>>> powdenest(((x**(2*i))**(3*y))**x, force=True)
x**(6*i*x*y)
```

```
>>> powdenest(((x**(2*a/3))**(3*y/i))**x)
((x**(2*a/3))**(3*y/i))**x
>>> powdenest((x**(2*i)*y**(4*i))**z, force=True)
(x*y**2)**(2*i*z)
```

```
>>> n = Symbol('n', negative=True)
```

```
>>> powdenest((x**i)**y, force=True)
x**(i*y)
>>> powdenest((n**i)**x, force=True)
(n**i)**x
```

3.20.14 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 $\text{gamma}(x)*\text{gamma}(1-x) == \text{pi}/\text{sin}(\text{pi}*x)$.
- 2.Reduce the number of gammas by applying the multiplication theorem $\text{gamma}(x)*\text{gamma}(x+1/n)*\dots*\text{gamma}(x+(n-1)/n) == C*\text{gamma}(n*x)$.
- 3.Reduce the number of prefactors by absorbing them into gammas, where possible.

All transformation rules can be found (or was derived from) here:

- 1.<http://functions.wolfram.com/GammaBetaErf/Pochhammer/17/01/02/>
- 2.<http://functions.wolfram.com/GammaBetaErf/Pochhammer/27/01/0005/>

Examples

```
>>> from diofant.simplify import combsimp
>>> from diofant import factorial, binomial
>>> from diofant.abc import n, k
```

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

3.20.15 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].

See also:

[diofant.solvers.solvers.unrad](#) (page 1066)

References

[1] <http://researcher.watson.ibm.com/researcher/files/us-fagin/symb85.pdf>

[2] D. J. Jeffrey and A. D. Rich, 'Simplifying Square Roots of Square Roots by Denesting' (available at <http://www.cybertester.com/data/denest.pdf>)

Examples

```
>>> from diofant.simplify.sqrtdenest import sqrtdenest
>>> from diofant import sqrt
>>> sqrtdenest(sqrt(5 + 2 * sqrt(6)))
sqrt(2) + sqrt(3)
```

3.20.16 Common Subexpression Elimination**cse**

`diofant.simplify.cse_main.cse(exprs, symbols=None, optimizations=None, postprocess=None, order='canonical')`

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'`.

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 import cse, SparseMatrix
>>> from diofant.abc import x, y, z, w
>>> cse(((w + x + y + z)*(w + y + z))/(w + x)**3)
([(x0, y + z), (x1, w + x)], [(w + x0)*(x0 + x1)/x1**3])
```

Note that currently, $y + z$ will not get substituted if $-y - z$ is used.

```
>>> cse(((w + x + y + z)*(w - y - z))/(w + x)**3)
([(x0, w + x)], [(w - y - z)*(x0 + y + z)/x0**3])
```

List of expressions with recursive substitutions:

```
>>> m = SparseMatrix([x + y, x + y + z])
>>> cse([(x+y)**2, x + y + z, y + z, x + z + y, m])
([(x0, x + y), (x1, x0 + z)], [x0**2, x1, y + z, x1, Matrix([
[x0],
[x1]])])
```

Note: the type and mutability of input matrices is retained.

```
>>> isinstance(_[1][-1], SparseMatrix)
True
```

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

```
>>> from diofant.simplify.cse_main import opt_cse
>>> from diofant.abc import x
>>> 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=None, order='canonical')`
 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'`.

3.20.17 Hypergeometric Function Expansion

hyperexpand

`diofant.simplify.hyperexpand.hyperexpand(f, allow_hyper=False, rewrite='default')`

Expand hypergeometric functions. If `allow_hyper` is True, allow partial simplification (that is a result different from input, but still containing hypergeometric functions).

Examples

```
>>> from diofant.simplify.hyperexpand import hyperexpand
>>> from diofant.functions import hyper
>>> from diofant.abc import z
>>> 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
```

3.20.18 Traversal Tools

use

`diofant.simplify.traversaltools.use(expr, func, level=0, args=(), kwargs={})`
 Use `func` to transform `expr` at the given level.

Examples

```
>>> from diofant import use, expand
>>> from diofant.abc import x, y
```

```
>>> f = (x + y)**2*x + 1
```

```
>>> use(f, expand, level=2)
x*(x**2 + 2*x*y + y**2) + 1
>>> expand(f)
x**3 + 2*x**2*y + x*y**2 + 1
```

3.20.19 EPath Tools

EPath class

class `diofant.simplify.epathtools.EPath`

Manipulate expressions using paths.

EPath grammar in EBNF notation:

```
literal ::= /[A-Za-z_][A-Za-z_0-9]*/
number ::= /-?\d+/
type ::= literal
attribute ::= literal "?"
all ::= "*"
slice ::= "[" number? (":" number? (":" number?))?" "]"
range ::= all | slice
query ::= (type | attribute) ("|" (type | attribute))*
selector ::= range | query range?
path ::= "/" selector ("/" selector)*
```

See the docstring of the `epath()` function.

apply(`expr, func, args=None, kwargs=None`)

Modify parts of an expression selected by a path.

Examples

```
>>> from diofant.simplify.epathtools import EPath
>>> from diofant import sin, cos, E
>>> from diofant.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.apply(expr, lambda expr: expr**2)
[((x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.apply(expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

`select(expr)`

Retrieve parts of an expression selected by a path.

Examples

```
>>> from diofant.simplify.epathtools import EPath
>>> from diofant import sin, cos, E
>>> from diofant.abc import x, y, z, t
```

```
>>> path = EPath("/*/[0]/Symbol")
>>> expr = [((x, 1), 2), ((3, y), z)]
```

```
>>> path.select(expr)
[x, y]
```

```
>>> path = EPath("/*/*/Symbol")
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> path.select(expr)
[x, x, y]
```

epath

`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 945) retrieves elements selected by the path. Otherwise it applies `func` to each matching element.

Note that it is more efficient to create an `EPath` object and use the `select` and `apply` methods of that object, since this will compile the path string only once. This function should only be used as a convenient shortcut for interactive use.

This is the supported syntax:

- **select all:** `/*` Equivalent of `for arg in args:`

- select slice:** `/[0]` or `/[1:5]` or `/[1:5:2]` Supports standard Python's slice syntax.
- select by type:** `/list` or `/list|tuple` Emulates `isinstance`.
- select by attribute:** `/__iter__?` Emulates `hasattr`.

Parameters `path` : str | EPath

A path as a string or a compiled EPath.

expr : Basic | iterable

An expression or a container of expressions.

func : callable (optional)

A callable that will be applied to matching parts.

args : tuple (optional)

Additional positional arguments to `func`.

kwargs : dict (optional)

Additional keyword arguments to `func`.

Examples

```
>>> from diofant.simplify.epathtools import epath
>>> from diofant import sin, cos, E
>>> from diofant.abc import x, y, z, t
```

```
>>> path = "/*/[0]/Symbol"
>>> expr = [(x, 1), 2), ((3, y), z)]
```

```
>>> epath(path, expr)
[x, y]
>>> epath(path, expr, lambda expr: expr**2)
[(x**2, 1), 2), ((3, y**2), z)]
```

```
>>> path = "/*/*/Symbol"
>>> expr = t + sin(x + 1) + cos(x + y + E)
```

```
>>> epath(path, expr)
[x, x, y]
>>> epath(path, expr, lambda expr: 2*expr)
t + sin(2*x + 1) + cos(2*x + 2*y + E)
```

3.21 Details on the Hypergeometric Function Expansion

This page describes how the function `hyperexpand()` (page 943) and related code work. For usage, see the documentation of the `simplify` module.

3.21.1 Hypergeometric Function Expansion Algorithm

This section describes the algorithm used to expand hypergeometric functions. Most of it is based on the papers [Roach1996] (page 1273) and [Roach1997] (page 1273).

Recall that the hypergeometric function is (initially) defined as

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n z^n}{(b_1)_n \dots (b_q)_n n!}.$$

It turns out that there are certain differential operators that can change the a_p and b_q parameters by integers. If a sequence of such operators is known that converts the set of indices a_r^0 and b_s^0 into a_p and b_q , then we shall say the pair a_p, b_q is reachable from a_r^0, b_s^0 . Our general strategy is thus as follows: given a set a_p, b_q of parameters, try to look up an origin a_r^0, b_s^0 for which we know an expression, and then apply the sequence of differential operators to the known expression to find an expression for the Hypergeometric function we are interested in.

Notation

In the following, the symbol a will always denote a numerator parameter and the symbol b will always denote a denominator parameter. The subscripts p, q, r, s denote vectors of that length, so e.g. a_p denotes a vector of p numerator parameters. The subscripts i and j denote "running indices", so they should usually be used in conjunction with a "for all i ". E.g. $a_i < 4$ for all i . Uppercase subscripts I and J denote a chosen, fixed index. So for example $a_I > 0$ is true if the inequality holds for the one index I we are currently interested in.

Incrementing and decrementing indices

Suppose $a_i \neq 0$. Set $A(a_i) = \frac{z}{a_i} \frac{d}{dz} + 1$. It is then easy to show that $A(a_i) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p + e_i \\ b_q \end{matrix} \middle| z \right)$, where e_i is the i -th unit vector. Similarly for $b_j \neq 1$ we set $B(b_j) = \frac{z}{b_j - 1} \frac{d}{dz} + 1$ and find $B(b_j) {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p \\ b_q - e_j \end{matrix} \middle| z \right)$. Thus we can increment upper and decrement lower indices at will, as long as we don't go through zero. The $A(a_i)$ and $B(b_j)$ are called shift operators.

It is also easy to show that $\frac{d}{dz} {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = \frac{a_1 \dots a_p}{b_1 \dots b_q} {}_pF_q \left(\begin{matrix} a_p + 1 \\ b_q + 1 \end{matrix} \middle| z \right)$, where $a_p + 1$ is the vector $a_1 + 1, a_2 + 1, \dots$ and similarly for $b_q + 1$. Combining this with the shift operators, we arrive at one form of the Hypergeometric differential equation: $\left[\frac{d}{dz} \prod_{j=1}^q B(b_j) - \frac{a_1 \dots a_p}{(b_1 - 1) \dots (b_q - 1)} \prod_{i=1}^p A(a_i) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. This holds if all shift operators are defined, i.e. if no $a_i = 0$ and no $b_j = 1$. Clearing denominators and multiplying through by z we arrive at the following equation: $\left[z \frac{d}{dz} \prod_{j=1}^q \left(z \frac{d}{dz} + b_j - 1 \right) - z \prod_{i=1}^p \left(z \frac{d}{dz} + a_i \right) \right] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$. Even though our derivation does not show it, it can be checked that this equation holds whenever the ${}_pF_q$ is defined.

Notice that, under suitable conditions on a_I, b_J , each of the operators $A(a_i)$, $B(b_j)$ and $z \frac{d}{dz}$ can be expressed in terms of $A(a_I)$ or $B(b_J)$. Our next aim is to write the Hypergeometric differential equation as follows: $[XA(a_I) - r] {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right) = 0$, for some operator X and some constant r to be determined. If $r \neq 0$, then we can write this as $\frac{-1}{r} X {}_pF_q \left(\begin{matrix} a_p + e_I \\ b_q \end{matrix} \middle| z \right) = {}_pF_q \left(\begin{matrix} a_p \\ b_q \end{matrix} \middle| z \right)$, and so $\frac{-1}{r} X$ undoes the shifting of $A(a_I)$, whence it will be called an inverse-shift operator.

Now $A(a_I)$ exists if $a_I \neq 0$, and then $z \frac{d}{dz} = a_I A(a_I) - a_I$. 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 $z \frac{d}{dz} \prod_{j=1}^q (z \frac{d}{dz} + b_j - 1) = (a_I A(a_I) - a_I) \prod_{j=1}^q (a_I A(a_I) - a_I + b_j - 1)$. 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 $B(a_J)$, we arrive at the following rules:

- An index a_I can be decremented if $a_I \neq 1$ and $a_I \neq b_j$ for all b_j .
- An index b_j can be incremented if $b_j \neq -1$ and $b_j \neq a_i$ for all a_i .

Combined with the conditions (stated above) for the existence of shift operators, we have thus established the rules of the game!

Reduction of Order

Notice that, quite trivially, if $a_I = b_J$, we have ${}_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = {}_{p-1} F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$, 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 ${}_p F_q \left(\begin{smallmatrix} a_p \\ b_q \end{smallmatrix} \middle| z \right) = \left(p_0 + p_1 z \frac{d}{dz} + p_2 \left(z \frac{d}{dz} \right)^2 + \dots \right) {}_{p-1} F_{q-1} \left(\begin{smallmatrix} a_p^* \\ b_q^* \end{smallmatrix} \middle| z \right)$.

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:

- ${}_rF_s \left(\begin{matrix} a_r^0 \\ b_s^0 \end{matrix} \middle| z \right) = CB$
- $z \frac{d}{dz} B = MB$.

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.

3.21.2 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

$$R(z) = \sum_{r=0}^{l_u-1} res_{s=r+b}.$$

We finally need to investigate the other poles. Set $r = l_u + t, t \geq 0$. A computation shows

$$\frac{\Gamma(k_i - l_u - t)}{\Gamma(l_i - l_u - t)} = \frac{1}{(k_i - l_u - t)_{l_i - k_i}} = \frac{(-1)^{\delta_i}}{(l_u - l_i + 1)_{\delta_i}} \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t},$$

where $\delta_i = l_i - k_i$.

Also

$$\Gamma(b_j - l_u - b - t) = \frac{\Gamma(b_j - l_u - b)}{(-1)^t (l_u + b + 1 - b_j)_t},$$

$$\Gamma(1 - a_j + l_u + b + t) = \Gamma(1 - a_j + l_u + b) (1 - a_j + l_u + b)_t$$

and

$$res_{s=b+l_u+t} \Gamma(b - s) = - \frac{(-1)^{l_u+t}}{(l_u + t)!} = - \frac{(-1)^{l_u}}{l_u!} \frac{(-1)^t}{(l_u + 1)_t}.$$

Hence

$$res_{s=b+l_u+t} = -z^{b+l_u} \frac{(-1)^{l_u}}{l_u!} \prod_{i=1}^u \frac{(-1)^{\delta_i}}{(l_u - k_i + 1)_{\delta_i}} \frac{\prod_{j=1}^n \Gamma(1 - a_j + l_u + b) \prod_{j=1}^m \Gamma(b_j - l_u - b)^*}{\prod_{j=n+1}^p \Gamma(a_j - l_u - b)^* \prod_{j=m+1}^q \Gamma(1 - b_j + l_u + b)}$$

$$\times z^t \frac{(-1)^t}{(l_u + 1)_t} \prod_{i=1}^u \frac{(l_u - l_i + 1)_t}{(l_u - k_i + 1)_t} \frac{\prod_{j=1}^n (1 - a_j + l_u + b)_t \prod_{j=n+1}^p (-1)^t (l_u + b + 1 - a_j)_t^*}{\prod_{j=1}^m (-1)^t (l_u + b + 1 - b_j)_t^* \prod_{j=m+1}^q (1 - b_j + l_u + b)_t}$$

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.)

3.21.3 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 956). 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:

$$C(z) = z \cdot {}_1F_2 \left(\begin{matrix} \frac{1}{4} \\ \frac{1}{2}, \frac{5}{4} \end{matrix} \middle| -\frac{\pi^2 z^4}{16} \right).$$

First we try to add this formula to the lookup table by using the (simpler) function `add(ap,bq,res)`. The first two arguments are simply the lists containing the parameter sets

of ${}_1F_2$. The `res` argument is a little bit more complicated. We only know $C(z)$ in terms of ${}_1F_2(\dots|f(z))$ 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 330) inside any branched function by `exp_polar()` (page 330) and what we get is right for *all* w .) Hence we can write the formula as

$$C(g(w)) = g(w) \cdot {}_1F_2\left(\frac{1}{4} \middle| \frac{5}{4} w\right).$$

and trivially

$${}_1F_2\left(\frac{1}{4} \middle| \frac{5}{4} w\right) = \frac{C(g(w))}{g(w)} = \frac{C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}\right)}{\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) w^{\frac{1}{4}}}$$

which is exactly what is needed for the third parameter, `res`, in `add`. Finally, the whole function call to add this rule to the table looks like:

```
add([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 usefull pieces. In our case we find that $n = \max(1, 2 + 1) = 3$. For computing the derivatives, we have to use the operator $z \frac{d}{dz}$. 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!

```
>>> from diofant import Symbol, sqrt, exp, I, pi, fresnelc, root, diff, expand
>>> z = Symbol("z")
>>> B0 = sqrt(pi)*exp(-I*pi/4)*fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/\
...     (2*root(z,4))
>>> z * diff(B0, z)
z*(cosh(2*sqrt(z))/(4*z) - E**(-I*pi/4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*z**(1/4)/
↳sqrt(pi))/(8*z**(5/4)))
>>> expand(_)
cosh(2*sqrt(z))/4 - E**(-I*pi/4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*z**(1/4)/sqrt(pi))/
↳(8*z**(1/4))
```

Formatting this result nicely we obtain

$$B'_1 = -\frac{1}{4} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} + \frac{1}{4} \cosh(2\sqrt{z})$$

Computing the second derivative we find

```
>>> from diofant import (Symbol, cosh, sqrt, pi, exp, I, fresnelc, root,
...                       diff, expand)
>>> z = Symbol("z")
>>> B1prime = cosh(2*sqrt(z))/4 - sqrt(pi)*exp(-I*pi/4)*\
...           fresnelc(2*root(z,4)*exp(I*pi/4)/sqrt(pi))/(8*root(z,4))
>>> z * diff(B1prime, z)
z*(-cosh(2*sqrt(z))/(16*z) + sinh(2*sqrt(z))/(4*sqrt(z)) + E**(-I*pi/
↳4)*sqrt(pi)*fresnelc(2*E**(I*pi/4)*z**(1/4)/sqrt(pi))/(32*z**(5/4)))
>>> 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)*z**(1/4)/sqrt(pi))/(32*z**(1/4))
```

which can be printed as

$$B'_2 = \frac{1}{16} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} - \frac{1}{16} \cosh(2\sqrt{z}) + \frac{1}{4} \sinh(2\sqrt{z})\sqrt{z}$$

We see the common pattern and can collect the pieces. Hence it makes sense to choose B_1 and B_2 as follows

$$B = \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{\pi} \exp\left(-\frac{i\pi}{4}\right) C\left(\frac{2}{\sqrt{\pi}} \exp\left(\frac{i\pi}{4}\right) z^{\frac{1}{4}}\right)}{2z^{\frac{1}{4}}} \\ \cosh(2\sqrt{z}) \\ \sinh(2\sqrt{z})\sqrt{z} \end{pmatrix}$$

(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 ${}_pF_q(\dots|z) = CB$ the entries of C are obviously

$$C = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Finally we have to compute the entries of the 3×3 matrix M such that $z \frac{d}{dz} B = MB$ holds. This is easy. We already computed the first part $z \frac{d}{dz} B_0$ above. This gives us the first row of M . For the second row we have:

```
>>> from diofant import Symbol, cosh, sqrt, diff
>>> z = Symbol("z")
>>> B1 = cosh(2*sqrt(z))
>>> z * diff(B1, z)
sqrt(z)*sinh(2*sqrt(z))
```

and for the third one

```
>>> from diofant import Symbol, sinh, sqrt, expand, diff
>>> z = Symbol("z")
>>> B2 = sinh(2*sqrt(z))*sqrt(z)
>>> expand(z * diff(B2, z))
sqrt(z)*sinh(2*sqrt(z))/2 + z*cosh(2*sqrt(z))
```

Now we have computed the entries of this matrix to be

$$M = \begin{pmatrix} -\frac{1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 \\ 0 & z & \frac{1}{2} \end{pmatrix}$$

Note that the entries of C and M should typically be rational functions in z , with rational coefficients. This is all we need to do in order to add a new formula to the lookup table for `hyperexpand`.

3.21.4 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 [Luke1969] (page 1274) and [Prudnikov1990] (page 1274). They are all tested numerically.

$$\begin{aligned} {}_0F_0(|z) &= e^z \\ {}_1F_0(a|z) &= (-z + 1)^{-a} \\ {}_2F_1\left(a, a - \frac{1}{2} \middle| z\right) &= 2^{2a-1} (\sqrt{-z + 1} + 1)^{-2a+1} \\ {}_2F_1\left(\begin{matrix} 1, 1 \\ 2 \end{matrix} \middle| z\right) &= -\frac{1}{z} \log(-z + 1) \\ {}_2F_1\left(\begin{matrix} \frac{1}{2}, 1 \\ \frac{3}{2} \end{matrix} \middle| z\right) &= \frac{1}{\sqrt{z}} \operatorname{atanh}(\sqrt{z}) \end{aligned}$$

$$\begin{aligned}
 {}_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 \operatorname{exp}_{\text{polar}}(i\pi))^{-a} \gamma(a, z \operatorname{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}, a \middle| z\right) &= -\frac{i\sqrt{\pi}a\sqrt{\frac{1}{z}}}{2a-1} \operatorname{erf}(i\sqrt{z}) - \frac{a (z \operatorname{exp}_{\text{polar}}(i\pi))^{-a}}{2a-1} \gamma(a, z \operatorname{exp}_{\text{polar}}(i\pi)) \\
 {}_2F_2\left(1, 1 \middle| z\right) &= \frac{1}{z} (-\log(z) + \operatorname{Ei}(z)) - \frac{\gamma}{z} \\
 {}_0F_1\left(\frac{1}{2} \middle| z\right) &= \cosh(2\sqrt{z}) \\
 {}_0F_1\left(b \middle| z\right) &= z^{-\frac{b}{2} + \frac{1}{2}} I_{b-1}(2\sqrt{z}) \Gamma(b) \\
 {}_0F_3\left(\frac{1}{2}, a, a + \frac{1}{2} \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(a, a + \frac{1}{2}, 2a \middle| z\right) &= \left(2\sqrt{z} \operatorname{exp}_{\text{polar}}\left(\frac{i\pi}{2}\right)\right)^{-2a+1} I_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} \operatorname{exp}_{\text{polar}}\left(\frac{i\pi}{4}\right)\right) J_{2a-1}\left(2\sqrt{2}\sqrt[4]{z} \operatorname{exp}_{\text{polar}}\left(\frac{i\pi}{4}\right)\right) \Gamma^2(2a) \\
 {}_1F_2\left(a - \frac{1}{2}, 2a \middle| z\right) &= 2 \cdot 4^{a-1} z^{-a+1} I_{a-\frac{3}{2}}(\sqrt{z}) I_{a-\frac{1}{2}}(\sqrt{z}) \Gamma\left(a - \frac{1}{2}\right) \Gamma\left(a + \frac{1}{2}\right) - 4^{a-\frac{1}{2}} z^{-a+\frac{1}{2}} I_{a-\frac{1}{2}}^2(\sqrt{z}) \Gamma^2\left(a + \frac{1}{2}\right) \\
 {}_1F_2\left(b, -\frac{1}{2} \middle| z\right) &= \frac{\pi I_{-b+1}(\sqrt{z}) I_{b-1}(\sqrt{z})}{\sin(\pi b)} (-b+1)
 \end{aligned}$$

$$\begin{aligned}
 {}_1F_2\left(\frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{1}{2\sqrt{z}} \operatorname{Shi}(2\sqrt{z}) \\
 {}_1F_2\left(\frac{3}{2}, \frac{7}{4} \middle| z\right) &= \frac{3\sqrt{\pi}}{4z^{\frac{3}{4}}} e^{-\frac{3i}{4}\pi} S\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) \\
 {}_1F_2\left(\frac{1}{2}, \frac{5}{4} \middle| z\right) &= \frac{e^{-\frac{i\pi}{4}}\sqrt{\pi}}{2\sqrt[4]{z}} C\left(\frac{2\sqrt[4]{z}}{\sqrt{\pi}} e^{\frac{i\pi}{4}}\right) \\
 {}_2F_3\left(2a, b, 2a - \frac{1}{2} - b + 1 \middle| z\right) &= \left(\frac{\sqrt{z}}{2}\right)^{-2a+1} I_{2a-b}(\sqrt{z}) I_{b-1}(\sqrt{z}) \Gamma(b) \Gamma(2a - b + 1) \\
 {}_2F_3\left(\frac{1}{2}, \frac{1}{2}, \frac{3}{2} \middle| z\right) &= \frac{1}{z} (-\log(2\sqrt{z}) + \operatorname{Chi}(2\sqrt{z})) - \frac{\gamma}{z} \\
 {}_3F_3\left(\frac{1}{2}, \frac{1}{2}, a \middle| z\right) &= -\frac{e^z a}{z(a^2 - 2a + 1)} + \frac{a(-z)^{-a}}{(a-1)^2} (\Gamma(a) - \Gamma(a, -z)) + \frac{a}{z(a^2 - 2a + 1)} (-a + 1) (\log(-z) + \operatorname{E}_1(-z) + \gamma) +
 \end{aligned}$$

diofant.simplify.hyperexpand.add_formulae(formulae)
 Create our knowledge base.

3.21.5 References

3.22 Stats

Diofant statistics module

Introduces a random variable type into the Diofant language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc... or built with functions like FiniteRV.

Queries on random expressions can be made using the functions

Expression	Meaning
P(condition)	Probability
E(expression)	Expected value
variance(expression)	Variance
density(expression)	Probability Density Function
sample(expression)	Produce a realization
where(condition)	Where the condition is true

3.22.1 Examples

```

>>> from diofant.stats import P, E, variance, Die, Normal
>>> from diofant import Eq, simplify
>>> X, Y = Die('X', 6), Die('Y', 6) # Define two six sided dice
>>> Z = Normal('Z', 0, 1) # Declare a Normal random variable with mean 0, std 1
>>> P(X>3) # Probability X is greater than 3
1/2
>>> E(X+Y) # Expectation of the sum of two dice
7
>>> variance(X+Y) # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1)) # Probability of Z being greater than 1
-erf(sqrt(2)/2)/2 + 1/2
    
```


3.22.2 Random Variable Types

Finite Types

`diofant.stats.DiscreteUniform(name, items)`

Create a Finite Random Variable representing a uniform distribution over the input set.

Returns a `RandomSymbol`.

Examples

```
>>> from diofant.stats import DiscreteUniform, density
>>> from diofant import symbols, Rational, Symbol
```

```
>>> s3, s5 = Rational(1, 3), Rational(1, 5)
```

```
>>> X = DiscreteUniform('X', symbols('a b c')) # equally likely over a, b, c
>>> density(X).dict == {Symbol('a'): s3, Symbol('b'): s3, Symbol('c'): s3}
True
```

```
>>> Y = DiscreteUniform('Y', list(range(5))) # distribution over a range
>>> density(Y).dict == {0: s5, 1: s5, 2: s5, 3: s5, 4: s5}
True
```

`diofant.stats.Die(name, sides=6)`

Create a Finite Random Variable representing a fair die.

Returns a `RandomSymbol`.

```
>>> from diofant.stats import Die, density
```

```
>>> D6 = Die('D6', 6) # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
```

```
>>> D4 = Die('D4', 4) # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

`diofant.stats.Bernoulli(name, p, succ=1, fail=0)`

Create a Finite Random Variable representing a Bernoulli process.

Returns a `RandomSymbol`

```
>>> from diofant.stats import Bernoulli, density
>>> from diofant import Rational, Symbol
```

```
>>> X = Bernoulli('X', Rational(3, 4)) # 1-0 Bernoulli variable, probability = 3/4
>>> density(X).dict == {0: Rational(1, 4), 1: Rational(3, 4)}
True
```

```
>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X).dict == {Symbol('Heads'): Rational(1, 2), Symbol('Tails'):
↳Rational(1, 2)}
True
```

`diofant.stats.Coin(name, p=Rational(1, 2))`

Create a Finite Random Variable representing a Coin toss.

Probability p is the chance of getting "Heads." Half by default

Returns a RandomSymbol.

```
>>> from diofant.stats import Coin, density
>>> from diofant import Rational, Symbol
```

```
>>> H, T = Symbol('H'), Symbol('T')
```

```
>>> C = Coin('C') # A fair coin toss
>>> density(C).dict == {H: Rational(1, 2), T: Rational(1, 2)}
True
```

```
>>> C2 = Coin('C2', Rational(3, 5)) # An unfair coin
>>> density(C2).dict == {H: Rational(3, 5), T: Rational(2, 5)}
True
```

`diofant.stats.Binomial(name, n, p, succ=1, fail=0)`

Create a Finite Random Variable representing a binomial distribution.

Returns a RandomSymbol.

Examples

```
>>> from diofant.stats import Binomial, density
>>> from diofant import S
```

```
>>> X = Binomial('X', 4, S.Half) # Four "coin flips"
>>> density(X).dict
{0: 1/16, 1: 1/4, 2: 3/8, 3: 1/4, 4: 1/16}
```

`diofant.stats.Hypergeometric(name, N, m, n)`

Create a Finite Random Variable representing a hypergeometric distribution.

Returns a RandomSymbol.

Examples

```
>>> from diofant.stats import Hypergeometric, density
>>> from diofant import S
```

```
>>> X = Hypergeometric('X', 10, 5, 3) # 10 marbles, 5 white (success), 3 draws
>>> density(X).dict
{0: 1/12, 1: 5/12, 2: 5/12, 3: 1/12}
```

`diofant.stats.FiniteRV(name, density)`

Create a Finite Random Variable given a dict representing the density.

Returns a RandomSymbol.

```
>>> from diofant.stats import FiniteRV, P, E
```

```
>>> density = {0: .1, 1: .2, 2: .3, 3: .4}
>>> X = FiniteRV('X', density)
```

```
>>> E(X)
2.0000000000000000
>>> P(X >= 2)
0.7000000000000000
```

Discrete Types

`diofant.stats.Geometric`(*name*, *p*)

Create a discrete random variable with a Geometric distribution.

The density of the Geometric distribution is given by

$$f(k) := p(1 - p)^{k-1}$$

Parameters p: A probability between 0 and 1

Returns A RandomSymbol.

References

[1] http://en.wikipedia.org/wiki/Geometric_distribution [2] <http://mathworld.wolfram.com/GeometricDistribution.html>

Examples

```
>>> from diofant.stats import Geometric, density, E, variance
>>> from diofant import Symbol, S
```

```
>>> p = S.One / 5
>>> z = Symbol("z")
```

```
>>> X = Geometric("x", p)
```

```
>>> density(X)(z)
(4/5)**(z - 1)/5
```

```
>>> E(X)
5
```

```
>>> variance(X)
20
```

`diofant.stats.Poisson`(*name*, *lamda*)

Create a discrete random variable with a Poisson distribution.

The density of the Poisson distribution is given by

$$f(k) := \frac{\lambda^k e^{-\lambda}}{k!}$$

Parameters lamda: Positive number, a rate

Returns A RandomSymbol.

References

[1] http://en.wikipedia.org/wiki/Poisson_distribution [2] <http://mathworld.wolfram.com/PoissonDistribution.html>

Examples

```
>>> from diofant.stats import Poisson, density, E, variance
>>> from diofant import Symbol, simplify
```

```
>>> rate = Symbol("lambda", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Poisson("x", rate)
```

```
>>> density(X)(z)
E**(-lambda)*lambda**z/factorial(z)
```

```
>>> E(X)
lambda
```

```
>>> simplify(variance(X))
lambda
```

Continuous Types

`diofant.stats.Arcsin`(*name*, *a=0*, *b=1*)

Create a Continuous Random Variable with an arcsin distribution.

The density of the arcsin distribution is given by

$$f(x) := \frac{1}{\pi\sqrt{(x-a)(b-x)}}$$

with $x \in [a, b]$. It must hold that $-\infty < a < b < \infty$.

Parameters a : Real number, the left interval boundary

b : Real number, the right interval boundary

Returns A RandomSymbol.

References

[R524] (page 1274)

Examples

```
>>> from diofant.stats import Arcsin, density
>>> from diofant import Symbol, simplify
```

```
>>> a = Symbol("a", extended_real=True)
>>> b = Symbol("b", extended_real=True)
>>> z = Symbol("z")
```

```
>>> X = Arcsin("x", a, b)
```

```
>>> density(X)(z)
1/(pi*sqrt((-a + z)*(b - z)))
```

`diofant.stats.Benini`(*name*, *alpha*, *beta*, *sigma*)

Create a Continuous Random Variable with a Benini distribution.

The density of the Benini distribution is given by

$$f(x) := e^{-\alpha \log \frac{x}{\sigma} - \beta \log^2 \left[\frac{x}{\sigma} \right]} \left(\frac{\alpha}{x} + \frac{2\beta \log \frac{x}{\sigma}}{x} \right)$$

This is a heavy-tailed distribution and is also known as the log-Rayleigh distribution.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

sigma : Real number, $\sigma > 0$, a scale

Returns A RandomSymbol.

References

[R525] (page 1274), [R526] (page 1274)

Examples

```
>>> from diofant.stats import Benini, density
>>> from diofant import Symbol, simplify, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Benini("x", alpha, beta, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
E
- alpha*log|-----| - beta*log |-----| |
      / z \      2/ z \ /      / z \ \
      \sigma/      \sigma/ |alpha      \sigma|
*|-----| + -----|
  \ z      z      /
```

`diofant.stats.Beta`(*name*, *alpha*, *beta*)

Create a Continuous Random Variable with a Beta distribution.

The density of the Beta distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}$$

with $x \in [0, 1]$.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

Returns A RandomSymbol.

References

[R527] (page 1274), [R528] (page 1274)

Examples

```
>>> from diofant.stats import Beta, density, E, variance
>>> from diofant import Symbol, simplify, pprint, expand_func
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Beta("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      beta - 1
z      *(-z + 1)
-----
beta(alpha, beta)
```

```
>>> expand_func(simplify(E(X, meijerg=True)))
alpha/(alpha + beta)
```

`diofant.stats.BetaPrime`(*name*, *alpha*, *beta*)

Create a continuous random variable with a Beta prime distribution.

The density of the Beta prime distribution is given by

$$f(x) := \frac{x^{\alpha-1}(1+x)^{-\alpha-\beta}}{B(\alpha, \beta)}$$

with $x > 0$.

Parameters **alpha** : Real number, $\alpha > 0$, a shape

beta : Real number, $\beta > 0$, a shape

Returns A RandomSymbol.

References

[R529] (page 1274), [R530] (page 1274)

Examples

```
>>> from diofant.stats import BetaPrime, density
>>> from diofant import Symbol, pprint
```

```
>>> alpha = Symbol("alpha", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = BetaPrime("x", alpha, beta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
alpha - 1      -alpha - beta
z      *(z + 1)
-----
beta(alpha, beta)
```

`diofant.stats.Cauchy`(*name*, *x0*, *gamma*)

Create a continuous random variable with a Cauchy distribution.

The density of the Cauchy distribution is given by

$$f(x) := \frac{1}{\pi} \arctan\left(\frac{x - x_0}{\gamma}\right) + \frac{1}{2}$$

Parameters **x0** : Real number, the location

gamma : Real number, $\gamma > 0$, the scale

Returns A `RandomSymbol`.

References

[R531] (page 1274), [R532] (page 1274)

Examples

```
>>> from diofant.stats import Cauchy, density
>>> from diofant import Symbol
```

```
>>> x0 = Symbol("x0")
>>> gamma = Symbol("gamma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Cauchy("x", x0, gamma)
```

```
>>> density(X)(z)
1/(pi*gamma*(1 + (-x0 + z)**2/gamma**2))
```

`diofant.stats.Chi`(*name*, *k*)

Create a continuous random variable with a Chi distribution.

The density of the Chi distribution is given by

$$f(x) := \frac{2^{1-k/2} x^{k-1} e^{-x^2/2}}{\Gamma(k/2)}$$

with $x \geq 0$.

Parameters *k* : A positive Integer, $k > 0$, the number of degrees of freedom

Returns A RandomSymbol.

References

[R533] (page 1274), [R534] (page 1274)

Examples

```
>>> from diofant.stats import Chi, density, E, std
>>> from diofant import Symbol, simplify
```

```
>>> k = Symbol("k", integer=True)
>>> z = Symbol("z")
```

```
>>> X = Chi("x", k)
```

```
>>> density(X)(z)
2**(-k/2 + 1)*E**(-z**2/2)*z**(k - 1)/gamma(k/2)
```

`diofant.stats.ChiNoncentral`(*name*, *k*, *l*)

Create a continuous random variable with a non-central Chi distribution.

The density of the non-central Chi distribution is given by

$$f(x) := \frac{e^{-(x^2+\lambda^2)/2} x^k \lambda}{(\lambda x)^{k/2}} I_{k/2-1}(\lambda x)$$

with $x \geq 0$. Here, $I_\nu(x)$ is the *modified Bessel function of the first kind* (page 389).

Parameters *k* : A positive Integer, $k > 0$, the number of degrees of freedom

l : Shift parameter

Returns A RandomSymbol.

References

[R535] (page 1274)

Examples

```
>>> from diofant.stats import ChiNoncentral, density, E, std
>>> from diofant import Symbol, simplify
```

```
>>> k = Symbol("k", integer=True)
>>> l = Symbol("l")
>>> z = Symbol("z")
```

```
>>> X = ChiNoncentral("x", k, l)
```

```
>>> density(X)(z)
E**(-l**2/2 - z**2/2)*l*z**k*(l*z)**(-k/2)*besseli(k/2 - 1, l*z)
```

`diofant.stats.ChiSquared(name, k)`

Create a continuous random variable with a Chi-squared distribution.

The density of the Chi-squared distribution is given by

$$f(x) := \frac{1}{2^{\frac{k}{2}} \Gamma(\frac{k}{2})} x^{\frac{k}{2}-1} e^{-\frac{x}{2}}$$

with $x \geq 0$.

Parameters `k`: A positive Integer, $k > 0$, the number of degrees of freedom

Returns A RandomSymbol.

References

[R536] (page 1274), [R537] (page 1274)

Examples

```
>>> from diofant.stats import ChiSquared, density, E, variance
>>> from diofant import Symbol, simplify, combsimp, expand_func
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> z = Symbol("z")
```

```
>>> X = ChiSquared("x", k)
```

```
>>> density(X)(z)
2**(-k/2)*E**(-z/2)*z**(k/2 - 1)/gamma(k/2)
```

```
>>> combsimp(E(X))
k
```

```
>>> simplify(expand_func(variance(X)))
2*k
```

`diofant.stats.Dagum(name, p, a, b)`

Create a continuous random variable with a Dagum distribution.

The density of the Dagum distribution is given by

$$f(x) := \frac{ap}{x} \left(\frac{\left(\frac{x}{b}\right)^{ap}}{\left(\left(\frac{x}{b}\right)^a + 1\right)^{p+1}} \right)$$

with $x > 0$.

Parameters **p** : Real number, $p > 0$, a shape

a : Real number, $a > 0$, a shape

b : Real number, $b > 0$, a scale

Returns A RandomSymbol.

References

[R538] (page 1274)

Examples

```
>>> from diofant.stats import Dagum, density
>>> from diofant import Symbol, simplify
```

```
>>> p = Symbol("p", positive=True)
>>> b = Symbol("b", positive=True)
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Dagum("x", p, a, b)
```

```
>>> density(X)(z)
a*p*(z/b)**(a*p)*((z/b)**a + 1)**(-p - 1)/z
```

`diofant.stats.Erlang(name, k, l)`

Create a continuous random variable with an Erlang distribution.

The density of the Erlang distribution is given by

$$f(x) := \frac{\lambda^k x^{k-1} e^{-\lambda x}}{(k-1)!}$$

with $x \in [0, \infty]$.

Parameters **k** : Integer

l : Real number, $\lambda > 0$, the rate

Returns A RandomSymbol.

References

[R539] (page 1274), [R540] (page 1274)

Examples

```
>>> from diofant.stats import Erlang, density, cdf, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> k = Symbol("k", integer=True, positive=True)
>>> l = Symbol("l", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Erlang("x", k, l)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
-l*z  k  k - 1
E    *l *z
-----
gamma(k)
```

```
>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/ k*lowergamma(k, 0)  k*lowergamma(k, l*z)
|-----+----- for z >= 0
| gamma(k + 1)      gamma(k + 1)
|
| 0
\ otherwise
```

```
>>> simplify(E(X))
k/l
```

```
>>> simplify(variance(X))
k/l**2
```

`diofant.stats.Exponential`(*name*, *rate*)

Create a continuous random variable with an Exponential distribution.

The density of the exponential distribution is given by

$$f(x) := \lambda \exp(-\lambda x)$$

with $x > 0$. Note that the expected value is $1/\lambda$.

Parameters *rate* : A positive Real number, $\lambda > 0$, the rate (or inverse scale/inverse mean)

Returns A RandomSymbol.

References

[R541] (page 1274), [R542] (page 1274)

Examples

```
>>> from diofant.stats import Exponential, density, cdf, E
>>> from diofant.stats import variance, std, skewness
>>> from diofant import Symbol
```

```
>>> l = Symbol("lambda", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Exponential("x", l)
```

```
>>> density(X)(z)
E**(-lambda*z)*lambda
```

```
>>> cdf(X)(z)
Piecewise((1 - E**(-lambda*z), z >= 0), (0, true))
```

```
>>> E(X)
1/lambda
```

```
>>> variance(X)
lambda**(-2)
```

```
>>> skewness(X)
2
```

```
>>> X = Exponential('x', 10)
```

```
>>> density(X)(z)
10*E**(-10*z)
```

```
>>> E(X)
1/10
```

```
>>> std(X)
1/10
```

`diofant.stats.FDistribution`(*name*, *d1*, *d2*)

Create a continuous random variable with a F distribution.

The density of the F distribution is given by

$$f(x) := \frac{\sqrt{\frac{(d_1 x)^{d_1} d_2^{d_2}}{(d_1 x + d_2)^{d_1 + d_2}}}}{x \mathbf{B}\left(\frac{d_1}{2}, \frac{d_2}{2}\right)}$$

with $x > 0$.

Parameters **d1** : $d_1 > 0$ a parameter

d2 : $d_2 > 0$ a parameter

Returns A RandomSymbol.

References

[R543] (page 1274), [R544] (page 1274)

Examples

```
>>> from diofant.stats import FDistribution, density
>>> from diofant import Symbol, simplify, pprint
```

```
>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")
```

```
>>> X = FDistribution("x", d1, d2)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
d2
--
2 / -----
d1 -d1 - d2
d2 * \ / (d1*z) *(d1*z + d2)
-----
/d1 d2\
z*beta|--, --|
 \2 2 /
```

`diofant.stats.FisherZ(name, d1, d2)`

Create a Continuous Random Variable with an Fisher's Z distribution.

The density of the Fisher's Z distribution is given by

$$f(x) := \frac{2d_1^{d_1/2} d_2^{d_2/2}}{B(d_1/2, d_2/2)} \frac{e^{d_1 z}}{(d_1 e^{2z} + d_2)^{(d_1+d_2)/2}}$$

Parameters **d1** : $d_1 > 0$, degree of freedom

d2 : $d_2 > 0$, degree of freedom

Returns A RandomSymbol.

References

[R545] (page 1274), [R546] (page 1274)

Examples

```
>>> from diofant.stats import FisherZ, density
>>> from diofant import Symbol, simplify, pprint
```

```
>>> d1 = Symbol("d1", positive=True)
>>> d2 = Symbol("d2", positive=True)
>>> z = Symbol("z")
```

```
>>> X = FisherZ("x", d1, d2)
```

```

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
          d1  d2          -  - - -  -  - -
          --  --          2    2
d1*z  2    2  / 2*z  \
2*E   *d1 *d2 *\E   *d1 + d2/
-----
          /d1  d2\
          beta|--, --|
          \2    2 /

```

`diofant.stats.Frechet`(*name*, *a*, *s*=1, *m*=0)
 Create a continuous random variable with a Frechet distribution.
 The density of the Frechet distribution is given by

$$f(x) := \frac{\alpha}{s} \left(\frac{x - m}{s} \right)^{-1-\alpha} e^{-\left(\frac{x-m}{s}\right)^{-\alpha}}$$

with $x \geq m$.

Parameters a : Real number, $a \in (0, \infty)$ the shape

s : Real number, $s \in (0, \infty)$ the scale

m : Real number, $m \in (-\infty, \infty)$ the minimum

Returns A RandomSymbol.

References

[R547] (page 1274)

Examples

```

>>> from diofant.stats import Frechet, density, E, std
>>> from diofant import Symbol, simplify

```

```

>>> a = Symbol("a", positive=True)
>>> s = Symbol("s", positive=True)
>>> m = Symbol("m", extended_real=True)
>>> z = Symbol("z")

```

```

>>> X = Frechet("x", a, s, m)

```

```

>>> density(X)(z)
E**(-((-m + z)/s)**(-a))*a*((-m + z)/s)**(-a - 1)/s

```

`diofant.stats.Gamma`(*name*, *k*, *theta*)
 Create a continuous random variable with a Gamma distribution.
 The density of the Gamma distribution is given by

$$f(x) := \frac{1}{\Gamma(k)\theta^k} x^{k-1} e^{-\frac{x}{\theta}}$$

with $x \in [0, 1]$.

Parameters **k** : Real number, $k > 0$, a shape

theta : Real number, $\theta > 0$, a scale

Returns A RandomSymbol.

References

[R548] (page 1274), [R549] (page 1274)

Examples

```
>>> from diofant.stats import Gamma, density, cdf, E, variance
>>> from diofant import Symbol, pprint, simplify
```

```
>>> k = Symbol("k", positive=True)
>>> theta = Symbol("theta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Gamma("x", k, theta)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
  -z
  ----
  theta      -k  k - 1
  E      *theta *z
  -----
  gamma(k)
```

```
>>> C = cdf(X, meijerg=True)(z)
>>> pprint(C, use_unicode=False)
/
|          k*lowergamma|k, ----|
| k*lowergamma(k, 0)   \  theta/
<- ----- + ----- for z >= 0
|      gamma(k + 1)      gamma(k + 1)
|
\          0
\          otherwise
```

```
>>> E(X)
theta*gamma(k + 1)/gamma(k)
```

```
>>> V = simplify(variance(X))
>>> pprint(V, use_unicode=False)
  2
k*theta
```

`diofant.stats.GammaInverse`(*name*, *a*, *b*)

Create a continuous random variable with an inverse Gamma distribution.

The density of the inverse Gamma distribution is given by

$$f(x) := \frac{\beta^\alpha}{\Gamma(\alpha)} x^{-\alpha-1} \exp\left(\frac{-\beta}{x}\right)$$

with $x > 0$.

Parameters **a** : Real number, $a > 0$ a shape

b : Real number, $b > 0$ a scale

Returns A RandomSymbol.

References

[R550] (page 1274)

Examples

```
>>> from diofant.stats import GammaInverse, density, cdf, E, variance
>>> from diofant import Symbol, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = GammaInverse("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
-b
---
z  a  -a - 1
E  *b *z
-----
gamma(a)
```

`diofant.stats.Kumaraswamy`(*name*, *a*, *b*)

Create a Continuous Random Variable with a Kumaraswamy distribution.

The density of the Kumaraswamy distribution is given by

$$f(x) := abx^{a-1}(1-x^a)^{b-1}$$

with $x \in [0, 1]$.

Parameters **a** : Real number, $a > 0$ a shape

b : Real number, $b > 0$ a shape

Returns A RandomSymbol.

References

[R551] (page 1274)

Examples


```
>>> from diofant.stats import Kumaraswamy, density, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> a = Symbol("a", positive=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Kumaraswamy("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      b - 1
  a - 1 / a \
a*b*z    *\ - z + 1/
```

`diofant.stats.Laplace`(*name*, *mu*, *b*)

Create a continuous random variable with a Laplace distribution.

The density of the Laplace distribution is given by

$$f(x) := \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

Parameters *mu* : Real number, the location (mean)

b : Real number, $b > 0$, a scale

Returns A RandomSymbol.

References

[R552] (page 1274), [R553] (page 1274)

Examples

```
>>> from diofant.stats import Laplace, density
>>> from diofant import Symbol
```

```
>>> mu = Symbol("mu")
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Laplace("x", mu, b)
```

```
>>> density(X)(z)
E**(-Abs(mu - z)/b)/(2*b)
```

`diofant.stats.Logistic`(*name*, *mu*, *s*)

Create a continuous random variable with a logistic distribution.

The density of the logistic distribution is given by

$$f(x) := \frac{e^{-(x-\mu)/s}}{s(1 + e^{-(x-\mu)/s})^2}$$

Parameters **mu** : Real number, the location (mean)

s : Real number, $s > 0$ a scale

Returns A RandomSymbol.

References

[R554] (page 1274), [R555] (page 1274)

Examples

```
>>> from diofant.stats import Logistic, density
>>> from diofant import Symbol
```

```
>>> mu = Symbol("mu", extended_real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Logistic("x", mu, s)
```

```
>>> density(X)(z)
E**((mu - z)/s)/(s*(E**((mu - z)/s) + 1)**2)
```

`diofant.stats.LogNormal`(*name*, *mean*, *std*)

Create a continuous random variable with a log-normal distribution.

The density of the log-normal distribution is given by

$$f(x) := \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

with $x \geq 0$.

Parameters **mu** : Real number, the log-scale

sigma : Real number, $\sigma^2 > 0$ a shape

Returns A RandomSymbol.

References

[R556] (page 1274), [R557] (page 1274)

Examples

```
>>> from diofant.stats import LogNormal, density
>>> from diofant import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", extended_real=True)
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = LogNormal("x", mu, sigma)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      2
      -(-mu + log(z))
      -----
      2
      2*sigma
      -----
      \ / 2 *E
      -----
      2*\ / pi *sigma*z
```

```
>>> X = LogNormal('x', 0, 1) # Mean 0, standard deviation 1
```

```
>>> density(X)(z)
sqrt(2)*E**(-log(z)**2/2)/(2*sqrt(pi)*z)
```

`diofant.stats.Maxwell(name, a)`

Create a continuous random variable with a Maxwell distribution.

The density of the Maxwell distribution is given by

$$f(x) := \sqrt{\frac{2}{\pi}} \frac{x^2 e^{-x^2/(2a^2)}}{a^3}$$

with $x \geq 0$.

Parameters a : Real number, $a > 0$

Returns A RandomSymbol.

References

[R558] (page 1274), [R559] (page 1275)

Examples

```
>>> from diofant.stats import Maxwell, density, E, variance
>>> from diofant import Symbol, simplify
```

```
>>> a = Symbol("a", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Maxwell("x", a)
```

```
>>> density(X)(z)
sqrt(2)*E**(-z**2/(2*a**2))*z**2/(sqrt(pi)*a**3)
```

```
>>> E(X)
2*sqrt(2)*a/sqrt(pi)
```

```
>>> simplify(variance(X))
a**2*(-8 + 3*pi)/pi
```

`diofant.stats.Nakagami`(*name*, *mu*, *omega*)

Create a continuous random variable with a Nakagami distribution.

The density of the Nakagami distribution is given by

$$f(x) := \frac{2\mu^\mu}{\Gamma(\mu)\omega^\mu} x^{2\mu-1} \exp\left(-\frac{\mu}{\omega}x^2\right)$$

with $x > 0$.

Parameters **mu** : Real number, $\mu \geq \frac{1}{2}$ a shape

omega : Real number, $\omega > 0$, the spread

Returns A RandomSymbol.

References

[R560] (page 1275)

Examples

```
>>> from diofant.stats import Nakagami, density, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", positive=True)
>>> omega = Symbol("omega", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Nakagami("x", mu, omega)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      2
    -mu*z
-----
  omega      mu      -mu 2*mu - 1
2*E      *mu *omega *z
-----
                    gamma(mu)
```

```
>>> simplify(E(X, meijerg=True))
sqrt(mu)*sqrt(omega)*gamma(mu + 1/2)/gamma(mu + 1)
```

```
>>> V = simplify(variance(X, meijerg=True))
>>> pprint(V, use_unicode=False)
      2
    omega*gamma (mu + 1/2)
omega - -----
    gamma(mu)*gamma(mu + 1)
```

`diofant.stats.Normal(name, mean, std)`

Create a continuous random variable with a Normal distribution.

The density of the Normal distribution is given by

$$f(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Parameters `mu` : Real number, the mean

`sigma` : Real number, $\sigma^2 > 0$ the variance

Returns A RandomSymbol.

References

[R561] (page 1275), [R562] (page 1275)

Examples

```
>>> from diofant.stats import Normal, density, E, std, cdf, skewness
>>> from diofant import Symbol, simplify, pprint, factor, together, factor_terms
```

```
>>> mu = Symbol("mu")
>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Normal("x", mu, sigma)
```

```
>>> density(X)(z)
sqrt(2)*E**(-(-mu + z)**2/(2*sigma**2))/(2*sqrt(pi)*sigma)
```

```
>>> C = simplify(cdf(X))(z) # it needs a little more help...
>>> pprint(C, use_unicode=False)
      / _____ \
      | \ / 2 *(mu - z) |
  erf|-----|
      \      2*sigma    /  1
  - ---- + -
      2                    2
```

```
>>> simplify(skewness(X))
0
```

```
>>> X = Normal("x", 0, 1) # Mean 0, standard deviation 1
>>> density(X)(z)
sqrt(2)*E**(-z**2/2)/(2*sqrt(pi))
```

```
>>> E(2*X + 1)
1
```

```
>>> simplify(std(2*X + 1))
2
```

`diofant.stats.Pareto`(*name*, *xm*, *alpha*)

Create a continuous random variable with the Pareto distribution.

The density of the Pareto distribution is given by

$$f(x) := \frac{\alpha x_m^\alpha}{x^{\alpha+1}}$$

with $x \in [x_m, \infty]$.

Parameters **xm** : Real number, $x_m > 0$, a scale

alpha : Real number, $\alpha > 0$, a shape

Returns A RandomSymbol.

References

[R563] (page 1275), [R564] (page 1275)

Examples

```
>>> from diofant.stats import Pareto, density
>>> from diofant import Symbol
```

```
>>> xm = Symbol("xm", positive=True)
>>> beta = Symbol("beta", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Pareto("x", xm, beta)
```

```
>>> density(X)(z)
beta*xm**beta*z**(-beta - 1)
```

`diofant.stats.QuadraticU`(*name*, *a*, *b*)

Create a Continuous Random Variable with a U-quadratic distribution.

The density of the U-quadratic distribution is given by

$$f(x) := \alpha(x - \beta)^2$$

with $x \in [a, b]$.

Parameters **a** : Real number

b : Real number, $a < b$

Returns A RandomSymbol.

References

[R565] (page 1275)

Examples

```
>>> from diofant.stats import QuadraticU, density, E, variance
>>> from diofant import Symbol, simplify, factor, pprint
```

```
>>> a = Symbol("a", extended_real=True)
>>> b = Symbol("b", extended_real=True)
>>> z = Symbol("z")
```

```
>>> X = QuadraticU("x", a, b)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/
| / a b \
|12*|- - - + z|
| \ 2 2 /
<----- for And(a <= z, z <= b)
|
|      3
|      (-a + b)
|
|      0
\      otherwise
```

`diofant.stats.RaisedCosine`(*name*, *mu*, *s*)

Create a Continuous Random Variable with a raised cosine distribution.

The density of the raised cosine distribution is given by

$$f(x) := \frac{1}{2s} \left(1 + \cos \left(\frac{x - \mu}{s} \pi \right) \right)$$

with $x \in [\mu - s, \mu + s]$.

Parameters mu : Real number

s : Real number, $s > 0$

Returns A RandomSymbol.

References

[R566] (page 1275)

Examples

```
>>> from diofant.stats import RaisedCosine, density, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu", extended_real=True)
>>> s = Symbol("s", positive=True)
>>> z = Symbol("z")
```

```
>>> X = RaisedCosine("x", mu, s)
```

```

>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
/ /pi*(-mu + z)\
|cos|-----| + 1
| \      s      /
<----- for And(z <= mu + s, mu - s <= z)
|      2*s
|
\      0
                    otherwise

```

`diofant.stats.Rayleigh`(*name*, *sigma*)

Create a continuous random variable with a Rayleigh distribution.

The density of the Rayleigh distribution is given by

$$f(x) := \frac{x}{\sigma^2} e^{-x^2/2\sigma^2}$$

with $x > 0$.

Parameters `sigma` : Real number, $\sigma > 0$

Returns A RandomSymbol.

References

[R567] (page 1275), [R568] (page 1275)

Examples

```

>>> from diofant.stats import Rayleigh, density, E, variance
>>> from diofant import Symbol, simplify

```

```

>>> sigma = Symbol("sigma", positive=True)
>>> z = Symbol("z")

```

```

>>> X = Rayleigh("x", sigma)

```

```

>>> density(X)(z)
E**(-z**2/(2*sigma**2))*z/sigma**2

```

```

>>> E(X)
sqrt(2)*sqrt(pi)*sigma/2

```

```

>>> variance(X)
-pi*sigma**2/2 + 2*sigma**2

```

`diofant.stats.StudentT`(*name*, *nu*)

Create a continuous random variable with a student's t distribution.

The density of the student's t distribution is given by

$$f(x) := \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)} \left(1 + \frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}$$

Parameters `nu` : Real number, $\nu > 0$, the degrees of freedom

Returns A `RandomSymbol`.

References

[R569] (page 1275), [R570] (page 1275)

Examples

```
>>> from diofant.stats import StudentT, density, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> nu = Symbol("nu", positive=True)
>>> z = Symbol("z")
```

```
>>> X = StudentT("x", nu)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      nu  1
      -  -  -
      2    2

 /      2\
 |      z |
 | 1 + -- |
 \      nu/
-----
\ / nu *beta|1/2, --|
 \              2 /
```

`diofant.stats.Triangular`(*name*, *a*, *b*, *c*)

Create a continuous random variable with a triangular distribution.

The density of the triangular distribution is given by

$$f(x) := \begin{cases} 0 & \text{for } x < a, \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{for } a \leq x < c, \\ \frac{2}{b-a} & \text{for } x = c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{for } c < x \leq b, \\ 0 & \text{for } b < x. \end{cases}$$

Parameters `a` : Real number, $a \in (-\infty, \infty)$

`b` : Real number, $a < b$

`c` : Real number, $a \leq c \leq b$

Returns A `RandomSymbol`.

References

[R571] (page 1275), [R572] (page 1275)

Examples

```
>>> from diofant.stats import Triangular, density, E
>>> from diofant import Symbol, pprint
```

```
>>> a = Symbol("a")
>>> b = Symbol("b")
>>> c = Symbol("c")
>>> z = Symbol("z")
```

```
>>> X = Triangular("x", a,b,c)
```

```
>>> pprint(density(X)(z), use_unicode=False)
/      -2*a + 2*z
|----- for And(a <= z, z < c)
| (-a + b)*(-a + c)
|
|      2
|      ----- for z = c
<      -a + b
|
|      2*b - 2*z
|----- for And(z <= b, c < z)
| (-a + b)*(b - c)
|
|      0
\      otherwise
```

`diofant.stats.Uniform(name, left, right)`

Create a continuous random variable with a uniform distribution.

The density of the uniform distribution is given by

$$f(x) := \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

with $x \in [a, b]$.

Parameters **a** : Real number, $-\infty < a$ the left boundary

b : Real number, $a < b < \infty$ the right boundary

Returns A RandomSymbol.

References

[R573] (page 1275), [R574] (page 1275)

Examples

```
>>> from diofant.stats import Uniform, density, cdf, E, variance, skewness
>>> from diofant import Symbol, simplify
```

```
>>> a = Symbol("a", negative=True)
>>> b = Symbol("b", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Uniform("x", a, b)
```

```
>>> density(X)(z)
Piecewise((1/(-a + b), And(a <= z, z <= b)), (0, true))
```

```
>>> cdf(X)(z)
-a/(-a + b) + z/(-a + b)
```

```
>>> simplify(E(X))
a/2 + b/2
```

```
>>> simplify(variance(X))
a**2/12 - a*b/6 + b**2/12
```

`diofant.stats.UniformSum(name, n)`

Create a continuous random variable with an Irwin-Hall distribution.

The probability distribution function depends on a single parameter n which is an integer.

The density of the Irwin-Hall distribution is given by

$$f(x) := \frac{1}{(n-1)!} \sum_{k=0}^{\lfloor x \rfloor} (-1)^k \binom{n}{k} (x-k)^{n-1}$$

Parameters n : A positive Integer, $n > 0$

Returns A RandomSymbol.

References

[R575] (page 1275), [R576] (page 1275)

Examples

```
>>> from diofant.stats import UniformSum, density
>>> from diofant import Symbol, pprint
```

```
>>> n = Symbol("n", integer=True)
>>> z = Symbol("z")
```

```
>>> X = UniformSum("x", n)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
floor(z)
```

$$\sum_{k=0}^{\lfloor z \rfloor} (-1)^k (z-k)^{n-1} \frac{n!}{k!}$$

```
-----
(n - 1)!
```

`diofant.stats.VonMises`(*name*, *mu*, *k*)

Create a Continuous Random Variable with a von Mises distribution.

The density of the von Mises distribution is given by

$$f(x) := \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

with $x \in [0, 2\pi]$.

Parameters *mu* : Real number, measure of location

k : Real number, measure of concentration

Returns A RandomSymbol.

References

[R577] (page 1275), [R578] (page 1275)

Examples

```
>>> from diofant.stats import VonMises, density, E, variance
>>> from diofant import Symbol, simplify, pprint
```

```
>>> mu = Symbol("mu")
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = VonMises("x", mu, k)
```

```
>>> D = density(X)(z)
>>> pprint(D, use_unicode=False)
      k*cos(mu - z)
      E
-----
2*pi*besseli(0, k)
```

`diofant.stats.Weibull`(*name*, *alpha*, *beta*)

Create a continuous random variable with a Weibull distribution.

The density of the Weibull distribution is given by

$$f(x) := \begin{cases} \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Parameters *lambda* : Real number, $\lambda > 0$ a scale

k : Real number, $k > 0$ a shape

Returns A RandomSymbol.

References

[R579] (page 1275), [R580] (page 1275)

Examples

```
>>> from diofant.stats import Weibull, density, E, variance
>>> from diofant import Symbol, simplify
```

```
>>> l = Symbol("lambda", positive=True)
>>> k = Symbol("k", positive=True)
>>> z = Symbol("z")
```

```
>>> X = Weibull("x", l, k)
```

```
>>> density(X)(z)
E**(-(z/lambda)**k)*k*(z/lambda)**(k - 1)/lambda
```

```
>>> simplify(E(X))
lambda*gamma(1 + 1/k)
```

```
>>> simplify(variance(X))
lambda**2*(-gamma(1 + 1/k)**2 + gamma(1 + 2/k))
```

`diofant.stats.WignerSemicircle(name, R)`

Create a continuous random variable with a Wigner semicircle distribution.

The density of the Wigner semicircle distribution is given by

$$f(x) := \frac{2}{\pi R^2} \sqrt{R^2 - x^2}$$

with $x \in [-R, R]$.

Parameters R : Real number, $R > 0$, the radius

Returns A *RandomSymbol*.

References

[R581] (page 1275), [R582] (page 1275)

Examples

```
>>> from diofant.stats import WignerSemicircle, density, E
>>> from diofant import Symbol, simplify
```

```
>>> R = Symbol("R", positive=True)
>>> z = Symbol("z")
```

```
>>> X = WignerSemicircle("x", R)
```

```
>>> density(X)(z)
2*sqrt(R**2 - z**2)/(pi*R**2)
```

```
>>> E(X)
0
```

`diofant.stats.ContinuousRV`(*symbol*, *density*, *set=Interval(-oo, oo, true, true)*)

Create a Continuous Random Variable given the following:

- a symbol - a probability density function - set on which the pdf is valid (defaults to entire real line)

Returns a RandomSymbol.

Many common continuous random variable types are already implemented. This function should be necessary only very rarely.

Examples

```
>>> from diofant import Symbol, sqrt, exp, pi
>>> from diofant.stats import ContinuousRV, P, E
```

```
>>> x = Symbol("x")
```

```
>>> pdf = sqrt(2)*exp(-x**2/2)/(2*sqrt(pi)) # Normal distribution
>>> X = ContinuousRV(x, pdf)
```

```
>>> E(X)
0
>>> P(X>0)
1/2
```

3.22.3 Interface

`diofant.stats.P`(*condition*, *given_condition=None*, *numsamples=None*, *evaluate=True*, ***kwargs*)

Probability that a condition is true, optionally given a second condition

Parameters **condition** : Combination of Relationals containing RandomSymbols

The condition of which you want to compute the probability

given_condition : Combination of Relationals containing RandomSymbols

A conditional expression. $P(X > 1, X > 0)$ is expectation of $X > 1$ given $X > 0$

numsamples : int

Enables sampling and approximates the probability with this many samples

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from diofant.stats import P, Die
>>> from diofant import Eq
>>> X, Y = Die('X', 6), Die('Y', 6)
>>> P(X > 3)
1/2
>>> P(Eq(X, 5), X > 2) # Probability that X == 5 given that X > 2
1/4
>>> P(X > Y)
5/12
```

`diofant.stats.E(expr, condition=None, numsamples=None, evaluate=True, **kwargs)`

Returns the expected value of a random expression

Parameters `expr` : Expr containing RandomSymbols

The expression of which you want to compute the expectation value

given : Expr containing RandomSymbols

A conditional expression. $E(X, X>0)$ is expectation of X given $X > 0$

numsamples : int

Enables sampling and approximates the expectation with this many samples

evalf : Bool (defaults to True)

If sampling return a number rather than a complex expression

evaluate : Bool (defaults to True)

In case of continuous systems return unevaluated integral

Examples

```
>>> from diofant.stats import E, Die
>>> X = Die('X', 6)
>>> E(X)
7/2
>>> E(2*X + 1)
8
```

```
>>> E(X, X > 3) # Expectation of X given that it is above 3
5
```

`diofant.stats.density(expr, condition=None, evaluate=True, numsamples=None, **kwargs)`

Probability density of a random expression, optionally given a second condition.

This density will take on different forms for different types of probability spaces. Discrete variables produce Dicts. Continuous variables produce Lambdas.

Parameters `expr` : Expr containing RandomSymbols

The expression of which you want to compute the density value

condition : Relational containing RandomSymbols

A conditional expression. `density(X > 1, X > 0)` is density of $X > 1$ given $X > 0$

numsamples : int

Enables sampling and approximates the density with this many samples

Examples

```
>>> from diofant.stats import density, Die, Normal
>>> from diofant import Symbol
```

```
>>> x = Symbol('x')
>>> D = Die('D', 6)
>>> X = Normal(x, 0, 1)
```

```
>>> density(D).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
>>> density(2*D).dict
{2: 1/6, 4: 1/6, 6: 1/6, 8: 1/6, 10: 1/6, 12: 1/6}
>>> density(X)(x)
sqrt(2)*E**(-x**2/2)/(2*sqrt(pi))
```

`diofant.stats.given(expr, condition=None, **kwargs)`

Conditional Random Expression From a random expression and a condition on that expression creates a new probability space from the condition and returns the same expression on that conditional probability space.

Examples

```
>>> from diofant.stats import given, density, Die
>>> X = Die('X', 6)
>>> Y = given(X, X > 3)
>>> density(Y).dict
{4: 1/3, 5: 1/3, 6: 1/3}
```

Following convention, if the condition is a random symbol then that symbol is considered fixed.

```
>>> from diofant.stats import Normal
>>> from diofant import pprint
>>> from diofant.abc import z
```

```
>>> X = Normal('X', 0, 1)
>>> Y = Normal('Y', 0, 1)
>>> pprint(density(X + Y, Y)(z), use_unicode=False)
      2
    -(-Y + z)
    -----
      2
```


$$\frac{\sqrt{2} * E}{2 * \sqrt{\pi}}$$

`diofant.stats.where(condition, given_condition=None, **kwargs)`

Returns the domain where a condition is True.

Examples

```
>>> from diofant.stats import where, Die, Normal
>>> from diofant import symbols, And
```

```
>>> D1, D2 = Die('a', 6), Die('b', 6)
>>> a, b = D1.symbol, D2.symbol
>>> X = Normal('x', 0, 1)
```

```
>>> where(X**2<1)
Domain: And(-1 < x, x < 1)
```

```
>>> where(X**2<1).set
(-1, 1)
```

```
>>> where(And(D1<=D2 , D2<3))
Domain: Or(And(Eq(a, 1), Eq(b, 1)), And(Eq(a, 1), Eq(b, 2)), And(Eq(a, 2), Eq(b, 2)))
```

`diofant.stats.variance(X, condition=None, **kwargs)`

Variance of a random expression

Expectation of $(X-E(X))^2$

Examples

```
>>> from diofant.stats import Die, E, Bernoulli, variance
>>> from diofant import simplify, Symbol
```

```
>>> X = Die('X', 6)
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> variance(2*X)
35/3
```

```
>>> simplify(variance(B))
p*(-p + 1)
```

`diofant.stats.std(X, condition=None, **kwargs)`

Standard Deviation of a random expression

Square root of the Expectation of $(X-E(X))^2$

Examples

```
>>> from diofant.stats import Bernoulli, std
>>> from diofant import Symbol, simplify
```

```
>>> p = Symbol('p')
>>> B = Bernoulli('B', p, 1, 0)
```

```
>>> simplify(std(B))
sqrt(p*(-p + 1))
```

`diofant.stats.sample(expr, condition=None, **kwargs)`
A realization of the random expression

Examples

```
>>> from diofant.stats import Die, sample
>>> X, Y, Z = Die('X', 6), Die('Y', 6), Die('Z', 6)
```

```
>>> die_roll = sample(X + Y + Z) # A random realization of three dice
```

`diofant.stats.sample_iter(expr, condition=None, numsamples=oo, **kwargs)`
Returns an iterator of realizations from the expression given a condition

`expr`: Random expression to be realized
`condition`: A conditional expression (optional)
`numsamples`: Length of the iterator (defaults to infinity)

See also:

[diofant.stats.sample](#) (page 990), [diofant.stats.rv.sampling_P](#) (page 992), [diofant.stats.rv.sampling_E](#) (page 992), [diofant.stats.rv.sample_iter_lambdify](#) (page 992), [diofant.stats.rv.sample_iter_subs](#) (page 992)

Examples

```
>>> from diofant.stats import Normal, sample_iter
>>> X = Normal('X', 0, 1)
>>> expr = X*X + 3
>>> iterator = sample_iter(expr, numsamples=3)
>>> list(iterator)
[12, 4, 7]
```

3.22.4 Mechanics

Diofant Stats employs a relatively complex class hierarchy.

RandomDomains are a mapping of variables to possible values. For example we might say that the symbol `Symbol('x')` can take on the values `{1, 2, 3, 4, 5, 6}`.

```
class diofant.stats.rv.RandomDomain
```

A PSpace, or Probability Space, combines a RandomDomain with a density to provide probabilistic information. For example the above domain could be enhanced by a finite density $\{1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6\}$ to fully define the roll of a fair die named x.

class diofant.stats.rv.PSpace

A RandomSymbol represents the PSpace's symbol 'x' inside of Diofant expressions.

class diofant.stats.rv.RandomSymbol

The RandomDomain and PSpace classes are almost never directly instantiated. Instead they are subclassed for a variety of situations.

RandomDomains and PSpaces must be sufficiently general to represent domains and spaces of several variables with arbitrarily complex densities. This generality is often unnecessary. Instead we often build SingleDomains and SinglePSpaces to represent single, univariate events and processes such as a single die or a single normal variable.

class diofant.stats.rv.SinglePSpace

class diofant.stats.rv.SingleDomain

Another common case is to collect together a set of such univariate random variables. A collection of independent SinglePSpaces or SingleDomains can be brought together to form a ProductDomain or ProductPSpace. These objects would be useful in representing three dice rolled together for example.

class diofant.stats.rv.ProductDomain

class diofant.stats.rv.ProductPSpace

The Conditional adjective is added whenever we add a global condition to a RandomDomain or PSpace. A common example would be three independent dice where we know their sum to be greater than 12.

class diofant.stats.rv.ConditionalDomain

We specialize further into Finite and Continuous versions of these classes to represent finite (such as dice) and continuous (such as normals) random variables.

class diofant.stats.frv.FiniteDomain

class diofant.stats.frv.FinitePSpace

class diofant.stats.crv.ContinuousDomain

class diofant.stats.crv.ContinuousPSpace

Additionally there are a few specialized classes that implement certain common random variable types. There is for example a DiePSpace that implements SingleFinitePSpace and a NormalPSpace that implements SingleContinuousPSpace.

class diofant.stats.frv_types.DiePSpace

class diofant.stats.crv_types.NormalPSpace

RandomVariables can be extracted from these objects using the PSpace.values method.

As previously mentioned Diofant Stats employs a relatively complex class structure. Inheritance is widely used in the implementation of end-level classes. This tactic was chosen to balance between the need to allow Diofant to represent arbitrarily defined random variables and optimizing for common cases. This complicates the code but is structured to only be important to those working on extending Diofant Stats to other random variable types.

Users will not use this class structure. Instead these mechanics are exposed through variable creation functions Die, Coin, FiniteRV, Normal, Exponential, etc.... These build the appropriate SinglePSpaces and return the corresponding RandomVariable. Conditional and Product

spaces are formed in the natural construction of Diofant expressions and the use of interface functions `E`, `Given`, `Density`, etc....

`diofant.stats.Die()`

`diofant.stats.Normal()`

There are some additional functions that may be useful. They are largely used internally.

`diofant.stats.rv.random_symbols(expr)`

Returns all `RandomSymbols` within a Diofant Expression.

`diofant.stats.rv.pspace(expr)`

Returns the underlying `Probability Space` of a random expression.

For internal use.

Examples

```
>>> from diofant.stats import pspace, Normal
>>> from diofant.stats.rv import ProductPSpace
>>> X = Normal('X', 0, 1)
>>> pspace(2*X + 1) == X.pspace
True
```

`diofant.stats.rv.rs_swap(a, b)`

Build a dictionary to swap `RandomSymbols` based on their underlying symbol.

i.e. if $X = ('x', \text{pspace1})$ and $Y = ('x', \text{pspace2})$ then X and Y match and the key, value pair $\{X:Y\}$ will appear in the result

Inputs: collections a and b of random variables which share common symbols Output: dict mapping RVs in a to RVs in b

`diofant.stats.rv.sampling_P(condition, given_condition=None, numsamples=1, evalf=True, **kwargs)`

Sampling version of `P`

See also:

[diofant.stats.P](#) (page 986), [diofant.stats.rv.sampling_E](#) (page 992), [diofant.stats.rv.sampling_density](#) (page 992)

`diofant.stats.rv.sampling_E(expr, given_condition=None, numsamples=1, evalf=True, **kwargs)`

Sampling version of `E`

See also:

[diofant.stats.P](#) (page 986), [diofant.stats.rv.sampling_P](#) (page 992), [diofant.stats.rv.sampling_density](#) (page 992)

`diofant.stats.rv.sample_iter_lambdify(expr, condition=None, numsamples=oo, **kwargs)`

See `sample_iter`

Uses `lambdify` for computation. This is fast but does not always work.

`diofant.stats.rv.sample_iter_subs(expr, condition=None, numsamples=oo, **kwargs)`

See `sample_iter`

Uses `subs` for computation. This is slow but almost always works.

`diofant.stats.rv.sampling_density`(*expr*, *given_condition=None*, *numsamples=1*,
***kwargs*)

Sampling version of density

See also:

[`diofant.stats.density`](#) (page 987), [`diofant.stats.rv.sampling_P`](#) (page 992), [`diofant.stats.rv.sampling_E`](#) (page 992)

`diofant.stats.rv.independent`(*a*, *b*)

Independence of two random expressions

Two expressions are independent if knowledge of one does not change computations on the other.

See also:

[`diofant.stats.rv.dependent`](#) (page 993)

Examples

```
>>> from diofant.stats import Normal, independent, given
>>> from diofant import Tuple, Eq
```

```
>>> X, Y = Normal('X', 0, 1), Normal('Y', 0, 1)
>>> independent(X, Y)
True
>>> independent(2*X + Y, -Y)
False
>>> X, Y = given(Tuple(X, Y), Eq(X + Y, 3))
>>> independent(X, Y)
False
```

`diofant.stats.rv.dependent`(*a*, *b*)

Dependence of two random expressions

Two expressions are independent if knowledge of one does not change computations on the other.

See also:

[`diofant.stats.rv.independent`](#) (page 993)

Examples

```
>>> from diofant.stats import Normal, dependent, given
>>> from diofant import Tuple, Eq
```

```
>>> X, Y = Normal('X', 0, 1), Normal('Y', 0, 1)
>>> dependent(X, Y)
False
>>> dependent(2*X + Y, -Y)
True
>>> X, Y = given(Tuple(X, Y), Eq(X + Y, 3))
>>> dependent(X, Y)
True
```

3.23 ODE

3.23.1 User Functions

These are functions that are imported into the global namespace with `from diofant import *`. These functions (unlike *Hint Functions* (page 1001), 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 1041) docstring for supported methods). This can either be an *Equality* (page 112), 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 997). See Hints below for more options that you can use for hint.

simplify enables simplification by *odesimp()* (page 1002). 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 1000) 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.

x_0 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 994):

default: This uses whatever hint is returned first by `classify_ode()` (page 997). This is the default argument to `dsolve()` (page 994).

all: To make `dsolve()` (page 994) 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 1067) in `deu-tils.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 997) 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 997).

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 994) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than `all`, because `integrate()` (page 72) is an expensive routine.

best: To have `dsolve()` (page 994) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function, whether it contains any `Integral` classes (i.e. unevaluatable integrals), and which one is the shortest in size.

See also the `classify_ode()` (page 997) docstring for more info on hints, and the `ode` (page 1041) docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from diofant import Function, Derivative
>>> from diofant.abc import x # x is the independent variable
>>> f = Function("f")(x) # f is a function of x
>>> # f_ will be the derivative of f with respect to x
>>> f_ = Derivative(f, x)
```

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use `dsolve()` (page 994).
- `dsolve()` (page 994) always returns an `Equality` (page 112) 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 112), 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

```
>>> from diofant import Function, dsolve, Eq, Derivative, sin, cos, symbols, exp,
↳ E
>>> from diofant.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
```

```
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='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)))]
>>> t = symbols('t')
>>> x, y = symbols('x, y', cls=Function)
>>> eq = (Eq(Derivative(x(t),t), 12*t*x(t) + 8*y(t)), Eq(Derivative(y(t),t),
↳ 21*x(t) + 7*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0(t) + C2*x0(t)*Integral(8**Integral(7*t, t)*E**Integral(12*t, t)/
↳ x0(t)**2, t)),
Eq(y(t), C1*y0(t) + C2*(E**Integral(7*t, t)*E**Integral(12*t, t)/x0(t) +
y0(t)*Integral(8**Integral(7*t, t)*E**Integral(12*t, t)/x0(t)**2, t))]
>>> eq = (Eq(Derivative(x(t),t),x(t)*y(t)*sin(t)), Eq(Derivative(y(t),t),
↳ y(t)**2*sin(t)))
```



```

>>> C1, C2 = symbols('C1, C2')
>>> dsolve(eq) == {Eq(x(t), -E**C1/(C2*E**C1 - cos(t))), Eq(y(t), -1/(C1 -
↳cos(t)))}
True

```

classify_ode

`diofant.solvers.ode.classify_ode(eq, func=None, dict=False, init=None, **kwargs)`

Returns a tuple of possible `dsolve()` (page 994) classifications for an ODE.

The tuple is ordered so that first item is the classification that `dsolve()` (page 994) 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 994) use a different classification, use `dsolve(ODE,func,hint=<classification>)`. See also the `dsolve()` (page 994) docstring for different meta-hints you can use.

If `dict` is true, `classify_ode()` (page 997) will return a dictionary of `hint:match` expression terms. This is intended for internal use by `dsolve()` (page 994). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hintname` is the name of the hint without `_Integral`.

See `allhints` (page 1002) or the `ode` (page 1041) docstring for a list of all supported hints that can be returned from `classify_ode()` (page 997).

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 553) class in it. Note that a hint may do this anyway if `integrate()` (page 546) 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 546) is an expensive routine. If `dsolve()` (page 994) hangs, it is probably because `integrate()` (page 72) 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 546) is not used in solving the ODE for those method. For example, `n`th order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no `nth_linear_homogeneous_constant_coeff_Integrate` hint. You can easily evaluate any unevaluated `Integral` (page 553)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 `n`th in it, such as the `nth_linear` hints, this means

that the method used to applies to ODEs of any order.

indep and dep

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of $f(x)$, then `indep` will refer to x and `dep` will refer to f .

subs

If a hints has the word `subs` in it, it means the the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.

coeff

The word `coeff` in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (`help(ode)`). This is contrast to `coefficients`, as in `undetermined_coefficients`, which refers to the common name of a method.

_best

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a `_best` meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal best meta-hint.

Examples

```
>>> from diofant import Function, classify_ode, Eq
>>> from diofant.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('separable', '1st_linear', '1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep',
'1st_power_series', 'lie_group',
'nth_linear_constant_coeff_homogeneous',
'separable_Integral', '1st_linear_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

checkodesol

`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 112) that the solution satisfies, e.g.

$\text{Eq}(f(x), C1), \text{Eq}(f(x) + C1, 0)$; or simply an *Expr* (page 59), 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 922) cannot reduce the expression to 0 . If an expression returned by this function vanishes identically, then `sol` really is a solution to `ode`.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

Examples

```
>>> from diofant import Eq, Function, checkodesol, symbols
>>> x, C1 = symbols('x,C1')
>>> f = Function('f')
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

homogeneous_order

`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 1007) and `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1008)).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the list of symbols. If a declared function appears with different arguments than given in the list of symbols, None is returned.

Examples

```
>>> from diofant import Function, homogeneous_order, sqrt
>>> from diofant.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
>>> homogeneous_order(f(x), f(x), x)
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

infinitesimals

`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 $\varepsilon = 0$.

$$\left(\frac{\partial X(x, y; \varepsilon)}{\partial \varepsilon}\right) \Big|_{\varepsilon=0} = \xi, \left(\frac{\partial Y(x, y; \varepsilon)}{\partial \varepsilon}\right) \Big|_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> from diofant import Function, diff, Eq, pprint
>>> from diofant.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y) # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h
... - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
>>> pprint(genform, use_unicode=False)
- eta(x, y)*--(h(x, y)) - h(x, y)*--(xi(x, y)) + h(x, y)*|--(eta(x, y)) - --(
      dy                dy                \dy                dx
      \                d                d
```

$$\xi(x, y) \frac{\partial}{\partial x} - \xi(x, y) \frac{\partial}{\partial x} (h(x, y)) + \eta(x, y) \frac{\partial}{\partial x} = 0$$

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`.

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

Examples

```
>>> from diofant import Function, diff, exp
>>> from diofant.solvers.ode import infinitesimals
>>> from diofant.abc import x
>>> f = Function('f')
>>> eta = Function('eta')
>>> xi = Function('xi')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq) == [{eta(x, f(x)): exp(x**3/3), xi(x, f(x)): 0}]
True
```

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.

3.23.2 Hint Functions

These functions are intended for internal use by `dsolve()` (page 994) and others. Unlike *User Functions* (page 994), above, these are not intended for every-day use by ordinary Diofant users. Instead, functions such as `dsolve()` (page 994) 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_specia`

This is a list of hints in the order that they should be preferred by `classify_ode()` (page 997). 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 994) 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 1003).

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 553)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by `dsolve()` (page 994), as `dsolve()` (page 994) already calls `odesimp()` (page 1002), but the individual hint functions do not call `odesimp()` (page 1002) (because the `dsolve()` (page 994) wrapper does). Therefore, this function is designed for mainly internal use.

Examples

```
>>> from diofant import sin, symbols, dsolve, pprint, Function
>>> from diofant.solvers.ode import odesimp
>>> x, u2, C1 = symbols('x,u2,C1')
>>> f = Function('f')
```

```
>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_dep_Integral',
... simplify=False)
>>> pprint(eq, wrap_line=False, use_unicode=False)
```

$$\frac{x}{f(x)} \sqrt{-u^2 + 1}$$

$$\log(f(x)) = \log(C1) + \frac{\sin\left(\frac{\sqrt{1-u^2}}{u^2}\right)}{u^2} d(u^2)$$

```
>>> pprint(odesimp(eq, f(x), 1, {C1},
...           hint='1st_homogeneous_coeff_subs_indep_div_dep'),
...         use_unicode=False)
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 renumbers any *Symbol* (page 81) 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 1003).

Examples

```
>>> from diofant import symbols, Eq, pprint
>>> from diofant.solvers.ode import constant_renumber
>>> x, C0, C1, C2, C3, C4 = symbols('x,C:5')
```

Only constants in the given range (inclusive) are renumbered; the renumbering always starts from 1:

```
>>> constant_renumber(C1 + C3 + C4, 'C', 1, 3)
C1 + C2 + C4
>>> constant_renumber(C0 + C1 + C3 + C4, 'C', 2, 4)
C0 + 2*C1 + C2
>>> constant_renumber(C0 + 2*C1 + C2, 'C', 0, 1)
C1 + 3*C2
>>> pprint(C2 + C1*x + C3*x**2, use_unicode=False)
          2
C1*x + C2 + C3*x
>>> pprint(constant_renumber(C2 + C1*x + C3*x**2, 'C', 1, 3), use_unicode=False)
          2
C1 + C2*x + C3*x
```

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 994), 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 107)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 107)s are expanded so e^{C_1+x} will be simplified to $C_1 e^x$.

Use `constant_renumber()` (page 1003) 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 994).

Examples

```
>>> from diofant import symbols
>>> from diofant.solvers.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

sol_simplicity

`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 553)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 553)	<code>oo</code>

`oo` here means the Diofant infinity, which should compare greater than any integer.

If you already know `solve()` (page 1054) cannot solve `sol`, you can use `trysolving=False` to skip that step, which is the only potentially slow step. For example, `dsolve()` (page 994) with the `simplify=False` flag should do this.

If `sol` is a list of solutions, if the worst solution in the list returns `oo` it returns that, otherwise it returns `len(str(sol))`, that is, the length of the string representation of the whole list.

Examples

This function is designed to be passed to `min` as the key argument, such as `min(listofsolutions, key=lambda i: ode_sol_simplicity(i, f(x)))`.

```
>>> from diofant import symbols, Function, Eq, tan, cos, sqrt, Integral
>>> from diofant.solvers.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')
```

```
>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

1st_exact

`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:

```
>>> from diofant import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1 = symbols('x,y,t,x0,y0,C1')
>>> P, Q, F = map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(x0, t), (t, y0, y))), C1), use_unicode=False)
```

$$F(x, y) = \int_{x_0}^x P(t, y) dt + \int_{y_0}^y Q(x_0, t) dt = C_1$$

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.

References

[R492] (page 1275), [R493] (page 1275)

Examples

```
>>> from diofant import Function, dsolve, cos, sin
>>> from diofant.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

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 1004).

See the `ode_1st_homogeneous_coeff_subs_indep_div_dep()` (page 1008) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1007) docstrings for more information on these hints. Note that there is no `ode_1st_homogeneous_coeff_best_Integral` hint.

References

[R494] (page 1275), [R495] (page 1275)

Examples

```
>>> from diofant import Function, dsolve, pprint
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...             hint='1st_homogeneous_coeff_best', simplify=False),
...         use_unicode=False)
          /  2  \
          | 3*x  |
      log|----- + 1|
          | 2    |
          \| f (x) \|
log(f(x)) = log(C1) - -----
                        3
```

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 999).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $y = u_1 x$ (i.e. $u_1 = y/x$) will turn the differential equation into an equation separable in the variables x and u . If $h(u_1)$ is the function that results from making the substitution $u_1 = f(x)/x$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from diofant import Function, dsolve, pprint
>>> from diofant.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform, use_unicode=False)
```

```

/f(x)\    /f(x)\ d
g|----| + h|----|*--(f(x))
 \ x /    \ x / dx
>>> pprint(dsolve(genform, f(x),
...             hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'),
...         use_unicode=False)
...
          f(x)
          ----
          x
          /
          |
log(x) = C1 + |          -h(u1)
          |          ----- d(u1)
          |          u1*h(u1) + g(u1)
          |
          /

```

Where $u_1h(u_1) + g(u_1) \neq 0$ and $x \neq 0$.

See also:

[diofant.solvers.ode.ode_1st_homogeneous_coeff_best](#) (page 1006), [diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep](#) (page 1008)

References

[R496] (page 1275), [R497] (page 1275)

Examples

```

>>> from diofant import Function, dsolve
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...             hint='1st_homogeneous_coeff_subs_dep_div_indep',
...             simplify=False), use_unicode=False)
...
          /          3 \
          |3*f(x)  f (x)|
log|----- + -----|
  | x          3 |
  \          x /
log(x) = log(C1) - -----
                    3

```

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 $u_2 = \frac{\langle \text{independent variable} \rangle}{\langle \text{dependent variable} \rangle}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of `homogeneous_order()` (page 999).

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 $x = u_2y$ (i.e. $u_2 = x/y$) will turn the differential equation into an equation separable in the variables y and u_2 . If $h(u_2)$ is the function that results from making the substitution $u_2 = x/f(x)$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from diofant import Function, dsolve, pprint
>>> from diofant.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform, use_unicode=False)
 / x \   / x \ d
g|----| + h|----|*--(f(x))
 \f(x)/  \f(x)/ dx
>>> pprint(dsolve(genform, f(x),
...             hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'),
...       use_unicode=False)
      x
      ----
      f(x)
      /
      |
      |          -g(u2)
      |          ----- d(u2)
      |          u2*g(u2) + h(u2)
      |
      /
f(x) = E                                     *C1
```

Where $u_2g(u_2) + h(u_2) \neq 0$ and $f(x) \neq 0$.

See also the docstrings of `ode_1st_homogeneous_coeff_best()` (page 1006) and `ode_1st_homogeneous_coeff_subs_dep_div_indep()` (page 1007).

References

[R498] (page 1275), [R499] (page 1275)

Examples

```
>>> from diofant import Function, pprint, dsolve
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
...             hint='1st_homogeneous_coeff_subs_indep_div_dep',
```


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 1010)). The general solution is:

```
>>> from diofant import Function, dsolve, Eq, pprint
>>> from diofant.abc import x, n
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform, use_unicode=False)
P(x)*f(x) + --(f(x)) = Q(x)*f (x)
          dx
>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'),
...         use_unicode=False, wrap_line=False)
```

$$f(x) = E^{-\int P(x) dx} \left(\int (-n + 1) P(x) dx + C_1 + (n - 1) \int (-E^{-\int P(x) dx}) Q(x) dx \right)^{\frac{1}{-n}}$$

Note that the equation is separable when $n = 1$ (see the docstring of `ode_separable()` (page 1017)).

```
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
...               hint='separable_Integral'), use_unicode=False)
```

$$\frac{1}{y} dy = C_1 + \int (-P(x) + Q(x)) dx$$

References

[R502] (page 1275), [R503] (page 1275)

nth_linear_constant_coeff_homogeneous

`diofant.solvers.ode.ode_nth_linear_constant_coeff_homogeneous`(*eq*, *func*,
order,
match, *re-*
turns='sol')

Solves an n th order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \dots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation $a_n m^n + a_{n-1} m^{n-1} + \dots + a_1 m + a_0 = 0$. 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 734) instance will be return instead.

```
>>> from diofant import Function, dsolve, Eq
>>> from diofant.abc import x
>>> f = Function('f')
>>> dsolve(f(x).diff(x, 5) + 10*f(x).diff(x) - 2*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous')
...
Eq(f(x), 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>}`.

References

[R506] (page 1276), [R507] (page 1276)

Examples

```
>>> from diofant import Function, dsolve, pprint
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) -
...             2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
...             hint='nth_linear_constant_coeff_homogeneous'),
...         use_unicode=False)
      x          -2*x
f(x) = E *(C3 + C4*x) + E *(C1*sin(x) + C2*cos(x))
```

nth_linear_constant_coeff_undetermined_coefficients

`diofant.solvers.ode.ode_nth_linear_constant_coeff_undetermined_coefficients`(*eq*, *func*, *or_order*, *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

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \dots + a_1 f'(x) + a_0 f(x) = P(x),$$

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.

References

[R508] (page 1276), [R509] (page 1276)

Examples

```

>>> from diofant import Function, dsolve, pprint, exp, cos
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) -
...             4*exp(-x)*x**2 + cos(2*x), f(x),
...             hint='nth_linear_constant_coeff_undetermined_coefficients'),
...             use_unicode=False)

```

$$f(x) = -\frac{4\sin(2x)}{25} + \frac{3\cos(2x)}{25} + E^{-x} \sqrt{C1 + C2x} + \frac{4x}{3}$$

nth_linear_constant_coeff_variation_of_parameters

`diofant.solvers.ode.ode_nth_linear_constant_coeff_variation_of_parameters`(*eq*, *func*, *order*, *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

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \dots + a_1f'(x) + a_0f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{x=1}^n c_i(x)y_i(x),$$

where *y_i* is the *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

$$\sum_{x=1}^n \left(\int \frac{W_i(x)}{W(x)} dx \right) y_i(x),$$

where *W(x)* is the Wronskian of the fundamental system (the system of *n* linearly independent solutions to the homogeneous equation), and *W_i(x)* is the Wronskian of the fundamental system with the *i*th column replaced with [0, 0, ..., 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 994) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use `simplify=False` with `'nth_linear_constant_coeff_variation_of_parameters_Integral'` for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

References

[R510] (page 1276), [R511] (page 1276), [R512] (page 1276)

Examples

```
>>> from diofant import Function, dsolve, pprint, exp, log
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) +
...             3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
...             hint='nth_linear_constant_coeff_variation_of_parameters'),
...             use_unicode=False)
      /
      x |
f(x) = E * | C1 + C2*x + C3*x2 + ----- |
      \              3             \
              x *(6*log(x) - 11) |
              36                   /
```

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 924) 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 924) is smart enough to do most expansion and factoring necessary to convert a separable equation $F(x, y)$ into the proper form $P(x) \cdot Q(y)$. The general solution is:

```
>>> from diofant import Function, dsolve, Eq, pprint
>>> from diofant.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform, use_unicode=False)
      d
a(x)*b(f(x))*--(f(x)) = c(x)*d(f(x))
      dx
>>> pprint(dsolve(genform, f(x),
...             hint='separable_Integral'), use_unicode=False)
      f(x)
      /
      |
      | b(y)
      | ---- dy = C1 +
      | d(y)
      |
      /
      |
      | c(x)
      | ---- dx
      | a(x)
      |
      /
```

References

[R513] (page 1276)

Examples

```
>>> from diofant import Function, dsolve, Eq
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x) + x, 3*x*f(x)**2), f(x),
...             hint='separable', simplify=False), use_unicode=False)
      / 2      \      2
log\3*f (x) - 1/      x
----- = C1 + --
      6              2
```

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

```
>>> from diofant import Function, dsolve, Eq, pprint
>>> from diofant.abc import x, y, n
>>> f, g, k, l = map(Function, ['f', 'g', 'k', 'l'])
>>> genform = Eq(f(x)*(l(y).diff(y)) + k(x)*l(y) + g(x))
>>> pprint(genform, use_unicode=False)
      d
f(x)*--(l(y)) + g(x) + k(x)*l(y) = 0
      dy
>>> pprint(dsolve(genform, hint = 'almost_linear'), use_unicode=False)
      /      //      -y*g(x)      \
      |      ||      -----      | for k(x) = 0 |
      |      ||      f(x)          |
      |      ||      y*k(x)        |
      |      ||      -----      |
      |      ||      f(x)          |
      |      ||      -E            | *g(x)
      |      ||      -----      | otherwise
      |      ||      k(x)          |
      \      \\\
      \      \\\
```

See also:

[diofant.solvers.ode.ode_1st_linear\(\)](#) (page 1010)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```

>>> from diofant import Function, Derivative, pprint
>>> from diofant.solvers.ode import dsolve, classify_ode
>>> from diofant.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), E**(-x)*(C1 - Ei(x)))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'), use_unicode=False)
      -x
f(x) = E  *(C1 - Ei(x))

```

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 1006), [`diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_indep_div_dep\(\)`](#) (page 1008), [`diofant.solvers.ode.ode_1st_homogeneous_coeff_subs_dep_div_indep\(\)`](#) (page 1007)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```

>>> from diofant import Function, Derivative, pprint
>>> from diofant.solvers.ode import dsolve, classify_ode
>>> from diofant.abc import x
>>> f = Function('f')
>>> df = f(x).diff(x)

```

```
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[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'), use_unicode=False)

[f(x) = -x - \sqrt{C1 + 7*x^2} - 1, f(x) = -x + \sqrt{C1 + 7*x^2} - 1]
```

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:

```
>>> from diofant import Function, dsolve, Eq, pprint
>>> from diofant.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform, use_unicode=False)

      / n      \
d      f(x)*g\ x *f(x)/
--(f(x)) + -----
dx      x
>>> pprint(dsolve(genform, hint='separable_reduced'), use_unicode=False)

n
x *f(x)
/
|
|      1
|----- dy = C1 + log(x)
| y*(n - g(y))
|
/
```

See also:

[diofant.solvers.ode.ode_separable\(\)](#) (page 1017)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

Examples

```

>>> from diofant import Function, Derivative, pprint
>>> from diofant.solvers.ode import dsolve, classify_ode
>>> from diofant.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (-sqrt(C1*x**2 + 1) + 1)/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/x)]
>>> pprint(dsolve(eq, hint='separable_reduced'), use_unicode=False)

```

$$[f(x) = \frac{-\sqrt{C1*x^2 + 1} + 1}{x}, f(x) = \frac{\sqrt{C1*x^2 + 1} + 1}{x}]$$

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 1000) function which returns the infinitesimals of the transformation.

The coordinates r and s can be found by solving the following Partial Differential Equations.

$$\xi \frac{\partial r}{\partial x} + \eta \frac{\partial r}{\partial y} = 0$$

$$\xi \frac{\partial s}{\partial x} + \eta \frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x, y) \frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x, y) \frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting r and s in terms of x and y again.

References

[R514] (page 1276)

Examples

```

>>> from diofant import Function, dsolve, exp, pprint
>>> from diofant.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
...             hint='lie_group'), use_unicode=False)
      2 /      2\
      -x |      x |
f(x) = E  *|C1 + --|
          \      2 /

```

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. $F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1$

References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

Examples

```

>>> from diofant import Function, Derivative, pprint, exp
>>> from diofant.solvers.ode import dsolve
>>> from diofant.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'), use_unicode=False)
      3      4      5
      C1*x  C1*x  C1*x  / 6\
f(x) = C1 + C1*x - ---- + ---- + ---- + 0\ x /
          6      24      60

```

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

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x) = 0$$

For simplicity it is assumed that $P(x)$, $Q(x)$ and $R(x)$ are polynomials, it is sufficient that $\frac{Q(x)}{P(x)}$ and $\frac{R(x)}{P(x)}$ exists at x_0 . A recurrence relation is obtained by substituting y as $\sum_{n=0}^{\infty} a_n x^n$, in the differential equation, and equating the n th term. Using this relation various terms can be generated.

References

[R515] (page 1276), [R516] (page 1276)

Examples

```
>>> from diofant import dsolve, Function, pprint
>>> from diofant.abc import x, y
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'), use_unicode=False)
```

$$f(x) = C2 * \frac{x^4}{24} - \frac{x^2}{2} + 1 + C1 * x * \frac{x^2}{6} - \frac{x}{6} + 1 + 0 \frac{x^6}{6}$$

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

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x) = 0$$

A point is said to regular singular at x_0 if $x - x_0 \frac{Q(x)}{P(x)}$ and $(x - x_0)^2 \frac{R(x)}{P(x)}$ are analytic at x_0 . For simplicity $P(x)$, $Q(x)$ and $R(x)$ are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing $(x - x_0)P(x)$ and $((x - x_0)^2)Q(x)$ as power series solutions about x_0 . Find p_0 and q_0 which are the constants of the power series expansions.
2. Solve the indicial equation $f(m) = m(m - 1) + m * p_0 + q_0$, to obtain the roots m_1 and m_2 of the indicial equation.
3. If $m_1 - m_2$ is a non integer there exists two series solutions. If $m_1 = m_2$, there exists only one solution. If $m_1 - m_2$ is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form $x^m \sum_{n=0}^{\infty} a_n x^n$. The coefficients are determined by the following recurrence relation. $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$. For the case in which $m_1 - m_2$ is an integer, it can be seen from the recurrence relation that for the lower root

m , when n equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

References

- George E. Simmons, “Differential Equations with Applications and Historical Notes”, p.p 176 - 184

Examples

```
>>> from diofant import dsolve, Function, pprint
>>> from diofant.abc import x, y
>>> f = Function("f")
>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq), use_unicode=False)
```

$$f(x) = C2 \sqrt[4]{x} \sqrt[2]{x} + 1 + \frac{C1 \sqrt[6]{x} \sqrt[4]{x} \sqrt[2]{x} + 1}{x} + 0 \sqrt[6]{x}$$

3.23.3 Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

abaco1_simple

`diofant.solvers.ode.lie_heuristic_abaco1_simple(match, comp=False)`

The first heuristic uses the following four sets of assumptions on ξ and η

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and η to be a function of x , the PDE

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x} \right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi * \frac{\partial h}{\partial x} - \eta * \frac{\partial h}{\partial y} = 0$$

reduces to $f'(x) - f \frac{\partial h}{\partial y} = 0$ If $\frac{\partial h}{\partial y}$ is a function of x , then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A.C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abacol_product

`diofant.solvers.ode.lie_heuristic_abacol_product(match, comp=False)`

The second heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if $\frac{1}{h^2} \frac{\partial^2}{\partial x \partial y} \log(h)$ is separable in x and y , then the separated factors containing x is $f(x)$, and $g(y)$ is obtained by

$$e^{\int f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right) dy}$$

provided $f \frac{\partial}{\partial x} \left(\frac{1}{f * h} \right)$ is a function of y only.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) * g(y)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

`diofant.solvers.ode.lie_heuristic_bivariate(match, comp=False)`

The third heuristic assumes the infinitesimals ξ and η to be bi-variate polynomials in x and y . The assumption made here for the logic below is that h is a rational function in x and y though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329

chi

`diofant.solvers.ode.lie_heuristic_chi(match, comp=False)`

The aim of the fourth heuristic is to find the function $\chi(x, y)$ that satisfies the PDE $\frac{d\chi}{dx} + h\frac{d\chi}{dy} - \frac{\partial h}{\partial y}\chi = 0$.

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=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1.First $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial y^2}}$ is calculated. Let us say this value is A

2.If this is constant, then h is matched to the form $A(x) + B(x)e^{\frac{y}{C}}$ then, $\frac{e^{\int \frac{A(x)}{B(x)} dx}}{B(x)}$ 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=False)`

This heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y} \left[\left(h \frac{\partial^2}{\partial x^2} (h^{-1}) \right)^{-1} \right]$$

is separable in x and y ,

1. The separated factors containing y is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing x is $f''(x)$.
3. $h \frac{\partial^2}{\partial x^2} (h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y, x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

`diofant.solvers.ode.lie_heuristic_abaco2_unique_unknown(match, comp=False)`

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing x and y
2. Loop over each element f in the list, find $\frac{\partial f}{\partial y} = R$

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=False)`

This heuristic assumes

1. $\xi = ax + by + c$ and

2. $\eta = fx + gy + h$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that h need not be a rational function in this case.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

3.23.4 System of ODEs

These functions are intended for internal use by `dsolve()` (page 994) for system of differential equations.

`system_of_odes_linear_2eq_order1_type1`

`diofant.solvers.ode._linear_2eq_order1_type1(x, y, t, r, eq)`

It is classified under system of two linear homogeneous first-order constant-coefficient ordinary differential equations.

The equations which come under this type are

$$x' = ax + by,$$

$$y' = cx + dy$$

The characteristics equation is written as

$$\lambda^2 + (a + d)\lambda + ad - bc = 0$$

and its discriminant is $D = (a - d)^2 + 4bc$. There are several cases

1. Case when $ad - bc \neq 0$. The origin of coordinates, $x = y = 0$, is the only stationary point; it is - a node if $D = 0$ - a node if $D > 0$ and $ad - bc > 0$ - a saddle if $D > 0$ and $ad - bc < 0$ - a focus if $D < 0$ and $a + d \neq 0$ - a centre if $D < 0$ and $a + d = 0$.

1.1. If $D > 0$. The characteristic equation has two distinct real roots λ_1 and λ_2 . The general solution of the system in question is expressed as

$$x = C_1 b e^{\lambda_1 t} + C_2 b e^{\lambda_2 t}$$

$$y = C_1 (\lambda_1 - a) e^{\lambda_1 t} + C_2 (\lambda_2 - a) e^{\lambda_2 t}$$

where C_1 and C_2 being arbitrary constants

1.2. If $D < 0$. The characteristics equation has two conjugate roots, $\lambda_1 = \sigma + i\beta$ and $\lambda_2 = \sigma - i\beta$. The general solution of the system is given by

$$x = b e^{\sigma t} (C_1 \sin(\beta t) + C_2 \cos(\beta t))$$

$$y = e^{\sigma t} ([(\sigma - a)C_1 - \beta C_2] \sin(\beta t) + [\beta C_1 + (\sigma - a)C_2 \cos(\beta t)])$$

1.3. If $D = 0$ and $a \neq d$. The characteristic equation has two equal roots, $\lambda_1 = \lambda_2$. The general solution of the system is written as

$$x = 2b \left(C_1 + \frac{C_2}{a - d} + C_2 t \right) e^{\frac{a+d}{2} t}$$

$$y = [(d - a)C_1 + C_2 + (d - a)C_2 t] e^{\frac{a+d}{2} t}$$

1.4. If $D = 0$ and $a = d \neq 0$ and $b = 0$

$$x = C_1 e^{at}, y = (cC_1 t + C_2) e^{at}$$

1.5. If $D = 0$ and $a = d \neq 0$ and $c = 0$

$$x = (bC_1 t + C_2) e^{at}, y = C_1 e^{at}$$

2. Case when $ad - bc = 0$ and $a^2 + b^2 > 0$. The whole straight line $ax + by = 0$ consists of singular points. The original system of differential equations can be rewritten as

$$x' = ax + by, y' = k(ax + by)$$

2.1 If $a + bk \neq 0$, solution will be

$$x = bC_1 + C_2 e^{(a+bk)t}, y = -aC_1 + kC_2 e^{(a+bk)t}$$

2.2 If $a + bk = 0$, solution will be

$$x = C_1 (bkt - 1) + bC_2 t, y = k^2 bC_1 t + (bk^2 t + 1)C_2$$

system_of_odes_linear_2eq_order1_type2

diofant.solvers.ode._linear_2eq_order1_type2(x, y, t, r, eq)

The equations of this type are

$$x' = ax + by + k1, y' = cx + dy + k2$$

The general solution of this system is given by sum of its particular solution and the general solution of the corresponding homogeneous system is obtained from type1.

1. When $ad - bc \neq 0$. The particular solution will be $x = x_0$ and $y = y_0$ where x_0 and y_0 are determined by solving linear system of equations

$$ax_0 + by_0 + k1 = 0, cx_0 + dy_0 + k2 = 0$$

2. When $ad - bc = 0$ and $a^2 + b^2 > 0$. In this case, the system of equation becomes

$$x' = ax + by + k_1, y' = k(ax + by) + k_2$$

2.1 If $\sigma = a + bk \neq 0$, particular solution is given by

$$x = b\sigma^{-1}(c_1k - c_2)t - \sigma^{-2}(ac_1 + bc_2)$$

$$y = kx + (c_2 - c_1k)t$$

2.2 If $\sigma = a + bk = 0$, particular solution is given by

$$x = \frac{1}{2}b(c_2 - c_1k)t^2 + c_1t$$

$$y = kx + (c_2 - c_1k)t$$

system_of_odes_linear_2eq_order1_type3

`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

$$x = e^F(C_1e^G + C_2e^{-G}), y = e^F(C_1e^G - C_2e^{-G})$$

where C_1 and C_2 are arbitrary constants, and

$$F = \int f(t) dt, G = \int g(t) dt$$

system_of_odes_linear_2eq_order1_type4

`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

$$x = C_1x_0(t) + C_2x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt$$

$$y = C_1y_0(t) + C_2\left[\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} dt\right]$$

where C1 and C2 are arbitrary constants and

$$F(t) = e^{\int f(t) dt}, P(t) = e^{\int p(t) dt}$$

system_of_odes_linear_2eq_order2_type1

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

$$x = C_1be^{\lambda_1 t} + C_2be^{\lambda_2 t} + C_3be^{\lambda_3 t} + C_4be^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 - a)e^{\lambda_1 t} + C_2(\lambda_2^2 - a)e^{\lambda_2 t} + C_3(\lambda_3^2 - a)e^{\lambda_3 t} + C_4(\lambda_4^2 - a)e^{\lambda_4 t}$$

where C_1, \dots, C_4 are arbitrary constants.

1.2. If $D = 0$ and $a \neq d$:

$$x = 2C_1\left(bt + \frac{2bk}{a-d}\right)e^{\frac{kt}{2}} + 2C_2\left(bt + \frac{2bk}{a-d}\right)e^{-\frac{kt}{2}} + 2bC_3te^{\frac{kt}{2}} + 2bC_4te^{-\frac{kt}{2}}$$

$$y = C_1(d-a)te^{\frac{kt}{2}} + C_2(d-a)te^{-\frac{kt}{2}} + C_3[(d-a)t + 2k]e^{\frac{kt}{2}} + C_4[(d-a)t - 2k]e^{-\frac{kt}{2}}$$

where C_1, \dots, C_4 are arbitrary constants and $k = \sqrt{2(a+d)}$

1.3. If $D = 0$ and $a = d \neq 0$ and $b = 0$:

$$x = 2\sqrt{a}C_1e^{\sqrt{a}t} + 2\sqrt{a}C_2e^{-\sqrt{a}t}$$

$$y = cC_1te^{\sqrt{a}t} - cC_2te^{-\sqrt{a}t} + C_3e^{\sqrt{a}t} + C_4e^{-\sqrt{a}t}$$

1.4. If $D = 0$ and $a = d \neq 0$ and $c = 0$:

$$x = bC_1te^{\sqrt{at}} - bC_2te^{-\sqrt{at}} + C_3e^{\sqrt{at}} + C_4e^{-\sqrt{at}}$$

$$y = 2\sqrt{a}C_1e^{\sqrt{at}} + 2\sqrt{a}C_2e^{-\sqrt{at}}$$

2. When $ad - bc = 0$ and $a^2 + b^2 > 0$. Then the original system becomes

$$x'' = ax + by$$

$$y'' = k(ax + by)$$

2.1. If $a + bk \neq 0$:

$$x = C_1e^{t\sqrt{a+bk}} + C_2e^{-t\sqrt{a+bk}} + C_3bt + C_4b$$

$$y = C_1ke^{t\sqrt{a+bk}} + C_2ke^{-t\sqrt{a+bk}} - C_3at - C_4a$$

2.2. If $a + bk = 0$:

$$x = C_1bt^3 + C_2bt^2 + C_3t + C_4$$

$$y = kx + 6C_1t + 2C_2$$

system_of_odes_linear_2eq_order2_type2

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

$$u = tx' - x, v = ty' - y$$

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

$$x'' = f(t)(a_1x + b_1y)$$

$$y'' = f(t)(a_2x + b_2y)$$

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 by multiplying appropriate constants and adding together original equations we obtain two independent equations:

$$z_1'' = k_1f(t)z_1, z_1 = a_2x + (k_1 - a_1)y$$

$$z_2'' = k_2f(t)z_2, z_2 = a_2x + (k_2 - a_1)y$$

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 :

$$a_2x + (k_1 - a_1)y = C_1 \int e^{k_1F(t)} dt + C_2$$

$$a_2x + (k_2 - a_1)y = C_3 \int e^{k_2F(t)} dt + C_4$$

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

$$u' = atf(t)v, v' = btf(t)u$$

The general solution of this system has the form

If $ab > 0$:

$$u = C_1 a e^{\sqrt{ab} \int tf(t) dt} + C_2 a e^{-\sqrt{ab} \int tf(t) dt}$$

$$v = C_1 \sqrt{ab} e^{\sqrt{ab} \int tf(t) dt} - C_2 \sqrt{ab} e^{-\sqrt{ab} \int tf(t) dt}$$

If $ab < 0$:

$$u = C_1 a \cos(\sqrt{|ab|} \int tf(t) dt) + C_2 a \sin(-\sqrt{|ab|} \int tf(t) dt)$$

$$v = C_1 \sqrt{|ab|} \sin(\sqrt{|ab|} \int tf(t) dt) + C_2 \sqrt{|ab|} \cos(-\sqrt{|ab|} \int tf(t) dt)$$

where C_1 and C_2 are arbitrary constants. On substituting the value of u and v in above equations and integrating the resulting expressions, the general solution will become

$$x = C_3 t + t \int \frac{u}{t^2} dt, y = C_4 t + t \int \frac{v}{t^2} dt$$

where C_3 and C_4 are arbitrary constants.

system_of_odes_linear_2eq_order2_type9

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

$$x = Ae^{\lambda t}, y = Be^{\lambda t}$$

On substituting these expressions into the original system and collecting the coefficients of the unknown A and B , one obtains

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)A + (b_1\lambda + d_1)B = 0$$

$$(a_2\lambda + c_2)A + (\lambda^2 + (b_2 - 1)\lambda + d_2)B = 0$$

The determinant of this system must vanish for nontrivial solutions A, B to exist. This requirement results in the following characteristic equation for λ

$$(\lambda^2 + (a_1 - 1)\lambda + c_1)(\lambda^2 + (b_2 - 1)\lambda + d_2) - (b_1\lambda + d_1)(a_2\lambda + c_2) = 0$$

If all roots k_1, \dots, k_4 of this equation are distinct, the general solution of the original system of the differential equations has the form

$$x = C_1(b_1\lambda_1 + d_1)e^{\lambda_1 t} - C_2(b_1\lambda_2 + d_1)e^{\lambda_2 t} - C_3(b_1\lambda_3 + d_1)e^{\lambda_3 t} - C_4(b_1\lambda_4 + d_1)e^{\lambda_4 t}$$

$$y = C_1(\lambda_1^2 + (a_1 - 1)\lambda_1 + c_1)e^{\lambda_1 t} + C_2(\lambda_2^2 + (a_1 - 1)\lambda_2 + c_1)e^{\lambda_2 t} + C_3(\lambda_3^2 + (a_1 - 1)\lambda_3 + c_1)e^{\lambda_3 t} + C_4(\lambda_4^2 + (a_1 - 1)\lambda_4 + c_1)e^{\lambda_4 t}$$

system_of_odes_linear_2eq_order2_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:

$$x' = (a_1 f(t) + g(t))x + a_2 f(t)y + a_3 f(t)z$$

$$y' = b_1 f(t)x + (b_2 f(t) + g(t))y + b_3 f(t)z$$

$$z' = c_1 f(t)x + c_2 f(t)y + (c_3 f(t) + g(t))z$$

The transformation

$$x = e^{\int g(t) dt} u, y = e^{\int g(t) dt} v, z = e^{\int g(t) dt} w, \tau = \int f(t) dt$$

leads to the system of constant coefficient linear differential equations

$$u' = a_1 u + a_2 v + a_3 w$$

$$v' = b_1 u + b_2 v + b_3 w$$

$$w' = c_1 u + c_2 v + c_3 w$$

These system of equations are solved by homogeneous linear system of constant coefficients of n equations of first order. Then substituting the value of u, v and w in transformed equation gives value of x, y and z .

system_of_odes_linear_neq_order1_type1`diofant.solvers.ode._linear_neq_order1_type1(match_)`System of n first-order constant-coefficient linear differential equations

$$Mx' = Lx + f(t)$$

Notes

The nonhomogeneous case is not implemented yet. Mass-matrix assumed to be invertible and provided general solution uses the Jordan canonical form for $A = M^{-1}L$, see [R517] (page 1276).

References

[R517] (page 1276)

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:

$$x = C_1t + F(C_1, C_2), y = C_2t + G(C_1, C_2)$$

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>

3.23.5 Information on the ode module

This module contains `dsolve()` (page 994) and different helper functions that it uses.

`dsolve()` (page 994) 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 994).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 994) - Solves ODEs.
- `classify_ode()` (page 997) - Classifies ODEs into possible hints for `dsolve()` (page 994).
- `checkodesol()` (page 998) - Checks if an equation is the solution to an ODE.
- `homogeneous_order()` (page 999) - Returns the homogeneous order of an expression.
- `infinitesimals()` (page 1000) - 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 994) to obtain the functionality provided by these functions:

- `odesimp()` (page 1002) - Does all forms of ODE simplification.
- `ode_sol_simplicity()` (page 1004) - A key function for comparing solutions by simplicity.
- `constantsimp()` (page 1003) - Simplifies arbitrary constants.
- `constant_renumber()` (page 1003) - Renumber arbitrary constants.
- `_handle_Integral()` (page 1045) - 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 997) 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 997) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated `Integral` (page 553) class. `dsolve()` (page 994), which is the user wrapper function around all of this, will then call `odesimp()` (page 1002) 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 994) 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 1041) 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 1006), 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 994) 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 994) code. See also the `classify_ode()` (page 997) 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 1002) 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 994). 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 997) (if the match function is more than just a few lines, like `undetermined_coefficients_match()` (page 1044), it should go outside of `classify_ode()` (page 997)). It should match the ODE without solving for it as much as possible, so that `classify_ode()` (page 997) 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 997). `classify_ode()` (page 997) will then send this to `dsolve()` (page 994), 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 997) 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 994) 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 553) instead of `integrate()` (page 546). 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 45) will integrate your solution. If this is not the case, you will need to write special code in `_handle_Integral()` (page 1045). 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 1002) 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 1002) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many `log()` (page 331) terms, so `odesimp()` (page 1002) calls `logcombine()` (page 928) 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 1003) take better advantage of it. It is better to put simplification in `odesimp()` (page 1002) than in your method, because it can then be turned off with the `simplify` flag in `dsolve()` (page 994). 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 998) (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 994), 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

```
>>> from diofant import log, exp
>>> from diofant.solvers.ode import _undetermined_coefficients_match
>>> from diofant.abc import x
>>> _undetermined_coefficients_match(9*x*exp(x) + exp(-x), x) == {'test': True,
→ 'trialset': {x*exp(x), exp(-x), exp(x)}}
True
>>> _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()`.

3.24 PDE

3.24.1 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.

See also:

[diofant.solvers.pde.pde_separate_add](#) (page 1046), [diofant.solvers.pde.pde_separate_mul](#) (page 1046)

Examples

```
>>> from diofant import E, Eq, Function, pde_separate, Derivative as D
>>> from diofant.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[E**(-X(x))*Derivative(X(x), x), E**T(t)*Derivative(T(t), t)]
```

```
>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), x, x)/X(x), Derivative(T(t), t, t)/T(t)]
```

pde_separate_add

`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

```
>>> from diofant import E, Eq, Function, pde_separate_add, Derivative as D
>>> from diofant.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[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

```
>>> from diofant import Function, Eq, pde_separate_mul, Derivative as D
>>> from diofant.abc import x, y
>>> u, X, Y = map(Function, 'uXY')
```

```
>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), x, x)/X(x), Derivative(Y(y), y, y)/Y(y)]
```

pdsolve

`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:

```
>>> from diofant import Function, Derivative
>>> from diofant.abc import x, y # x and y are the independent variables
>>> f = Function("f")(x, y) # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See `test_pde.py` for many tests, which serves also as a set of examples for how to use `pdsolve()`.
- `pdsolve` always returns an Equality class (except for the case when the hint is “all” or “all_Integral”). Note that it is not possible to get an explicit solution for $f(x, y)$ as in the case of ODE’s
- Do `help(pde.pde_hintname)` to get help more information on a specific hint

Examples

```
>>> from diofant.solvers.pde import pdsolve
>>> from diofant import Function, diff, Eq
>>> from diofant.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> pdsolve(eq)
Eq(f(x, y), 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

```
>>> from diofant.solvers.pde import classify_pde
>>> from diofant import Function, diff, Eq
>>> from diofant.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)))
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```

checkpdesol

`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

```
>>> from diofant import Function, symbols, diff
>>> from diofant.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
>>> assert checkpdesol(eq, sol)[0]
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, 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,)))
```

3.24.2 Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

pde_1st_linear_constant_coeff_homogeneous

`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:

```
>>> from diofant.solvers import pdsolve
>>> from diofant.abc import x, y, a, b, c
>>> from diofant import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform, use_unicode=False)
      d          d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y)
      dx          dy

>>> pprint(pdsolve(genform), use_unicode=False)
      -c*(a*x + b*y)
      -----
              2      2
             a  + b

f(x, y) = E          *F(-a*y + b*x)
```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Examples

```
>>> from diofant.solvers.pde import (
... pde_1st_linear_constant_coeff_homogeneous)
>>> from diofant import pdsolve
>>> from diofant import Function, diff, pprint
>>> from diofant.abc import x,y
>>> f = Function('f')
```

```
>>> pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y))
Eq(f(x, y), E**(-x/2 - y/2)*F(x - y))
>>> pprint(pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y)), use_unicode=False)
      x   y
      - - -
      2   2
f(x, y) = E      *F(x - y)
```

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:

```
>>> from diofant.solvers import pdsolve
>>> from diofant.abc import x, y, a, b, c
>>> from diofant import Function, pprint
>>> f = Function('f')
>>> G = Function('G')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*u + b*ux + c*uy - G(x,y)
>>> pprint(genform, use_unicode=False)
      d           d
a*f(x, y) + b*--(f(x, y)) + c*--(f(x, y)) - G(x, y)
      dx           dy
>>> pprint(pdsolve(genform, hint='1st_linear_constant_coeff_Integral'), use_
↳ unicode=False)
```

$$f(x, y) = E^{\frac{-ax - by}{\sqrt{b^2 + c^2}}} * F(\eta) + \frac{a^2 x^2 + 2abx + b^2 y + c^2 y^2}{2(b^2 + c^2)} * G\left(\frac{b^2 x + c^2 y}{\sqrt{b^2 + c^2}}\right) + \frac{-b^2 \eta + c^2 x \eta}{2(b^2 + c^2)}$$

```

\
| d(xi) |
|       |
|       |
/
-----
//|eta=-b*y + c*x, xi=b*x + c*y

```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Examples

```

>>> from diofant.solvers.pde import pdsolve
>>> from diofant import Function, diff, exp
>>> from diofant.abc import x,y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), E**(x/2 - y)*(E**(x/2 + 4*y)/15 + F(4*x + 2*y)))

```

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

$$a(x,y)\frac{df(x,y)}{dx} + a(x,y)\frac{df(x,y)}{dy} + c(x,y)f(x,y) - G(x,y)$$

where $a(x,y)$, $b(x,y)$, $c(x,y)$ and $G(x,y)$ are arbitrary functions in x and y . This PDE is converted into an ODE by making the following transformation.

1] ξ as x

2] η as the constant in the solution to the differential equation $\frac{dy}{dx} = -\frac{b}{a}$

Making the following substitutions reduces it to the linear ODE

$$a(\xi, \eta)\frac{du}{d\xi} + c(\xi, \eta)u - d(\xi, \eta) = 0$$

which can be solved using `dsolve`.

The general form of this PDE is:


```

>>> from diofant.solvers.pde import pdsolve
>>> from diofant.abc import x, y
>>> from diofant import Function, pprint
>>> a, b, c, G, f= [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform, use_unicode=False)
-G(x, y) + a(x, y)*f(x, y) + b(x, y)* $\frac{d}{dx}$ (f(x, y)) + c(x, y)* $\frac{d}{dy}$ (f(x, y))

```

References

- Viktor Grigoryan, “Partial Differential Equations” Math 124A - Fall 2010, pp.7

Examples

```

>>> from diofant.solvers.pde import pdsolve
>>> from diofant import Function, diff, pprint, exp
>>> from diofant.abc import x,y
>>> f = Function('f')
>>> eq = x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), E**(y**2/2)*F(x*y) + 1)

```

3.24.3 Information on the pde module

This module contains pdsolve() and different helper functions that it uses. It is heavily inspired by the ode module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- pdsolve() - Solves PDE’s
- classify_pde() - Classifies PDEs into possible hints for dsolve().
- **pde_separate()** - **Separate variables in partial differential equation either by additive or multiplicative separation approach.**

These are the helper functions in this module:

- pde_separate_add() - Helper function for searching additive separable solutions.
- **pde_separate_mul()** - **Helper function for searching multiplicative separable solutions.**

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various pde_hint() functions for more information on each (run help(pde)):

- 1st order linear homogeneous partial differential equations with constant coefficients.

- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.

3.25 Solvers

The *solvers* module in Diofant implements methods for solving equations.

3.25.1 Algebraic equations

Use *solve()* (page 1054) to solve algebraic equations. We suppose all equations are equaled to 0, so solving $x^{**2} == 1$ translates into the following code:

```
>>> from diofant.solvers import solve
>>> from diofant import Symbol
>>> x = Symbol('x')
>>> solve(x**2 - 1, x)
[-1, 1]
```

The first argument for *solve()* (page 1054) is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`diofant.solvers.solvers.solve(f, *symbols, **flags)`

Algebraically solves equations and systems of equations.

Currently supported are:

- polynomial,
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions.

Input is formed as:

•**f**

- a single Expr or Poly that must be zero,
- an Equality
- a Relational expression or boolean
- iterable of one or more of the above

•**symbols (object(s) to solve for) specified as**

- none given (other non-numeric objects will be used)
- single symbol
- denested list of symbols e.g. `solve(f, x, y)`
- ordered iterable of symbols e.g. `solve(f, [x, y])`

•**flags**

'dict'=True (default is False) return list (perhaps empty) of solution mappings

- 'set=True (default is False)** return list of symbols and set of tuple(s) of solution(s)
- 'exclude=[] (default)'** don't try to solve for any of the free symbols in exclude; if expressions are given, the free symbols in them will be extracted automatically.
- 'check=True (default)'** If False, don't do any testing of solutions. This can be useful if one wants to include solutions that make any denominator zero.
- 'numerical=True (default)'** do a fast numerical check if f has only one symbol.
- 'minimal=True (default is False)'** a very fast, minimal testing.
- 'warn=True (default is False)'** show a warning if checksol() could not conclude.
- 'simplify=True (default)'** simplify all but polynomials of order 3 or greater before returning them and (if check is not False) use the general simplify function on the solutions and the expression obtained when they are substituted into the function which should be zero
- 'force=True (default is False)'** make positive all symbols without assumptions regarding sign.
- 'rational=True (default)'** recast Floats as Rational; if this option is not used, the system containing floats may fail to solve because of issues with polys. If rational=None, Floats will be recast as rationals but the answer will be recast as Floats. If the flag is False then nothing will be done to the Floats.
- 'manual=True (default is False)'** do not use the polys/matrix method to solve a system of equations, solve them one at a time as you might "manually"
- 'implicit=True (default is False)'** allows solve to return a solution for a pattern in terms of other functions that contain that pattern; this is only needed if the pattern is inside of some invertible function like cos, exp, ...
- 'particular=True (default is False)'** instructs solve to try to find a particular solution to a linear system with as many zeros as possible; this is very expensive
- 'quick=True (default is False)'** when using particular=True, use a fast heuristic instead to find a solution with many zeros (instead of using the very slow method guaranteed to find the largest number of zeros possible)
- 'cubics=True (default)'** return explicit solutions when cubic expressions are encountered
- 'quartics=True (default)'** return explicit solutions when quartic expressions are encountered
- 'quintics=True (default)'** return explicit solutions (if possible) when quintic expressions are encountered

Notes

solve() with check=True (default) will run through the symbol tags to eliminate unwanted solutions. If no assumptions are included all possible solutions will be returned.

```
>>> from diofant import Symbol, solve
>>> x = Symbol("x")
>>> solve(x**2 - 1)
[-1, 1]
```

By using the positive tag only one solution will be returned:

```
>>> pos = Symbol("pos", positive=True)
>>> solve(pos**2 - 1)
[1]
```

Assumptions aren't checked when `solve()` input involves relationals or bools.

When the solutions are checked, those that make any denominator zero are automatically excluded. If you do not want to exclude such solutions then use the `check=False` option:

```
>>> from diofant import sin, limit
>>> solve(sin(x)/x) # 0 is excluded
[pi]
```

If `check=False` then a solution to the numerator being zero is found: $x = 0$. In this case, this is a spurious solution since $\sin(x)/x$ has the well known limit (without discontinuity) of 1 at $x = 0$:

```
>>> solve(sin(x)/x, check=False)
[0, pi]
```

In the following case, however, the limit exists and is equal to the the value of $x = 0$ that is excluded when `check=True`:

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve(eq, x)
[]
>>> solve(eq, x, check=False)
[0]
>>> limit(eq, x, 0, '-')
0
>>> limit(eq, x, 0, '+')
0
```

Disabling high-order, explicit solutions

When solving polynomial expressions, one might not want explicit solutions (which can be quite long). If the expression is univariate, `RootOf` instances will be returned instead:

```
>>> solve(x**3 - x + 1)
[-1/((-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)) - (-1/2 -
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3, -(-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3 - 1/((-1/2 +
sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)), -(3*sqrt(69)/2 +
27/2)**(1/3)/3 - 1/(3*sqrt(69)/2 + 27/2)**(1/3)]
>>> solve(x**3 - x + 1, cubics=False)
[RootOf(x**3 - x + 1, 0), RootOf(x**3 - x + 1, 1), RootOf(x**3 - x + 1, 2)]
```

If the expression is multivariate, no solution might be returned:

```
>>> solve(x**3 - x + a, x, cubics=False)
[]
```

Sometimes solutions will be obtained even when a flag is `False` because the expression could be factored. In the following example, the equation can be factored as the product of a linear and a quadratic factor so explicit solutions (which did not require solving a cubic expression) are obtained:

```
>>> eq = x**3 + 3*x**2 + x - 1
>>> solve(eq, cubics=False)
[-1, -1 + sqrt(2), -sqrt(2) - 1]
```

Solving equations involving radicals

Because of Diofant's use of the principle root (issue [sympy/sympy#8789](#)), some solutions to radical equations will be missed unless `check=False`:

```
>>> from diofant import root
>>> eq = root(x**3 - 3*x**2, 3) + 1 - x
>>> solve(eq)
[]
>>> solve(eq, check=False)
[1/3]
```

In the above example there is only a single solution to the equation. Other expressions will yield spurious roots which must be checked manually; roots which give a negative argument to odd-powered radicals will also need special checking:

```
>>> from diofant import real_root, Rational
>>> eq = root(x, 3) - root(x, 5) + Rational(1, 7)
>>> solve(eq) # this gives 2 solutions but misses a 3rd
[RootOf(7*_p**5 - 7*_p**3 + 1, 1)**15,
 RootOf(7*_p**5 - 7*_p**3 + 1, 2)**15]
>>> sol = solve(eq, check=False)
>>> [abs(eq.subs(x,i).n(2)) for i in sol]
[0.48, 0.e-110, 0.e-110, 0.052, 0.052]
```

The first solution is negative so `real_root` must be used to see that it satisfies the expression:

```
>>> abs(real_root(eq.subs(x, sol[0])).n(2))
0.e-110
```

If the roots of the equation are not real then more care will be necessary to find the roots, especially for higher order equations. Consider the following expression:

```
>>> expr = root(x, 3) - root(x, 5)
```

We will construct a known value for this expression at $x = 3$ by selecting the 1-th root for each radical:

```
>>> expr1 = root(x, 3, 1) - root(x, 5, 1)
>>> v = expr1.subs(x, -3)
```

The solve function is unable to find any exact roots to this equation:

```
>>> eq = Eq(expr, v); eq1 = Eq(expr1, v)
>>> solve(eq, check=False), solve(eq1, check=False)
([], [])
```

The function `unrad`, however, can be used to get a form of the equation for which numerical roots can be found:

```
>>> from diofant.solvers.solvers import unrad
>>> from diofant import roots, pprint
>>> e, (p, cov) = unrad(eq)
>>> pvals = roots(e)
>>> inversion = solve(cov, x)[0]
>>> xvals = [inversion.subs(p, i) for i in pvals]
```

Although eq or eq1 could have been used to find xvals, the solution can only be verified with expr1:

```
>>> z = expr - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z.subs(x, xi).n()) < 1e-9]
[]
>>> z1 = expr1 - v
>>> pprint([xi.n(chop=1e-9) for xi in xvals if abs(z1.subs(x, xi).n()) < 1e-9])
[-3.0]
```

Examples

The output varies according to the input and can be seen by example:

```
>>> from diofant import solve, Poly, Eq, Function, exp, I, sqrt
>>> from diofant.abc import x, y, z, a, b
>>> f = Function('f')
```

- boolean or univariate Relational

```
>>> solve(x < 3)
And(-oo < x, x < 3)
```

- to always get a list of solution mappings, use flag dict=True

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> solve([x - 3, y - 1], dict=True) == [{x: 3, y: 1}]
True
```

- to get a list of symbols and set of solution(s) use flag set=True

```
>>> solve([x**2 - 3, y - 1], set=True) == ([x, y], {(-sqrt(3), 1), (sqrt(3),
↪1)})
True
```

- single expression and single symbol that is in the expression

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
>>> solve(x**2 - y**2, x, set=True) == ([x], {(-y,), (y,)})
True
```

```
>>> solve(x**4 - 1, x, set=True) == ([x], {(-1,), (1,), (-I,), (I,)})
True
```

- single expression with no symbol that is in the expression

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

- single expression with no symbol given

In this case, all free symbols will be selected as potential symbols to solve for. If the equation is univariate then a list of solutions is returned; otherwise - as is the case when symbols are given as an iterable of length > 1 - a list of mappings will be returned.

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2) == [{x: -y}, {x: y}]
True
>>> solve(z**2*x**2 - z**2*y**2) == [{x: -y}, {x: y}, {z: 0}]
True
>>> solve(z**2*x - z**2*y**2) == [{x: y**2}, {z: 0}]
True
```

- when an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save one from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method.

```
>>> solve(f(x) - x, f(x))
[x]
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
>>> solve(f(x).diff(x) - f(x) - x, f(x))
[-x + Derivative(f(x), x)]
>>> solve(x + exp(x)**2, exp(x), set=True) == ([exp(x)], {(-sqrt(-x)),
↳ (sqrt(-x),)})
True
```

```
>>> from diofant import Indexed, IndexedBase, Tuple, sqrt
>>> A = IndexedBase('A')
>>> eqs = Tuple(A[1] + A[2] - 3, A[1] - A[2] + 1)
>>> solve(eqs, eqs.atoms(Indexed)) == {A[1]: 1, A[2]: 2}
True
```

-To solve for a *symbol* implicitly, use 'implicit=True':

```
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-E**x]
```

-It is possible to solve for anything that can be targeted with subs:

```
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2) == {y: -2 + sqrt(3), x + 2: -sqrt(3)}
True
```

-Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2) == {y: -sqrt(3)/(x + 3), x + 2: (-2*x - 6 + sqrt(3))/(x + 3)}
True
>>> solve(eqs, y*x, x) == {x: -y - 4, x*y: -3*y - sqrt(3)}
True
```

-if you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1) # /\ -1 is targeted, too
[x/(y - 1)]
>>> [_subs(z, -1) for _ in solve((x - y + 1).subs(-1, z), 1)]
[-x + y]
```

-To solve for a function within a derivative, use dsolve.

- single expression and more than 1 symbol

-when there is a linear solution

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

-when undetermined coefficients are identified

*that are linear

```
>>> solve((a + b)*x - b + 2, a, b) == {a: -2, b: 2}
True
```

*that are nonlinear

```
>>> solve((a + b)*x - b**2 + 2, a, b, set=True) == ([a, b], {(-sqrt(2), sqrt(2)), (sqrt(2), -sqrt(2))})
True
```

-if there is no linear solution then the first successful attempt for a nonlinear solution will be returned

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



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

•iterable of one or more of the above

-involving relationals or bools

```
>>> solve([x < 3, x - 2])
Eq(x, 2)
>>> solve([x > 3, x - 2])
false
```

-when the system is linear

*with a solution

```
>>> solve([x - 3], x)
{x: 3}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y) == {x: -3, y: 1}
True
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y, z) == {x: -3, y: 1}
True
>>> solve((x + 5*y - 2, -3*x + 6*y - z), z, x, y) == {x: -5*y + 2, z:
↳21*y - 6}
True
```

*without a solution

```
>>> solve([x + 3, x - 3])
[]
```

-when the system is not linear

```
>>> solve([x**2 + y - 2, y**2 - 4], x, y, set=True) == ([x, y], {(-2, -2),
↳(0, 2), (2, -2)})
True
```

-if no symbols are given, all free symbols will be selected and a list of mappings returned

```
>>> solve([x - 2, x**2 + y]) == [{x: 2, y: -4}]
True
>>> solve([x - 2, x**2 + f(x)], {f(x), x}) == [{x: 2, f(x): -4}]
True
```

-if any equation doesn't depend on the symbol(s) given it will be eliminated from the equation set and an answer may be given implicitly in terms of variables that were not of interest

```
>>> solve([x - y, y - 3], x)
{x: y}
```

`diofant.solvers.solvers.solve_linear(lhs, rhs=0, symbols=[], exclude=[])`

Return a tuple derived from $f = \text{lhs} - \text{rhs}$ that is either:

(numerator, denominator) of f If this comes back as (0, 1) it means that f is independent of the symbols in symbols, e.g:

```
y*cos(x)**2 + y*sin(x)**2 - y = y*(0) = 0
cos(x)**2 + sin(x)**2 = 1
```

If it comes back as (0, 0) there is no solution to the equation amongst the symbols given.

If the numerator is not zero then the function is guaranteed to be dependent on a symbol in symbols.

or

(symbol, solution) where symbol appears linearly in the numerator of f, is in symbols (if given) and is not in exclude (if given).

No simplification is done to f other than and mul=True expansion, so the solution will correspond strictly to a unique solution.

Examples

```
>>> from diofant.solvers.solvers import solve_linear
>>> from diofant.abc import x, y, z
```

These are linear in x and 1/x:

```
>>> solve_linear(x + y**2)
(x, -y**2)
>>> solve_linear(1/x - y**2)
(x, y**(-2))
```

When not linear in x or y then the numerator and denominator are returned.

```
>>> solve_linear(x**2/y**2 - 3)
(x**2 - 3*y**2, y**2)
```

If the numerator is a symbol then (0, 0) is returned if the solution for that symbol would have set any denominator to 0:

```
>>> solve_linear(1/(1/x - 2))
(0, 0)
>>> 1/(1/x) # to Diofant, this looks like x ...
x
>>> solve_linear(1/(1/x)) # so a solution is given
(x, 0)
```

If x is allowed to cancel, then this appears linear, but this sort of cancellation is not done so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> solve_linear(x**2*(1/x - z**2/x))
(x**2*(-z**2 + 1), x)
```

You can give a list of what you prefer for x candidates:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

You can also indicate what variables you don't want to consider:

```
>>> solve_linear(x + y + z, exclude=[x, z])
(y, -x - z)
```

If only x was excluded then a solution for y or z might be obtained.

`diofant.solvers.solvers.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.

See also:

[diofant.matrices.matrices.MatrixBase.rref](#) (page 616)

Examples

```
>>> from diofant import Matrix, solve_linear_system
>>> from diofant.abc import x, y
```

Solve the following system:

```
x + 4 y == 2
-2 x + y == 14
```

```
>>> system = Matrix(( (1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y) == {x: -6, y: 2}
True
```

A degenerate system returns an empty dictionary.

```
>>> system = Matrix(( (0,0,0), (0,0,0) ))
>>> solve_linear_system(system, x, y)
{}
```

`diofant.solvers.solvers.solve_undetermined_coeffs(equ, coeffs, sym, **flags)`

Solve equation of a type $p(x; a_1, \dots, a_k) = q(x)$ where both p, q are univariate polynomials and f depends on k parameters. The result of this functions is a dictionary with symbolic values of those parameters with respect to coefficients in q.

This functions accepts both Equations class instances and ordinary Diofant expressions. Specification of parameters and variable is obligatory for efficiency and simplicity reason.

```
>>> from diofant import Eq, Rational
>>> from diofant.abc import a, b, c, x
>>> from diofant.solvers import solve_undetermined_coefs
```

```
>>> solve_undetermined_coefs(Eq(2*a*x + a+b, x), [a, b], x) == {a: Rational(1, 2),
↳ b: -Rational(1, 2)}
True
```

```
>>> solve_undetermined_coefs(Eq(a*c*x + a+b, x), [a, b], x) == {a: 1/c, b: -1/c}
True
```

`diofant.solvers.solvers.nsolve(*args, **kwargs)`
Solve a nonlinear equation system numerically:

```
nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)
```

`f` is a vector function of symbolic expressions representing the system. `args` are the variables. If there is only one variable, this argument can be omitted. `x0` is a starting vector close to a solution.

Use the `modules` keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of `lambdify`.

Overdetermined systems are supported.

```
>>> from diofant import Symbol, nsolve
>>> import diofant
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
[-1.19287309935246]
[ 1.27844411169911]
```

For one-dimensional functions the syntax is simplified:

```
>>> from diofant import sin, nsolve
>>> from diofant.abc import x
>>> print(nsolve(sin(x), x, 2))
3.14159265358979
>>> print(nsolve(sin(x), 2))
3.14159265358979
```

`mpmath.findroot` is used, you can find there more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that this routine works only with the numerator of the function in the one-dimensional case, and for very steep functions near the root this may lead to a failure in the verification of the root. In this case you should use the flag `verify = False` and independently verify the solution.

```
>>> from diofant import cos, cosh
>>> from diofant.abc import i
>>> f = cos(x)*cosh(x) - 1
>>> nsolve(f, 3.14*100)
```

```
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (1.39267e+230 > 2.1684e-
↳19)
>>> ans = nsolve(f, 3.14*100, verify=False); print(ans)
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:

```
>>> bounds = lambda i: (3.14*i, 3.14*(i + 1))
>>> print(nsolve(f, bounds(100), solver='bisect', verify=False))
315.730061685774
```

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

Checks whether expression *expr* satisfies all assumptions.

assumptions is a dict of assumptions: {'assumption': True|False, ...}.

Examples

```
>>> from diofant import Symbol, pi, I, exp
>>> from diofant.solvers.solvers import check_assumptions
```

```
>>> check_assumptions(-5, integer=True)
True
>>> check_assumptions(pi, extended_real=True, integer=False)
True
>>> check_assumptions(pi, extended_real=True, negative=True)
False
>>> check_assumptions(exp(I*pi/7), extended_real=False)
True
```

```
>>> x = Symbol('x', extended_real=True, positive=True)
>>> check_assumptions(2*x + 1, extended_real=True, positive=True)
True
>>> check_assumptions(-2*x - 5, extended_real=True, positive=True)
False
```

None is returned if `check_assumptions()` could not conclude.

```
>>> check_assumptions(2*x - 1, extended_real=True, positive=True)
>>> z = Symbol('z')
>>> check_assumptions(z, extended_real=True)
```

`diofant.solvers.solvers.checksol(f, symbol, sol=None, **flags)`

Checks whether *sol* is a solution of equation $f == 0$.

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag `simplify=True`, the values in the dictionary will be simplified. *f* can be a single equation or an iterable of equations. A solution must satisfy all equations in *f* to be considered valid; if a solution does not satisfy any

equation, False is returned; if one or more checks are inconclusive (and none are False) then None is returned.

Examples

```
>>> from diofant import symbols
>>> from diofant.solvers import checksol
>>> x, y = symbols('x,y')
>>> checksol(x**4 - 1, x, 1)
True
>>> checksol(x**4 - 1, x, 0)
False
>>> checksol(x**2 + y**2 - 5**2, {x: 3, y: 4})
True
```

To check if an expression is zero using checksol, pass it as `f` and send an empty dictionary for symbol:

```
>>> checksol(x**2 + x - x*(x + 1), {})
True
```

None is returned if checksol() could not conclude.

flags:

'numerical=True (default)' do a fast numerical check if `f` has only one symbol.

'minimal=True (default is False)' a very fast, minimal testing.

'warn=True (default is False)' show a warning if checksol() could not conclude.

'simplify=True (default)' simplify solution before substituting into function and simplify the function before trying specific simplifications

'force=True (default is False)' make positive all symbols without assumptions regarding sign.

`diofant.solvers.solvers.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
  four terms having sqrt() factors
* there are only two terms with radicals
```

Examples

```
>>> from diofant.solvers.solvers import unrad
>>> from diofant.abc import x
>>> from diofant import sqrt, Rational, root, real_roots, solve
```

```
>>> unrad(sqrt(x)*x**Rational(1, 3) + 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])
```

3.25.2 Ordinary Differential equations (ODEs)

See [ODE](#) (page 994).

3.25.3 Partial Differential Equations (PDEs)

See [PDE](#) (page 1045).

3.25.4 Deutils (Utilities for solving ODE's and PDE's)

`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

```
>>> from diofant import Function
>>> from diofant.solvers.deutils import ode_order
>>> from diofant.abc import x
>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 +
... f(x).diff(x), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```

3.25.5 Recurrence Equations

`diofant.solvers.recurr.solve(f, y, init=None)`

Solve univariate recurrence with rational coefficients.

Given k -th order linear recurrence $Ly = f$, or equivalently:

$$a_k(n)y(n+k) + a_{k-1}(n)y(n+k-1) + \dots + a_0(n)y(n) = f(n)$$

where $a_i(n)$, for $i = 0, \dots, k$, are polynomials or rational functions in n , and f is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in n , finds all solutions or returns `None`, if none were found.

Initial conditions can be given as a dictionary in two forms:

1. { n_0 : v_0, n_1 : v_1, \dots, n_m : v_m }
2. { $y(n_0)$: $v_0, y(n_1)$: $v_1, \dots, y(n_m)$: v_m }

or as a list `L` of values:

$$L = [v_0, v_1, \dots, v_m]$$

where $L[i] = v_i$, for $i = 0, \dots, m$, maps to $y(n_i)$.

See also:

[rsolve_poly](#) (page 1069), [rsolve_ratio](#) (page 1069), [rsolve_hyper](#) (page 1070)

Examples

Lets consider the following recurrence:

$$(n-1)y(n+2) - (n^2 + 3n - 2)y(n+1) + 2n(n+1)y(n) = 0$$

```
>>> from diofant import Function, rsolve
>>> from diofant.abc import n
>>> y = Function('y')
```

```
>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)
```



```
>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)
```

```
>>> rsolve(f, y(n), { y(0):0, y(1):3 })
3*2**n - 3*factorial(n)
```

`diofant.solvers.recurr.rsolve_poly`(*coeffs*, *f*, *n*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all polynomial solutions over field K of characteristic zero.

The algorithm performs two basic steps:

1. Compute degree N of the general polynomial solution.
2. Find all polynomials of degree N or less of $Ly = f$.

There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it's smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with $N + 1$ unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only r indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovšek.

It is possible to generalize the algorithm implemented here to the case of linear q -difference and differential equations.

Lets say that we would like to compute m -th Bernoulli polynomial up to a constant. For this we can use $b(n+1) - b(n) = mn^{m-1}$ recurrence, which has solution $b(n) = B_m + C$. For example:

```
>>> from diofant import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)
```

```
>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

References

[R518] (page 1276), [R519] (page 1276), [R520] (page 1276)

`diofant.solvers.recurr.rsolve_ratio`(*coeffs*, *f*, *n*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$, where f is a polynomial, we seek for all rational solutions over field K of characteristic zero.

This procedure accepts only polynomials, however if you are interested in solving recurrence with rational coefficients then use `rsolve` which will pre-process the given equation and run this procedure with polynomial arguments.

The algorithm performs two basic steps:

1. Compute polynomial $v(n)$ which can be used as universal denominator of any rational solution of equation $Ly = f$.

2. Construct new linear difference equation by substitution $y(n) = u(n)/v(n)$ and solve it for $u(n)$ finding all its polynomial solutions. Return None if none were found.

Algorithm implemented here is a revised version of the original Abramov's algorithm, developed in 1989. The new approach is much simpler to implement and has better overall efficiency. This method can be easily adapted to q-difference equations case.

Besides finding rational solutions alone, this functions is an important part of Hyper algorithm were it is used to find particular solution of inhomogeneous part of a recurrence.

See also:

[rsolve_hyper](#) (page 1070)

References

[R521] (page 1276)

Examples

```
>>> from diofant.abc import x
>>> from diofant.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x,
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

`diofant.solvers.recurr.rsolve_hyper`(*coeffs*, *f*, *n*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $Ly = f$ we seek for all hypergeometric solutions over field K of characteristic zero.

The inhomogeneous part can be either hypergeometric or a sum of a fixed number of pairwise dissimilar hypergeometric terms.

The algorithm performs three basic steps:

1. Group together similar hypergeometric terms in the inhomogeneous part of $Ly = f$, and find particular solution using Abramov's algorithm.
2. Compute generating set of L and find basis in it, so that all solutions are linearly independent.
3. Form final solution with the number of arbitrary constants equal to dimension of basis of L .

Term $a(n)$ is hypergeometric if it is annihilated by first order linear difference equations with polynomial coefficients or, in simpler words, if consecutive term ratio is a rational function.

The output of this procedure is a linear combination of fixed number of hypergeometric terms. However the underlying method can generate larger class of solutions - D'Alembertian terms.

Note also that this method not only computes the kernel of the inhomogeneous equation, but also reduces in to a basis so that solutions generated by this procedure are linearly independent

References

[R522] (page 1276), [R523] (page 1276)

Examples

```
>>> from diofant.solvers import rsolve_hyper
>>> from diofant.abc import x
```

```
>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 + sqrt(5)/2)**x + C1*(-sqrt(5)/2 + 1/2)**x
```

```
>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2
```

3.25.6 Systems of Polynomial Equations

`diofant.solvers.polysys.solve_poly_system(seq, *gens, **args)`
Solve a system of polynomial equations.

Examples

```
>>> from diofant import solve_poly_system
>>> from diofant.abc import x, y
```

```
>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]
```

`diofant.solvers.polysys.solve_triangulated(polys, *gens, **args)`
Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

References

1. Patrizia Gianni, Teo Mora, Algebraic Solution of System of Polynomial Equations using Groebner Bases, AAEECC-5 on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS 356 247-257, 1989

Examples

```
>>> from diofant.solvers.polysys import solve_triangulated
>>> from diofant.abc import x, y, z
```

```
>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]
```

```
>>> solve_triangular(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

3.25.7 Diophantine Equations (DEs)

See *Diophantine* (page 1072)

3.25.8 Inequalities

See *Inequality Solvers* (page 1093)

3.26 Diophantine

3.26.1 Diophantine equations

The word “Diophantine” comes with the name Diophantus, a mathematician lived in the great city of Alexandria sometime around 250 AD. Often referred to as the “father of Algebra”, Diophantus in his famous work “Arithmetica” presented 150 problems that marked the early beginnings of number theory, the field of study about integers and their properties. Diophantine equations play a central and an important part in number theory.

We call a “Diophantine equation” to an equation of the form, $f(x_1, x_2, \dots, x_n) = 0$ where $n \geq 2$ and x_1, x_2, \dots, x_n are integer variables. If we can find n integers a_1, a_2, \dots, a_n such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$ satisfies the above equation, we say that the equation is solvable. You can read more about Diophantine equations in ¹ and ².

Currently, following five types of Diophantine equations can be solved using *diophantine()* (page 1077) 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: $x_1^2 + x_2^2 + \dots + x_n^2 = k$

3.26.2 Module structure

This module contains *diophantine()* (page 1077) and helper functions that are needed to solve certain Diophantine equations. It’s structured in the following manner.

- *diophantine()* (page 1077)
 - *diop_solve()* (page 1077)

¹ Andreescu, Titu. Andrica, Dorin. Cucurezeanu, Ion. An Introduction to Diophantine Equations

² Diophantine Equation, Wolfram Mathworld, [online]. Available: <http://mathworld.wolfram.com/DiophantineEquation.html>

- * `classify_diop()` (page 1078)
- * `diop_linear()` (page 1078)
- * `diop_quadratic()` (page 1080)
- * `diop_ternary_quadratic()` (page 1084)
- * `diop_general_pythagorean()` (page 1085)
- * `diop_general_sum_of_squares()` (page 1086)
- `merge_solution()` (page 1088)

When an equation is given to `diophantine()` (page 1077), it factors the equation (if possible) and solves the equation given by each factor by calling `diop_solve()` (page 1077) separately. Then all the results are combined using `merge_solution()` (page 1088).

`diop_solve()` (page 1077) internally uses `classify_diop()` (page 1078) 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 1078) returned “linear” as the type of the equation, then `diop_solve()` (page 1077) calls `diop_linear()` (page 1078) to solve the equation.

Each of the functions, `diop_linear()` (page 1078), `diop_quadratic()` (page 1080), `diop_ternary_quadratic()` (page 1084), `diop_general_pythagorean()` (page 1085) and `diop_general_sum_of_squares()` (page 1086) solves a specific type of equations and the type can be easily guessed by its 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.

3.26.3 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.

```
>>> from diofant import symbols
>>> 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 1077).

```
>>> 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 1077) always return a set of tuples. But `diop_solve()` (page 1077) 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 1078), which is what `diop_solve()` (page 1077) 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:

```
>>> m = symbols("m", integer=True)
>>> diop_solve(2*x + 3*y - 5, m)
(3*m_0 - 5, -2*m_0 + 5)
```

For linear Diophantine equations, the customized parameter is the prefix used for each free variable in the solution. Consider the following example:

```
>>> diop_solve(2*x + 3*y - 5*z + 7, m)
(m_0, -9*m_0 - 5*m_1 - 14, -5*m_0 - 3*m_1 - 7)
```

In the solution above, `m_0` and `m_1` are independent free variables.

Please note that for the moment, users can set the parameter only for linear Diophantine equations and binary quadratic equations.

Let's try solving a binary quadratic equation which is an equation with two variables and has a degree of two. Before trying to solve these equations, an idea about various cases associated with the equation would help a lot. Please refer ³ and ⁴ for detailed analysis of different cases and the nature of the solutions. Let us define $\Delta = b^2 - 4ac$ w.r.t. the binary quadratic $ax^2 + bxy + cy^2 + dx + ey + f = 0$.

When $\Delta < 0$, there are either no solutions or only a finite number of solutions.

```
>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
{(2, 1), (5, 1)}
```

In the above equation $\Delta = (-4)^2 - 4 * 1 * 8 = -16$ and hence only a finite number of solutions exist.

When $\Delta = 0$ we might have either no solutions or parameterized solutions.

³ Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>

⁴ Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <http://www.jpr2718.org/ax2p.pdf>

```
>>> diophantine(3*x**2 - 6*x*y + 3*y**2 - 3*x + 7*y - 5)
set()
>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
{(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)}
>>> diophantine(x**2 + 2*x*y + y**2 - 3*x - 3*y)
{(t_0, -t_0), (t_0, -t_0 + 3)}
```

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> from diofant import sqrt
>>> n = symbols("n", integer=True)
>>> s = diophantine(x**2 - 2*y**2 - 2*x - 4*y, n)
>>> x_1, y_1 = s.pop()
>>> x_2, y_2 = s.pop()
>>> x_n = -(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 - sqrt(2)*(2*sqrt(2)
↳ + 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1
>>> x_1 == x_n or x_2 == x_n
True
>>> y_n = -sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 + sqrt(2)*(2*sqrt(2)
↳ + 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1
>>> y_1 == y_n or y_2 == y_n
True
```

Here n is an integer. Although x_n and y_n may not look like integers, substituting in specific values for n (and simplifying) shows that they are. For example consider the following example where we set n equal to 9.

```
>>> from diofant import simplify
>>> simplify(x_n.subs({n: 9}))
-9369318
```

Any binary quadratic of the form $ax^2 + bxy + cy^2 + dx + ey + f = 0$ can be transformed to an equivalent form $X^2 - DY^2 = N$.

```
>>> from diofant.solvers.diophantine import find_DN, diop_DN, transformation_to_DN
>>> find_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
(5, 920)
```

So, the above equation is equivalent to the equation $X^2 - 5Y^2 = 920$ after a linear transformation. If we want to find the linear transformation, we can use `transformation_to_DN()` (page 1082)

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
```

Here A is a 2 X 2 matrix and B is a 2 X 1 matrix such that the transformation

$$\begin{bmatrix} X \\ Y \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + B$$

gives the equation $X^2 - 5Y^2 = 920$. Values of A and B are as follows.

```
>>> A
Matrix([
[1/10, 3/10],
```

```
[ 0, 1/5]])
>>> B
Matrix([
 [ 1/5],
 [-11/5]])
```

We can solve an equation of the form $X^2 - DY^2 = N$ by passing D and N to `diop_DN()` (page 1080)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation does not have solutions.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$. These type of equations either have infinitely many solutions or no solutions (except the obvious solution (0, 0, 0))

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
set()
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2)}
```

If you are only interested about a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use `diop_ternary_quadratic()` (page 1084).

```
>>> 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 1084) first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses `descent()` (page 1085) to solve the latter equation. You can refer to the docs of `transformation_to_normal()` (page 1093) to find more on this. The equation $w^2 = AX^2 + BY^2$ can be solved more easily by using the Aforementioned `descent()` (page 1085).

```
>>> 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 Diophantine module.

```
>>> from diofant.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
{(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5,
 60*t3*t5, 210*t4*t5, 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)}
```

function `diop_general_pythagorean()` (page 1085) can also be called directly to solve the same equation. This is true about the general sum of squares too. Either you can call `diop_general_pythagorean()` (page 1085) or use the high level API.

```
>>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
{(8, 4, 4, 4, 0, 0)}
```


If you want to get a more thorough idea about the the Diophantine module please refer to the following blog.

<http://thilinaatdiofant.wordpress.com/>

3.26.4 References

3.26.5 User Functions

These are functions that are imported into the global namespace with `from diofant import *`. These functions are intended for use by ordinary users of Diofant.

diophantine

`diofant.solvers.diophantine.diophantine(eq, param=Symbol('t', integer=True))`

Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.

For example, when solving, $x^2 - y^2 = 0$ this is treated as $(x + y)(x - y) = 0$ and $x + y = 0$ and $x - y = 0$ are solved independently and combined. Each term is solved by calling `diop_solve()`.

Output of `diophantine()` is a set of tuples. Each tuple represents a solution of the input equation. In a tuple, solution for each variable is listed according to the alphabetic order of input variables. i.e. if we have an equation with two variables a and b , first element of the tuple will give the solution for a and the second element will give the solution for b .

Parameters eq : Relational or Expr

an equation (to be solved)

t : Symbol, optional

the parameter to be used in the solution.

See also:

`diofant.solvers.diophantine.diop_solve` (page 1077)

Examples

```
>>> from diofant.solvers.diophantine import diophantine
>>> from diofant.abc import x, y, z
>>> diophantine(x**2 - y**2)
{(-t_0, -t_0), (t_0, -t_0)}
```

diop_solve

`diofant.solvers.diophantine.diop_solve(eq, param=Symbol('t', integer=True))`

Solves the diophantine equation `eq`.

Similar to `diophantine()` but doesn't try to factor `eq` as latter does. 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.

See also:

[diofant.solvers.diophantine.diophantine](#) (page 1077)

Examples

```
>>> from diofant.solvers.diophantine import diop_solve
>>> from diofant.abc import x, y, z, w
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, -2*t_0 + 5)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, -4*t_1 + 5, t_0 - 3*t_1 + 5)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t_0, t_0 + t_1, -2*t_0 - 3*t_1 - 4*t_2 - 6, -t_0 - 2*t_1 - 3*t_2 - 6)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (2, -1), (2, 1)}
```

classify_diop

`diofant.solvers.diophantine.classify_diop(eq)`

Helper routine used by `diop_solve()` to find the type of the eq etc.

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)`. Type is an element in the set {"linear", "binary_quadratic", "general_pythagorean", "homogeneous_ternary_quadratic", "univariate", "general_sum_of_squares"}

Parameters eq : Expr

an expression, which is assumed to be zero.

Examples

```
>>> from diofant.solvers.diophantine import classify_diop
>>> from diofant.abc import x, y, z, w, t
>>> 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: 0, y**2: 1, x*y: -1}, 'binary_quadratic')
```

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.

See also:

`diofant.solvers.diophantine.diop_quadratic` (page 1080), `diofant.solvers.diophantine.diop_ternary_quadratic` (page 1084), `diofant.solvers.diophantine.diop_general_pythagorean` (page 1085), `diofant.solvers.diophantine.diop_general_sum_of_squares` (page 1086)

Examples

```
>>> from diofant.solvers.diophantine import diop_linear
>>> from diofant.abc import x, y, z
>>> from diofant import Integer
>>> diop_linear(2*x - 3*y - 5) # solves equation 2*x - 3*y - 5 = 0
(-3*t_0 - 5, -2*t_0 - 5)
```

Here $x = -3*t_0 - 5$ and $y = -2*t_0 - 5$

```
>>> diop_linear(2*x - 3*y - 4*z - 3)
(t_0, -6*t_0 - 4*t_1 + 3, 5*t_0 + 3*t_1 - 3)
```

base_solution_linear

`diofant.solvers.diophantine.base_solution_linear(c, a, b, t=None)`

Return the base solution for a linear diophantine equation with two variables.

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

```
>>> from diofant.solvers.diophantine import base_solution_linear
>>> from diofant.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, -2*t + 5)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

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.

See also:

`diofant.solvers.diophantine.diop_linear` (page 1078), `diofant.solvers.diophantine.diop_ternary_quadratic` (page 1084), `diofant.solvers.diophantine.diop_general_sum_of_squares` (page 1086), `diofant.solvers.diophantine.diop_general_pythagorean` (page 1085)

References

[R471] (page 1276), [R472] (page 1276)

Examples

```
>>> from diofant.abc import x, y, t
>>> from diofant.solvers.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

diop_DN

`diofant.solvers.diophantine.diop_DN(D, N, t=Symbol('t', integer=True))`

Solves the equation $x^2 - Dy^2 = N$.

Mainly concerned in the case $D > 0$, D is not a perfect square, which is the same as generalized Pell equation. To solve the generalized Pell equation this function Uses LMM algorithm. Refer [R473] (page 1276) for more details on the algorithm. Returns one solution for each class of the solutions. Other solutions of the class can be constructed according to the values of D and N . Returns a list containing the solution tuples (x, y) .

Parameters `D, N` : Integer

correspond to D and N in the equation.

`t` : Symbol, optional

is the parameter to be used in the solutions.

See also:

`diofant.solvers.diophantine.find_DN` (page 1083), `diofant.solvers.diophantine.diop_bf_DN` (page 1081)

References

[R473] (page 1276)

Examples

```
>>> from diofant.solvers.diophantine import diop_DN
>>> diop_DN(13, -4) # Solves equation x**2 - 13*y**2 = -4
[(3, 1), (393, 109), (36, 10)]
```

The output can be interpreted as follows: There are three fundamental solutions to the equation $x^2 - 13y^2 = -4$ given by (3, 1), (393, 109) and (36, 10). Each tuple is in the form (x, y), i. e. solution (3, 1) means that $x = 3$ and $y = 1$.

```
>>> diop_DN(986, 1) # Solves equation x**2 - 986*y**2 = 1
[(49299, 1570)]
```

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 = 1', only the solutions with $x \geq y$ are found. For more details, see the References.

References

[R474] (page 1276), [R475] (page 1276)

Examples

```
>>> from diofant.solvers.diophantine import cornacchia
>>> cornacchia(2, 3, 35) # equation 2x**2 + 3y**2 = 35
{(2, 3), (4, 1)}
>>> cornacchia(1, 1, 25) # equation x**2 + y**2 = 25
{(4, 3)}
```

diop_bf_DN

`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. For more information on the case refer [\[R476\]](#) (page 1276). 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.

See also:

[diofant.solvers.diophantine.diop_DN](#) (page 1080)

References

[\[R476\]](#) (page 1276)

Examples

```
>>> from diofant.solvers.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

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. Refer [\[R477\]](#) (page 1276) for more detailed information on the transformation. This function returns a tuple (A, B) where A is a 2 X 2 matrix and B is a 2 X 1 matrix such that,

$\text{Transpose}([x \ y]) = A * \text{Transpose}([X \ Y]) + B$

Parameters eq : Expr

the quadratic expression to be transformed.

See also:

[diofant.solvers.diophantine.find_DN](#) (page 1083)

References

[\[R477\]](#) (page 1276)

Examples

```
>>> from diofant.abc import x, y
>>> from diofant.solvers.diophantine import transformation_to_DN
>>> from diofant.solvers.diophantine import classify_diop
>>> A, B = transformation_to_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
>>> A
Matrix([
[1/26, 3/26],
[ 0, 1/13]])
>>> B
Matrix([
[-6/13],
[-4/13]])
```

A, B returned are such that $\text{Transpose}((x\ y)) = A * \text{Transpose}((X\ Y)) + B$. Substituting these values for x and y and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```
>>> from diofant.abc import X, Y
>>> from diofant import Matrix, simplify, Subs
>>> 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(Subs(x**2 - 3*x*y - y**2 - 2*y + 1, (x, y), (u, v)).doit())
>>> 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.

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.

See also:

[diofant.solvers.diophantine.transformation_to_DN](#) (page 1082)

References

[R478] (page 1276)

Examples

```
>>> from diofant.abc import x, y
>>> from diofant.solvers.diophantine import find_DN
>>> find_DN(x**2 - 3*x*y - y**2 - 2*y + 1)
(13, -884)
```

Interpretation of the output is that we get $X^2 - 13Y^2 = -884$ after transforming $x^2 - 3xy - y^2 - 2y + 1$ using the transformation returned by `transformation_to_DN()`.

diop_ternary_quadratic

`diofant.solvers.diophantine.diop_ternary_quadratic(eq)`

Solves the general quadratic ternary form, $ax^2 + by^2 + cz^2 + fxy + gyz + hzx = 0$.

Parameters `eq` : Expr

should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

Returns tuple

which is a base solution for the above equation. If there are no solutions, `(None, None, None)` is returned.

Examples

```
>>> from diofant.abc import x, y, z
>>> from diofant.solvers.diophantine import diop_ternary_quadratic
>>> diop_ternary_quadratic(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic(45*x**2 - 7*y**2 - 8*x*y - z**2)
(28, 45, 105)
>>> diop_ternary_quadratic(x**2 - 49*y**2 - z**2 + 13*z*y - 8*x*y)
(9, 1, 5)
```

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.

Examples


```
>>> from diofant.solvers.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(36)
6
>>> square_factor(1)
1
```

descent

`diofant.solvers.diophantine.descent(A, B)`

Lagrange's *descent*() with lattice-reduction to find solutions to $x^2 = Ay^2 + Bz^2$.

Here A and B should be square free and pairwise prime. Always should be called with suitable A and B so that the above equation has solutions.

This is more faster than the normal Lagrange's descent algorithm because the gaussian reduction is used.

References

[R479] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import descent
>>> descent(3, 1) # x**2 = 3*y**2 + z**2
(1, 0, 1)
```

$(x, y, z) = (1, 0, 1)$ is a solution to the above equation.

```
>>> descent(41, -113)
(-16, -3, 1)
```

diop_general_pythagorean

`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.solvers.diophantine import diop_general_pythagorean
>>> from diofant.abc import a, b, c, d, e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2 + 10*m2**2 + 10*m3**2 - 10*m4**2, 15*m1**2 + 15*m2**2 + 15*m3**2
↳ + 15*m4**2, 15*m1*m4, 12*m2*m4, 60*m3*m4)
```

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. Currently there is no way to set `limit` using higher level API's like `diophantine()` or `diop_solve()` but that will be fixed soon.

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. Refer [\[R480\]](#) (page 1277) for more details.

References

[\[R480\]](#) (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import diop_general_sum_of_squares
>>> from diofant.abc import a, b, c, d, e, f
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345) == {(0,
↳ 48, 5, 4, 0)}
True
```

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 upto 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 there are no partitions of n with size k then an empty tuple is 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.

References

[R481] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(3, 1, 1)
>>> next(g)
(2, 2, 1)
```

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, None, None) if $n = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$. See [R482] (page 1277) for more details.

Parameters **n** : int

a non-negative integer.

References

[R482] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import sum_of_three_squares
>>> sum_of_three_squares(44542)
(207, 37, 18)
```

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.

References

[R483] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import sum_of_four_squares
>>> sum_of_four_squares(3456)
(8, 48, 32, 8)
>>> sum_of_four_squares(1294585930293)
(0, 1137796, 2161, 1234)
```

3.26.6 Internal Functions

These functions are intended for the internal use in Diophantine module.

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.

extended_euclid

`diofant.solvers.diophantine.extended_euclid(a, b)`

For given a, b returns a tuple containing integers x, y and d such that $ax + by = d$. Here $d = \gcd(a, b)$.

Parameters a, b : Integer

Returns tuple

x, y and $\gcd(a, b)$.

Examples

```
>>> from diofant.solvers.diophantine import extended_euclid
>>> extended_euclid(4, 6)
(-1, 1, 2)
>>> extended_euclid(3, 5)
(2, -1, 1)
```

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. Refer [\[R484\]](#) (page 1277) for more detailed information.

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.

References

[\[R484\]](#) (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import PQa
>>> pqa = PQa(13, 4, 5) # (13 + sqrt(5))/4
>>> next(pqa) # (P_0, Q_0, a_0, A_0, B_0, G_0)
(13, 4, 3, 3, 1, -1)
>>> next(pqa) # (P_1, Q_1, a_1, A_1, B_1, G_1)
(-1, 1, 1, 4, 1, 3)
```

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 . See reference [\[R485\]](#) (page 1277). 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

References

[\[R485\]](#) (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

simplified

`diofant.solvers.diophantine.simplified(x, y, z)`

Simplify the solution (x, y, z) .

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 $ax^2 + by^2 + cz^2 + fxy + gyz + hxz = 0$.

References

[\[R486\]](#) (page 1277)

Examples

```
>>> from diofant.abc import x, y, z
>>> from diofant.solvers.diophantine import parametrize_ternary_quadratic
>>> parametrize_ternary_quadratic(x**2 + y**2 - z**2)
(2*p*q, p**2 - q**2, p**2 + q**2)
```

Here p and q are two co-prime integers.

```
>>> parametrize_ternary_quadratic(3*x**2 + 2*y**2 - z**2 - 2*x*y + 5*y*z - 7*y*z)
(2*p**2 - 2*p*q - q**2, 2*p**2 + 2*p*q - q**2, 2*p**2 - 2*p*q + 3*q**2)
>>> parametrize_ternary_quadratic(124*x**2 - 30*y**2 - 7729*z**2)
(-1410*p**2 - 363263*q**2, 2700*p**2 + 30916*p*q - 695610*q**2, -60*p**2 +
↳5400*p*q + 15458*q**2)
```

diop_ternary_quadratic_normal

`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

```
>>> from diofant.abc import x, y, z
>>> from diofant.solvers.diophantine import diop_ternary_quadratic_normal
>>> diop_ternary_quadratic_normal(x**2 + 3*y**2 - z**2)
(1, 0, 1)
>>> diop_ternary_quadratic_normal(4*x**2 + 5*y**2 - z**2)
(1, 0, 2)
>>> diop_ternary_quadratic_normal(34*x**2 - 3*y**2 - 301*z**2)
(4, 9, 1)
```

ldescent

`diofant.solvers.diophantine.ldescent(A, B)`

Uses Lagrange's method to find a non trivial solution to $w^2 = Ax^2 + By^2$.

Here, $A \neq 0$ and $B \neq 0$ and A and B are square free. Output a tuple (w_0, x_0, y_0) which is a solution to the above equation.

References

[R487] (page 1277), [R488] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import ldescent
>>> ldescent(1, 1) # w^2 = x^2 + y^2
(1, 1, 0)
>>> ldescent(4, -7) # w^2 = 4x^2 - 7y^2
(2, -1, 0)
```

This means that $x = -1, y = 0$ and $w = 2$ is a solution to the equation $w^2 = 4x^2 - 7y^2$

```
>>> ldescent(5, -1) # w^2 = 5x^2 - y^2
(2, 1, -1)
```

gaussian_reduce

`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

[R489] (page 1277), [R490] (page 1277)

holzer

`diofant.solvers.diophantine.holzer(x_0, y_0, z_0, a, b, c)`

Simplify the solution (x_0, y_0, z_0) of the equation $ax^2 + by^2 = cz^2$ with $a, b, c > 0$ and $z_0^2 \geq |ab|$ to a new reduced solution (x, y, z) such that $z^2 \leq |ab|$.

prime_as_sum_of_two_squares

`diofant.solvers.diophantine.prime_as_sum_of_two_squares(p)`

Represent a prime p which is congruent to 1 mod 4, as a sum of two squares.

References

[R491] (page 1277)

Examples

```
>>> from diofant.solvers.diophantine import prime_as_sum_of_two_squares
>>> prime_as_sum_of_two_squares(5)
(2, 1)
```

pairwise_prime

`diofant.solvers.diophantine.pairwise_prime(a, b, c)`

Transform $ax^2 + by^2 + cz^2 = 0$ into an equivalent equation $a'x^2 + b'y^2 + c'z^2 = 0$ where a', b', c' are pairwise relatively prime.

Returns a tuple containing a', b', c' . $\gcd(a, b, c)$ should equal 1 for this to work. The solutions for $ax^2 + by^2 + cz^2 = 0$ can be recovered from the solutions of $a'x^2 + b'y^2 + c'z^2 = 0$.

See also:

`diofant.solvers.diophantine.make_prime` (page 1093), `diofant.solvers.diophantine.reconstruct` (page 1093)

Examples

```
>>> from diofant.solvers.diophantine import pairwise_prime
>>> pairwise_prime(6, 15, 10)
(5, 2, 3)
```

make_prime

`diofant.solvers.diophantine.make_prime(a, b, c)`

Transform the equation $ax^2 + by^2 + cz^2 = 0$ to an equivalent equation $a'x^2 + b'y^2 + c'z^2 = 0$ with $\gcd(a', b') = 1$.

Returns a tuple (a', b', c') which satisfies above conditions. Note that in the returned tuple $\gcd(a', c')$ and $\gcd(b', c')$ can take any value.

See also:

[`diofant.solvers.diophantine.reconstruct`](#) (page 1093)

Examples

```
>>> from diofant.solvers.diophantine import make_prime
>>> make_prime(4, 2, 7)
(2, 1, 14)
```

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 a transformed version of the above equation.

transformation_to_normal

`diofant.solvers.diophantine.transformation_to_normal(eq)`

Returns the transformation Matrix from general ternary quadratic equation eq to normal form.

General form of the ternary quadratic equation is $ax^2 + by^2 + cz^2 + dxy + eyz + fxz$. This function returns a 3X3 transformation Matrix which transforms the former equation to the form $ax^2 + by^2 + cz^2 = 0$. This is not used in solving ternary quadratics. Only implemented for the sake of completeness.

3.27 Inequality Solvers

`diofant.solvers.inequalities.solve_rational_inequalities(eqs)`

Solve a system of rational inequalities with rational coefficients.

See also:

[`solve_poly_inequality`](#) (page 1094)

Examples

```
>>> from diofant.abc import x
>>> from diofant import Poly
>>> from diofant.solvers.inequalities import solve_rational_inequalities
```

```
>>> solve_rational_inequalities([[(Poly(-x + 1), Poly(1, x)), '>='],
...                               ((Poly(-x + 1), Poly(1, x)), '<=')]])
{1}
```

```
>>> solve_rational_inequalities([[(Poly(x), Poly(1, x)), '!='],
...                               ((Poly(-x + 1), Poly(1, x)), '>=')]])
(-oo, 0) U (0, 1]
```

`diofant.solvers.inequalities.solve_poly_inequality(poly, rel)`

Solve a polynomial inequality with rational coefficients.

See also:

[solve_poly_inequalities](#) (page 1094)

Examples

```
>>> from diofant import Poly
>>> from diofant.solvers.inequalities import solve_poly_inequality
>>> from diofant.abc import x
```

```
>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')
[{0}]
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')
[(-oo, -1), (-1, 1), (1, oo)]
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')
[{-1}, {1}]
```

`diofant.solvers.inequalities.solve_poly_inequalities(polys)`

Solve polynomial inequalities with rational coefficients.

Examples

```
>>> from diofant.solvers.inequalities import solve_poly_inequalities
>>> from diofant.polys import Poly
>>> from diofant.abc import x
>>> solve_poly_inequalities(((Poly(x**2 - 3), ">"),
...                          (Poly(-x**2 + 1), ">")))
(-oo, -sqrt(3)) U (-1, 1) U (sqrt(3), oo)
```

`diofant.solvers.inequalities.reduce_rational_inequalities(exprs, gen, relational=True)`

Reduce a system of rational inequalities with rational coefficients.

Examples

```
>>> from diofant import Poly, Symbol
>>> from diofant.solvers.inequalities import reduce_rational_inequalities
```

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

```
>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)
>>> reduce_rational_inequalities([[x + 2 > 0]], x)
-2 < x
>>> reduce_rational_inequalities([[x + 2, ">"]], x)
-2 < x
>>> reduce_rational_inequalities([[x + 2]], x)
Eq(x, -2)
```

`diofant.solvers.inequalities.reduce_piecewise_inequality(expr, rel, gen)`
Reduce an inequality with nested piecewise functions.

See also:

[*reduce_piecewise_inequalities*](#) (page 1095)

Examples

```
>>> from diofant import Abs, Symbol, Piecewise
>>> from diofant.solvers.inequalities import reduce_piecewise_inequality
```

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

```
>>> reduce_piecewise_inequality(Abs(x - 5) - 3, '<', x)
And(2 < x, x < 8)
>>> reduce_piecewise_inequality(Abs(x + 2)*3 - 13, '<', x)
And(-19/3 < x, x < 7/3)
```

```
>>> reduce_piecewise_inequality(Piecewise((1, x < 1),
...                                     (3, True)) - 1, '>', x)
1 <= x
```

`diofant.solvers.inequalities.reduce_piecewise_inequalities(exprs, gen)`
Reduce a system of inequalities with nested piecewise functions.

See also:

[*reduce_piecewise_inequality*](#) (page 1095)

Examples

```
>>> from diofant import Abs, Symbol
>>> from diofant.solvers.inequalities import reduce_piecewise_inequalities
```

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

```
>>> reduce_piecewise_inequalities([(Abs(3*x - 5) - 7, '<'),
...                               (Abs(x + 25) - 13, '>')], x)
And(-2/3 < x, Or(-12 < x, x < -38), x < 4)
>>> reduce_piecewise_inequalities([(Abs(x - 4) + Abs(3*x - 5) - 7, '<')], x)
And(1/2 < x, x < 4)
```

`diofant.solvers.inequalities.reduce_inequalities(inequalities, symbols=[])`
 Reduce a system of inequalities with rational coefficients.

Examples

```
>>> from diofant.solvers.inequalities import reduce_inequalities
```

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

```
>>> reduce_inequalities(0 <= x + 3, [])
-3 <= x
>>> reduce_inequalities(0 <= x + y*2 - 1, [x])
-2*y + 1 <= x
```

`diofant.solvers.inequalities.solve_univariate_inequality(expr, gen, relational=True)`
 Solves a real univariate inequality.

Examples

```
>>> from diofant.solvers.inequalities import solve_univariate_inequality
>>> from diofant.core.symbol import Symbol
```

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

```
>>> solve_univariate_inequality(x**2 >= 4, x)
Or(2 <= x, x <= -2)
>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
(-oo, -2] U [2, oo)
```

3.28 Tensors

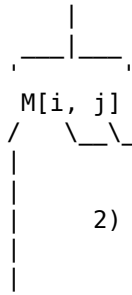
A module to manipulate symbolic objects with indices including tensors

3.28.1 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:

```

>>> from diofant.tensor import IndexedBase, Idx
>>> from diofant import symbols
>>> M = IndexedBase('M')
>>> i, j = symbols('i j', cls=Idx)
>>> M[i, j]
M[i, j]
  
```

Repeated indices in a product implies a summation, so to express a matrix-vector product in terms of Indexed objects:

```

>>> x = IndexedBase('x')
>>> M[i, j]*x[j]
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:

```

>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> M[i, j].shape
  
```

```
(m, n)
>>> M[i, j].ranges
[(0, m - 1), (0, n - 1)]
```

The above can be compared with the following:

```
>>> A[i, 2, j].shape
(dim1, 2*dim1, dim2)
>>> A[i, 2, j].ranges
[(0, m - 1), None, (0, n - 1)]
```

To analyze the structure of indexed expressions, you can use the methods `get_indices()` and `get_contraction_structure()`:

```
>>> from diofant.tensor import get_indices, get_contraction_structure
>>> get_indices(A[i, j, j]) == ({i}, {})
True
>>> get_contraction_structure(A[i, j, j]) == {(j,): {A[i, j, j]}}
True
```

See the appropriate docstrings for a detailed explanation of the output.

class `diofant.tensor.indexed.Idx`

Represents an integer index as an Integer or integer expression.

There are a number of ways to create an `Idx` object. The constructor takes two arguments:

label An integer or a symbol that labels the index.

range Optionally you can specify a range as either

- Symbol or integer: This is interpreted as a dimension. Lower and upper bounds are set to 0 and range - 1, respectively.
- tuple: The two elements are interpreted as the lower and upper bounds of the range, respectively.

Note: the `Idx` constructor is rather pedantic in that it only accepts integer arguments. The only exception is that you can use `oo` and `-oo` to specify an unbounded range. For all other cases, both label and bounds must be declared as integers, e.g. if `n` is given as an argument then `n.is_integer` must return `True`.

For convenience, if the label is given as a string it is automatically converted to an integer symbol. (Note: this conversion is not done for range or dimension arguments.)

Examples

```
>>> from diofant.tensor import IndexedBase, Idx
>>> from diofant import symbols, oo
>>> n, i, L, U = symbols('n i L U', integer=True)
```

If a string is given for the label an integer Symbol is created and the bounds are both `None`:

```
>>> idx = Idx('qwerty'); idx
qwerty
```

```
>>> idx.lower, idx.upper
(None, None)
```

Both upper and lower bounds can be specified:

```
>>> idx = Idx(i, (L, U)); idx
i
>>> idx.lower, idx.upper
(L, U)
```

When only a single bound is given it is interpreted as the dimension and the lower bound defaults to 0:

```
>>> idx = Idx(i, n); idx.lower, idx.upper
(0, n - 1)
>>> idx = Idx(i, 4); idx.lower, idx.upper
(0, 3)
>>> idx = Idx(i, oo); idx.lower, idx.upper
(0, oo)
```

The label can be a literal integer instead of a string/Symbol:

```
>>> idx = Idx(2, n); idx.lower, idx.upper
(0, n - 1)
>>> idx.label
2
```

label

Returns the label (Integer or integer expression) of the Idx object.

Examples

```
>>> from diofant import Idx, Symbol
>>> Idx(2).label
2
>>> j = Symbol('j', integer=True)
>>> Idx(j).label
j
>>> Idx(j + 1).label
j + 1
```

lower

Returns the lower bound of the Index.

Examples

```
>>> from diofant import Idx
>>> Idx('j', 2).lower
0
>>> Idx('j', 5).lower
0
>>> Idx('j').lower is None
True
```

upper

Returns the upper bound of the Index.

Examples

```
>>> from diofant import Idx
>>> Idx('j', 2).upper
1
>>> Idx('j', 5).upper
4
>>> Idx('j').upper is None
True
```

class diofant.tensor.indexed.Indexed

Represents a mathematical object with indices.

```
>>> from diofant.tensor import Indexed, IndexedBase, Idx
>>> from diofant import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j)
A[i, j]
```

It is recommended that Indexed objects are created via IndexedBase:

```
>>> A = IndexedBase('A')
>>> Indexed('A', i, j) == A[i, j]
True
```

base

Returns the IndexedBase of the Indexed object.

Examples

```
>>> from diofant.tensor import Indexed, IndexedBase, Idx
>>> from diofant import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).base
A
>>> B = IndexedBase('B')
>>> B == B[i, j].base
True
```

indices

Returns the indices of the Indexed object.

Examples

```
>>> from diofant.tensor import Indexed, Idx
>>> from diofant import symbols
>>> i, j = symbols('i j', cls=Idx)
>>> Indexed('A', i, j).indices
(i, j)
```


ranges

Returns a list of tuples with lower and upper range of each index.

If an index does not define the data members upper and lower, the corresponding slot in the list contains None instead of a tuple.

Examples

```
>>> from diofant import Indexed,Idx, symbols
>>> Indexed('A', Idx('i', 2), Idx('j', 4), Idx('k', 8)).ranges
[(0, 1), (0, 3), (0, 7)]
>>> Indexed('A', Idx('i', 3), Idx('j', 3), Idx('k', 3)).ranges
[(0, 2), (0, 2), (0, 2)]
>>> x, y, z = symbols('x y z', integer=True)
>>> Indexed('A', x, y, z).ranges
[None, None, None]
```

rank

Returns the rank of the Indexed object.

Examples

```
>>> from diofant.tensor import Indexed, Idx
>>> from diofant import symbols
>>> i, j, k, l, m = symbols('i:m', cls=Idx)
>>> Indexed('A', i, j).rank
2
>>> q = Indexed('A', i, j, k, l, m)
>>> q.rank
5
>>> q.rank == len(q.indices)
True
```

shape

Returns a list with dimensions of each index.

Dimensions is a property of the array, not of the indices. Still, if the IndexedBase does not define a shape attribute, it is assumed that the ranges of the indices correspond to the shape of the array.

```
>>> from diofant.tensor.indexed import IndexedBase, Idx
>>> from diofant import symbols
>>> n, m = symbols('n m', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', m)
>>> A = IndexedBase('A', shape=(n, n))
>>> B = IndexedBase('B')
>>> A[i, j].shape
(n, n)
>>> B[i, j].shape
(m, m)
```

class diofant.tensor.indexed.IndexedBase

Represent the base or stem of an indexed object

The `IndexedBase` class represent an array that contains elements. The main purpose of this class is to allow the convenient creation of objects of the `Indexed` class. The `__getitem__` method of `IndexedBase` returns an instance of `Indexed`. Alone, without indices, the `IndexedBase` class can be used as a notation for e.g. matrix equations, resembling what you could do with the `Symbol` class. But, the `IndexedBase` class adds functionality that is not available for `Symbol` instances:

- An `IndexedBase` object can optionally store shape information. This can be used in to check array conformance and conditions for numpy broadcasting. (TODO)
- An `IndexedBase` object implements syntactic sugar that allows easy symbolic representation of array operations, using implicit summation of repeated indices.
- The `IndexedBase` object symbolizes a mathematical structure equivalent to arrays, and is recognized as such for code generation and automatic compilation and wrapping.

```
>>> from diofant.tensor import IndexedBase, Idx
>>> from diofant import symbols
>>> 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!)

```
>>> m, n, o, p = symbols('m n o p', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> A[i, j].shape
(m, n)
>>> B = IndexedBase('B', shape=(o, p))
>>> B[i, j].shape
(o, p)
```

args

Returns the arguments used to create this `IndexedBase` object.

Examples

```
>>> from diofant import IndexedBase
>>> from diofant.abc import x, y
>>> IndexedBase('A', shape=(x, y)).args
(A, (x, y))
```

label

Returns the label of the `IndexedBase` object.

Examples

```
>>> from diofant import IndexedBase
>>> from diofant.abc import x, y
>>> IndexedBase('A', shape=(x, y)).label
A
```

shape

Returns the shape of the IndexedBase object.

Examples

```
>>> from diofant import IndexedBase, Idx, Symbol
>>> from diofant.abc import x, y
>>> IndexedBase('A', shape=(x, y)).shape
(x, y)
```

Note: If the shape of the IndexedBase is specified, it will override any shape information given by the indices.

```
>>> A = IndexedBase('A', shape=(x, y))
>>> B = IndexedBase('B')
>>> i = Idx('i', 2)
>>> j = Idx('j', 1)
>>> A[i, j].shape
(x, y)
>>> B[i, j].shape
(2, 1)
```

3.28.2 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.

`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

```
>>> from diofant.tensor.index_methods import get_contraction_structure
>>> from diofant import symbols, default_sort_key
>>> from diofant.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, k, l = map(Idx, ['i', 'j', 'k', 'l'])
>>> get_contraction_structure(x[i]*y[i] + A[j, j]) == {(i,): {x[i]*y[i]}, (j,):
↳ {A[j, j]}}
True
>>> get_contraction_structure(x[i]*y[j]) == {None: {x[i]*y[j]}}
True
```

A multiplication of contracted factors results in nested dicts representing the internal contractions.

```
>>> d = get_contraction_structure(x[i, i]*y[j, j])
>>> sorted(d.keys(), key=default_sort_key)
[None, x[i, i]*y[j, j]]
```

In this case, the product has no contractions:

```
>>> d[None] == {x[i, i]*y[j, j]}
True
```

Factors are contracted “first”:

```
>>> sorted(d[x[i, i]*y[j, j]], key=default_sort_key) == [{(i,): {x[i, i]}}, {(j,):
↳ {y[j, j]}}]
True
```

A parenthesized Add object is also returned as a nested dictionary. The term containing the parenthesis is a `Mul` with a contraction among the arguments, so it will be found as a key in the result. It stores the dictionary resulting from a recursive call on the Add expression.

```
>>> d = get_contraction_structure(x[i]*(y[i] + A[i, j]*x[j]))
>>> sorted(d.keys(), key=default_sort_key)
[(x[j]*A[i, j] + y[i])*x[i], (i,)]
>>> d[(i,)] == {(x[j]*A[i, j] + y[i])*x[i]}
True
```

```
>>> d[x[i]*(A[i, j]*x[j] + y[i])] ==[{None: {y[i]}, (j,): {x[j]*A[i, j]}}]
True
```

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]}
True
>>> nested_contractions = d[A[j, j]**A[i, i]]
>>> nested_contractions[0] == {(j,): {A[j, j]}}
True
>>> nested_contractions[1] == {(i,): {A[i, i]}}
True
```

The description of the contraction structure may appear complicated when represented with a string in the above examples, but it is easy to iterate over:

```
>>> from diofant import Expr
>>> for key in d:
...     if isinstance(key, Expr):
...         continue
...     for term in d[key]:
...         if term in d:
...             # treat deepest contraction first
...             pass
...     # treat outermost contractions here
```

`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

```
>>> from diofant.tensor.index_methods import get_indices
>>> from diofant import symbols
>>> from diofant.tensor import IndexedBase, Idx
>>> x, y, A = map(IndexedBase, ['x', 'y', 'A'])
>>> i, j, a, z = symbols('i j a z', integer=True)
```

The indices of the total expression is determined, Repeated indices imply a summation, for instance the trace of a matrix A:

```
>>> get_indices(A[i, i]) == (set(), {})
True
```

In the case of many terms, the terms are required to have identical outer indices. Else an `IndexConformanceException` is raised.

```
>>> get_indices(x[i] + A[i, j]*y[j]) == ({i}, {})
True
```

Exceptions

An `IndexConformanceException` means that the terms are not compatible, e.g.

```
>>> get_indices(x[i] + y[j])
Traceback (most recent call last):
...
IndexConformanceException: Indices are not consistent: x(i) + y(j)
```

Warning: The concept of *outer* indices applies recursively, starting on the deepest level. This implies that dummies inside parenthesis are assumed to be summed first, so that the following expression is handled gracefully:

```
>>> get_indices((x[i] + A[i, j]*y[j])*x[j]) == ({i, j}, {})
True
```

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.

3.28.3 Tensor

class `diofant.tensor.tensor._TensorManager`

Class to manage tensor properties.

Notes

Tensors belong to tensor commutation groups; each group has a label `comm`; there are predefined labels:

0 tensors commuting with any other tensor

1 tensors anticommuting among themselves

2 tensors not commuting, apart with those with `comm=0`

Other groups can be defined using `set_comm`; tensors in those groups commute with those with `comm=0`; by default they do not commute with any other group.

clear()

Clear the `TensorManager`.

comm_i2symbol(i)

Returns the symbol corresponding to the commutation group number.

comm_symbols2i(i)

get the commutation group number corresponding to `i`

`i` can be a symbol or a number or a string

If `i` is not already defined its commutation group number is set.

get_comm(i, j)

Return the commutation parameter for commutation group numbers `i, j`

see `_TensorManager.set_comm`

set_comm(*i, j, c*)

set the commutation parameter *c* for commutation groups *i, j*

Parameters *i, j* : symbols representing commutation groups

c : group commutation number

Notes

i, j can be symbols, strings or numbers, apart from 0, 1 and 2 which are reserved respectively for commuting, anticommuting tensors and tensors not commuting with any other group apart with the commuting tensors. For the remaining cases, use this method to set the commutation rules; by default *c*=None.

The group commutation number *c* is assigned in correspondence to the group commutation symbols; it can be

0 commuting

1 anticommuting

None no commutation property

Examples

G and GH do not commute with themselves and commute with each other; A is commuting.

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead, TensorManager
>>> Lorentz = TensorIndexType('Lorentz')
>>> i0,i1,i2,i3,i4 = tensor_indices('i0:5', Lorentz)
>>> A = tensorhead('A', [Lorentz], [[1]])
>>> G = tensorhead('G', [Lorentz], [[1]], 'Gcomm')
>>> GH = tensorhead('GH', [Lorentz], [[1]], 'GHcomm')
>>> TensorManager.set_comm('Gcomm', 'GHcomm', 0)
>>> (GH(i1)*G(i0)).canon_bp()
G(i0)*GH(i1)
>>> (G(i1)*G(i0)).canon_bp()
G(i1)*G(i0)
>>> (G(i1)*A(i0)).canon_bp()
A(i0)*G(i1)
```

set_comms(**args*)

set the commutation group numbers *c* for symbols *i, j*

Parameters *args* : sequence of (*i, j, c*)

class diofant.tensor.tensor.**TensorIndexType**

A **TensorIndexType** is characterized by its name and its metric.

Parameters *name* : name of the tensor type

metric : metric symmetry or metric object or None

dim : dimension, it can be a symbol or an integer or None

eps_dim : dimension of the epsilon tensor

dummy_fmt : name of the head of dummy indices

Notes

The metric parameter can be: `metric = False` symmetric metric (in Riemannian geometry)

`metric = True` antisymmetric metric (for spinor calculus)

`metric = None` there is no metric

`metric` can be an object having `name` and `antisym` attributes.

If there is a metric the metric is used to raise and lower indices.

In the case of antisymmetric metric, the following raising and lowering conventions will be adopted:

$\psi(a) = g(a,b)\psi(-b)$; $\chi(-a) = \chi(b)g(-b,-a)$

$g(-a,b) = \delta(-a,b)$; $g(b,-a) = -\delta(a,-b)$

where $\delta(-a,b) = \delta(b,-a)$ is the Kronecker delta (see `TensorIndex` for the conventions on indices).

If there is no metric it is not possible to raise or lower indices; e.g. the index of the defining representation of $SU(N)$ is 'covariant' and the conjugate representation is 'contravariant'; for $N > 2$ they are linearly independent.

`eps_dim` is by default equal to `dim`, if the latter is an integer; else it can be assigned (for use in naive dimensional regularization); if `eps_dim` is not an integer `epsilon` is `None`.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType
>>> from diofant.printing import sstr
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> Lorentz.metric
metric(Lorentz,Lorentz)
```

Examples with metric components data added, this means it is working on a fixed basis:

```
>>> Lorentz.data = [1, -1, -1, -1]
>>> print(sstr(Lorentz))
TensorIndexType(Lorentz, 0)
>>> print(sstr(Lorentz.data))
[[1 0 0 0]
 [0 -1 0 0]
 [0 0 -1 0]
 [0 0 0 -1]]
```


Attributes

name	
metric_name	(it is 'metric' or metric.name)
metric_antisym	
metric	(the metric tensor)
delta	(Kronecker delta)
epsilon	(the Levi-Civita epsilon tensor)
dim	
dim_eps	
dummy_fmt	
data	(a property to add ndarray values, to work in a specified basis.)

class diofant.tensor.tensor.TensorIndex

Represents an abstract tensor index.

Parameters name : name of the index, or True if you want it to be automatically assigned

tensor_type : TensorIndexType of the index

is_up : flag for contravariant index

Notes

Tensor indices are contracted with the Einstein summation convention.

An index can be in contravariant or in covariant form; in the latter case it is represented prepending a - to the index name.

Dummy indices have a name with head given by `tensor_type._dummy_fmt`

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, TensorIndex, _
↳ TensorSymmetry, TensorType, get_symmetric_group_sgs
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i = TensorIndex('i', Lorentz); i
i
>>> sym1 = TensorSymmetry(*get_symmetric_group_sgs(1))
>>> S1 = TensorType([Lorentz], sym1)
>>> A, B = S1('A,B')
>>> A(i)*B(-i)
A(L_0)*B(-L_0)
```

If you want the index name to be automatically assigned, just put True in the name field, it will be generated using the reserved character `_` in front of its name, in order to avoid conflicts with possible existing indices:

```
>>> i0 = TensorIndex(True, Lorentz)
>>> i0
_i0
>>> i1 = TensorIndex(True, Lorentz)
>>> i1
_i1
>>> A(i0)*B(-i1)
```

```
A(_i0)*B(-_i1)
>>> A(i0)*B(-i0)
A(L_0)*B(-L_0)
```

Attributes

name	
tensor_type	
is_up	

`diofant.tensor.tensor.tensor_indices(s, typ)`

Returns list of tensor indices given their names and their types

Parameters `s` : string of comma separated names of indices

typ : list of `TensorIndexType` of the indices

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
```

class `diofant.tensor.tensor.TensorSymmetry`

Monoterm symmetry of a tensor

Parameters `bsgs` : tuple (base, sgs) BSGS of the symmetry of the tensor

See also:

[*diofant.combinatorics.tensor_can.get_symmetric_group_sgs*](#) (page 249)

Notes

A tensor can have an arbitrary monoterm symmetry provided by its BSGS. Multiterm symmetries, like the cyclic symmetry of the Riemann tensor, are not covered.

Examples

Define a symmetric tensor

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, TensorSymmetry, TensorType, get_symmetric_group_sgs
↳ TensorSymmetry, TensorType, get_symmetric_group_sgs
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = TensorSymmetry(get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Attributes

base	(base of the BSGS)
generators	(generators of the BSGS)
rank	(rank of the tensor)

`diofant.tensor.tensor.tensorsymmetry(*args)`
Return a TensorSymmetry object.

One can represent a tensor with any monoterms slot symmetry group using a BSGS.

`args` can be a BSGS `args[0]` base `args[1]` sgs

Usually tensors are in (direct products of) representations of the symmetric group; `args` can be a list of lists representing the shapes of Young tableaux

Notes

For instance: `[[1]]` vector `[[1]*n]` symmetric tensor of rank `n` `[[n]]` antisymmetric tensor of rank `n` `[[2,2]]` monoterms slot symmetry of the Riemann tensor `[[1],[1]]` vector*vector `[[2],[1],[1]]` (antisymmetric tensor)*vector*vector

Notice that with the shape `[2,2]` we associate only the monoterms symmetries of the Riemann tensor; this is an abuse of notation, since the shape `[2,2]` corresponds usually to the irreducible representation characterized by the monoterms symmetries and by the cyclic symmetry.

Examples

Symmetric tensor using a Young tableau

```
>>> from diofant.tensor.tensor import TensorIndexType, TensorType, tensorsymmetry
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Symmetric tensor using a BSGS (base, strong generator set)

```
>>> from diofant.tensor.tensor import TensorSymmetry, get_symmetric_group_sgs
>>> sym2 = tensorsymmetry(*get_symmetric_group_sgs(2))
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

class `diofant.tensor.tensor.TensorType`

Class of tensor types.

Parameters `index_types` : list of `TensorIndexType` of the tensor indices

`symmetry` : `TensorSymmetry` of the tensor

Examples

Define a symmetric tensor

```
>>> from diofant.tensor.tensor import TensorIndexType, tensorsymmetry, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1, 1])
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> V = S2('V')
```

Attributes

index_types	
symmetry	
types	(list of TensorIndexType without repetitions)

class diofant.tensor.tensor.TensorHead

Tensor head of the tensor

Parameters name : name of the tensor

typ : list of TensorIndexType

comm : commutation group number

Notes

A TensorHead belongs to a commutation group, defined by a symbol on number comm (see `_TensorManager.set_comm`); tensors in a commutation group have the same commutation properties; by default comm is 0, the group of the commuting tensors.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensorhead, TensorType
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> A = tensorhead('A', [Lorentz, Lorentz], [[1],[1]])
```

Examples with ndarray values, the components data assigned to the TensorHead object are assumed to be in a fully-contravariant representation. In case it is necessary to assign components data which represents the values of a non-fully covariant tensor, see the other examples.

```
>>> from diofant.tensor.tensor import tensor_indices, tensorhead
>>> from diofant.printing import sstr
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+2*i for j in range(4)] for i in range(4)]
```

in order to retrieve data, it is also necessary to specify abstract indices enclosed by round brackets, then numerical indices inside square brackets.

```
>>> A(i0, i1)[0, 0]
0
>>> A(i0, i1)[2, 3] == 3+2*2
True
```

Notice that square brackets create a valued tensor expression instance:

```
>>> A(i0, i1)
A(i0, i1)
```

To view the data, just type:

```
>>> print(sstr(A.data))
[[0 1 2 3]
 [2 3 4 5]
 [4 5 6 7]
 [6 7 8 9]]
```

Turning to a tensor expression, covariant indices get the corresponding components data corrected by the metric:

```
>>> print(sstr(A(i0, -i1).data))
[[0 -1 -2 -3]
 [2 -3 -4 -5]
 [4 -5 -6 -7]
 [6 -7 -8 -9]]
```

```
>>> print(sstr(A(-i0, -i1).data))
[[0 -1 -2 -3]
 [-2 3 4 5]
 [-4 5 6 7]
 [-6 7 8 9]]
```

while if all indices are contravariant, the ndarray remains the same

```
>>> print(sstr(A(i0, i1).data))
[[0 1 2 3]
 [2 3 4 5]
 [4 5 6 7]
 [6 7 8 9]]
```

When all indices are contracted and components data are added to the tensor, accessing the data will return a scalar, no numpy object. In fact, numpy ndarrays are dropped to scalars if they contain only one element.

```
>>> A(i0, -i0)
A(L_0, -L_0)
>>> A(i0, -i0).data
-18
```

It is also possible to assign components data to an indexed tensor, i.e. a tensor with specified covariant and contravariant components. In this example, the covariant components data of the Electromagnetic tensor are injected into A :

```
>>> from diofant import symbols
>>> Ex, Ey, Ez, Bx, By, Bz = symbols('E_x E_y E_z B_x B_y B_z')
>>> c = symbols('c', positive=True)
```

Let's define F , an antisymmetric tensor, we have to assign an antisymmetric matrix to it, because $[[2]]$ stands for the Young tableau representation of an antisymmetric set of two elements:

```
>>> F = tensorhead('A', [Lorentz, Lorentz], [[2]])
>>> F(-i0, -i1).data = [
```

```
... [0, Ex/c, Ey/c, Ez/c],
... [-Ex/c, 0, -Bz, By],
... [-Ey/c, Bz, 0, -Bx],
... [-Ez/c, -By, Bx, 0]]
```

Now it is possible to retrieve the contravariant form of the Electromagnetic tensor:

```
>>> print(sstr(F(i0, i1).data))
[[0 -E_x/c -E_y/c -E_z/c]
 [E_x/c 0 -B_z B_y]
 [E_y/c B_z 0 -B_x]
 [E_z/c -B_y B_x 0]]
```

and the mixed contravariant-covariant form:

```
>>> print(sstr(F(i0, -i1).data))
[[0 E_x/c E_y/c E_z/c]
 [E_x/c 0 B_z -B_y]
 [E_y/c -B_z 0 B_x]
 [E_z/c B_y -B_x 0]]
```

To convert the numpy's ndarray to a diofant matrix, just cast:

```
>>> from diofant import Matrix
>>> Matrix(F.data)
Matrix([
 [ 0, -E_x/c, -E_y/c, -E_z/c],
 [E_x/c, 0, -B_z, B_y],
 [E_y/c, B_z, 0, -B_x],
 [E_z/c, -B_y, B_x, 0]])
```

Still notice, in this last example, that accessing components data from a tensor without specifying the indices is equivalent to assume that all indices are contravariant.

It is also possible to store symbolic components data inside a tensor, for example, define a four-momentum-like tensor:

```
>>> from diofant import symbols
>>> P = tensorhead('P', [Lorentz], [[1]])
>>> E, px, py, pz = symbols('E p_x p_y p_z', positive=True)
>>> P.data = [E, px, py, pz]
```

The contravariant and covariant components are, respectively:

```
>>> print(sstr(P(i0).data))
[E p_x p_y p_z]
>>> print(sstr(P(-i0).data))
[E -p_x -p_y -p_z]
```

The contraction of a 1-index tensor by itself is usually indicated by a power by two:

```
>>> P(i0)**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

As the power by two is clearly identical to $P_\mu P^\mu$, it is possible to simply contract the TensorHead object, without specifying the indices

```
>>> p**2
E**2 - p_x**2 - p_y**2 - p_z**2
```

Attributes

name	
index_types	
rank	
types	(equal to <code>typ.types</code>)
symmetry	(equal to <code>typ.symmetry</code>)
comm	(commutation group)

`commutes_with`(*other*)

Returns 0 if self and other commute, 1 if they anticommute.

Returns None if self and other neither commute nor anticommute.

`class diofant.tensor.tensor.TensExpr`

Abstract base class for tensor expressions

Notes

A tensor expression is an expression formed by tensors; currently the sums of tensors are distributed.

A `TensExpr` can be a `TensAdd` or a `TensMul`.

`TensAdd` objects are put in canonical form using the Butler-Portugal algorithm for canonicalization under monotermin symmetries.

`TensMul` objects are formed by products of component tensors, and include a coefficient, which is a Diofant expression.

In the internal representation contracted indices are represented by $(i_{pos1}, i_{pos2}, i_{comp1}, i_{comp2})$, where i_{comp1} is the position of the component tensor with contravariant index, i_{pos1} is the slot which the index occupies in that component tensor.

Contracted indices are therefore nameless in the internal representation.

`get_matrix`()

Returns ndarray components data as a matrix, if components data are available and ndarray dimension does not exceed 2.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensorsymmetry, TensorType
↳ TensorType
>>> from diofant import ones
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> sym2 = tensorsymmetry([1]*2)
>>> S2 = TensorType([Lorentz]*2, sym2)
>>> A = S2('A')
```

The tensor A is symmetric in its indices, as can be deduced by the [1,1] Young tableau when constructing $sym2$. One has to be careful to assign symmetric component data to A , as the symmetry properties of data are currently not checked to be compatible with the defined tensor symmetry.

```
>>> from diofant.tensor.tensor import tensor_indices, tensorhead
>>> Lorentz.data = [1, -1, -1, -1]
>>> i0, i1 = tensor_indices('i0:2', Lorentz)
>>> A.data = [[j+i for j in range(4)] for i in range(4)]
>>> A(i0, i1).get_matrix()
Matrix([
[0, 1, 2, 3],
[1, 2, 3, 4],
[2, 3, 4, 5],
[3, 4, 5, 6]])
```

It is possible to perform usual operation on matrices, such as the matrix multiplication:

```
>>> A(i0, i1).get_matrix()*ones(4, 1)
Matrix([
[ 6],
[10],
[14],
[18]])
```

```
>>> del A.data
```

class diofant.tensor.tensor.TensAdd

Sum of tensors

Parameters `free_args` : list of the free indices

Notes

Sum of more than one tensor are put automatically in canonical form.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensorhead, tensor_indices
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b = tensor_indices('a,b', Lorentz)
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(a) + q(a); t
p(a) + q(a)
>>> t(b)
p(b) + q(b)
```

Examples with components data added to the tensor expression:

```
>>> from diofant import eye
>>> Lorentz.data = [1, -1, -1, -1]
>>> a, b = tensor_indices('a, b', Lorentz)
>>> p.data = [2, 3, -2, 7]
>>> q.data = [2, 3, -2, 7]
```



```
>>> t = p(a) + q(a); t
p(a) + q(a)
>>> t(b)
p(b) + q(b)
```

The following are: $2^{**2} - 3^{**2} - 2^{**2} - 7^{**2} ==> -58$

```
>>> (p(a)*p(-a)).data
-58
>>> p(a)**2
-58
```

Attributes

args	(tuple of addends)
rank	(rank of the tensor)
free_args	(list of the free indices in sorted order)

canon_bp()

canonicalize using the Butler-Portugal algorithm for canonicalization under monoterm symmetries.

contract_metric(*g*)

Raise or lower indices with the metric *g*

Parameters *g* : metric

contract_all : if True, eliminate all *g* which are contracted

Notes

see the TensorIndexType docstring for the contraction conventions

static from_TIDS_list(*coeff*, *tids_list*)

Given a list of coefficients and a list of TIDS objects, construct a TensAdd instance, equivalent to the one that would result from creating single instances of TensMul and then adding them.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead, TensAdd
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j = tensor_indices('i,j', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> eA = 3*A(i, j)
>>> eB = 2*B(j, i)
>>> t1 = eA._tids
>>> t2 = eB._tids
>>> c1 = eA.coeff
>>> c2 = eB.coeff
>>> TensAdd.from_TIDS_list([c1, c2], [t1, t2])
2*B(i, j) + 3*A(i, j)
```

If the coefficient parameter is a scalar, then it will be applied as a coefficient on all TIDS objects.

```
>>> TensAdd.from_TIDS_list(4, [t1, t2])
4*A(i, j) + 4*B(i, j)
```

fun_eval(**index_tuples*)

Return a tensor with free indices substituted according to *index_tuples*

Parameters *index_types* : list of tuples (old_index,new_index)

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j) + A(i, -j)
>>> t.fun_eval((i, k), (-j, l))
A(k, L_0)*B(l, -L_0) + A(k, l)
```

substitute_indices(**index_tuples*)

Return a tensor with free indices substituted according to *index_tuples*

Parameters *index_types* : list of tuples (old_index,new_index)

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.substitute_indices((i,j), (j, k))
A(j, L_0)*B(-L_0, -k)
```

class diofant.tensor.tensor.TensMul

Product of tensors

Parameters *coeff* : Diofant coefficient of the tensor

args

Notes

args[0] list of TensorHead of the component tensors.

args[1] list of (ind,ipos,icomp) where ind is a free index, ipos is the slot position of ind in the icomp-th component tensor.

args[2] list of tuples representing dummy indices. (ipos1,ipos2,icomp1,icomp2) indicates that the contravariant dummy index is the ipos1-th slot position in the icomp1-th

component tensor; the corresponding covariant index is in the ipos2 slot position in the icomp2-th component tensor.

Attributes

components	(list of TensorHead of the component tensors)
types	(list of nonrepeated TensorIndexType)
free	(list of (ind, ipos, comp), see Notes)
dum	(list of (ipos1, ipos2, comp1, comp2), see Notes)
ext_rank	(rank of the tensor counting the dummy indices)
rank	(rank of the tensor)
coeff	(Diofant coefficient of the tensor)
free_args	(list of the free indices in sorted order)
is_canon_bp	(True if the tensor in in canonical form)

canon_bp()

Canonicalize using the Butler-Portugal algorithm for canonicalization under monotermin symmetries.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> A = tensorhead('A', [Lorentz]*2, [[2]])
>>> t = A(m0, -m1)*A(m1, -m0)
>>> t.canon_bp()
-A(L_0, L_1)*A(-L_0, -L_1)
>>> t = A(m0, -m1)*A(m1, -m2)*A(m2, -m0)
>>> t.canon_bp()
0
```

contract_metric(g)

Raise or lower indices with the metric g

Parameters g : metric

Notes

see the TensorIndexType docstring for the contraction conventions

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m0)*q(m1)*g(-m0, -m1)
```

```
>>> t.canon_bp()
metric(L_0, L_1)*p(-L_0)*q(-L_1)
>>> t.contract_metric(g).canon_bp()
p(L_0)*q(-L_0)
```

fun_eval(*index_tuples)

Return a tensor with free indices substituted according to `index_tuples`
`index_types` list of tuples (old_index,new_index)

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(i, k)*B(-k, -j); t
A(i, L_0)*B(-L_0, -j)
>>> t.fun_eval((i, k),(-j, l))
A(k, L_0)*B(-L_0, l)
```

get_indices()

Returns the list of indices of the tensor

The indices are listed in the order in which they appear in the component tensors. The dummy indices are given a name which does not collide with the names of the free indices.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> m0, m1, m2 = tensor_indices('m0,m1,m2', Lorentz)
>>> g = Lorentz.metric
>>> p, q = tensorhead('p,q', [Lorentz], [[1]])
>>> t = p(m1)*g(m0,m2)
>>> t.get_indices()
[m1, m0, m2]
```

perm2tensor(g, canon_bp=False)

Returns the tensor corresponding to the permutation `g`

For further details, see the method in TIDS with the same name.

sorted_components()

Returns a tensor with sorted components calling the corresponding method in a TIDS object.

split()

Returns a list of tensors, whose product is `self`

Dummy indices contracted among different tensor components become free indices with the same name as the one used to represent the dummy indices.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, \
↳ tensorhead
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> a, b, c, d = tensor_indices('a,b,c,d', Lorentz)
>>> A, B = tensorhead('A,B', [Lorentz]*2, [[1]*2])
>>> t = A(a,b)*B(-b,c)
>>> t
A(a, L_0)*B(-L_0, c)
>>> t.split()
[A(a, L_0), B(-L_0, c)]
```

`diofant.tensor.tensor.canon_bp(p)`
Butler-Portugal canonicalization

`diofant.tensor.tensor.tensor_mul(*a)`
product of tensors

`diofant.tensor.tensor.riemann_cyclic_replace(t r)`
replace Riemann tensor with an equivalent expression
 $R(m,n,p,q) \rightarrow 2/3*R(m,n,p,q) - 1/3*R(m,q,n,p) + 1/3*R(m,p,n,q)$

`diofant.tensor.tensor.riemann_cyclic(t2)`
replace each Riemann tensor with an equivalent expression satisfying the cyclic identity.
This trick is discussed in the reference guide to Cadabra.

Examples

```
>>> from diofant.tensor.tensor import TensorIndexType, tensor_indices, tensorhead, \
↳ riemann_cyclic
>>> Lorentz = TensorIndexType('Lorentz', dummy_fmt='L')
>>> i, j, k, l = tensor_indices('i,j,k,l', Lorentz)
>>> R = tensorhead('R', [Lorentz]*4, [[2, 2]])
>>> t = R(i,j,k,l)*(R(-i,-j,-k,-l) - 2*R(-i,-k,-j,-l))
>>> riemann_cyclic(t)
0
```

3.29 Utilities

This module contains some general purpose utilities that are used across Diofant.

3.29.1 Autowrap Module

The autowrap module works very well in tandem with the Indexed classes of the *Tensors* (page 1096). 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
>>> from diofant import symbols, IndexedBase, Idx, Eq
>>> A, x, y = map(IndexedBase, ['A', 'x', 'y'])
```

```
>>> m, n = symbols('m n', integer=True)
>>> i = Idx('i', m)
>>> j = Idx('j', n)
>>> instruction = Eq(y[i], A[i, j]*x[j]); instruction
Eq(y[i], 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.

Note: To use the autowrap module it must first be imported

```
>>> from diofant.utilities.autowrap import autowrap
```

This module provides a common interface for different external backends, such as f2py, fwrap, Cython, SWIG(?) etc. (Currently only f2py and Cython are implemented) The goal is to provide access to compiled binaries of acceptable performance with a one-button user interface, i.e.

```
>>> from diofant.abc import x,y
>>> expr = ((x - y)**(25)).expand()
>>> binary_callable = autowrap(expr)
>>> binary_callable(1, 2)
-1.0
```

The callable returned from autowrap() is a binary python function, not a 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()`.

```
>>> from diofant.utilities.autowrap import binary_function
>>> f = binary_function('f', expr)
>>> 2*f(x, y) + y
y + 2*f(x, y)
>>> (2*f(x, y) + y).evalf(2, subs={x: 1, y:2})
0.e-110
```

The idea is that a 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.

```
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=None*)

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 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

```
>>> from diofant.abc import x, y, z
>>> from diofant.utilities.autowrap import autowrap
>>> 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()`.

```
>>> from diofant.abc import x, y
>>> from diofant.utilities.autowrap import binary_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=None)`

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

[R583] (page 1277)

Examples

```
>>> from diofant.utilities.autowrap import ufuncify
>>> from diofant.abc import x, y
>>> import numpy as np
>>> f = ufuncify((x, y), y + x**2)
>>> type(f) is np.ufunc
True
>>> f([1, 2, 3], 2)
array([ 3.,  6., 11.])
>>> f(np.arange(5), 3)
array([ 3.,  4.,  7., 12., 19.])
```

For the F2Py and Cython backends, inputs are required to be equal length 1-dimensional arrays. The F2Py backend will perform type conversion, but the Cython backend will error if the inputs are not of the expected type.

```
>>> f_fortran = ufuncify((x, y), y + x**2, backend='F2Py')
>>> f_fortran(1, 2)
3
>>> f_fortran(numpy.array([1, 2, 3]), numpy.array([1.0, 2.0, 3.0]))
array([2.,  6., 12.])
```

```

>>> f_cython = ufuncify((x, y), y + x**2, backend='Cython')
>>> f_cython(1, 2)
Traceback (most recent call last):
...
TypeError: Argument '_x' has incorrect type (expected numpy.ndarray, got int)
>>> f_cython(numpy.array([1.0]), numpy.array([2.0]))
array([ 3.])

```

3.29.2 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
```

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.**Routine**(*name*, *arguments*, *results*, *local_vars*,
global_vars)

Generic description of evaluation routine for set of expressions.

A CodeGen class can translate instances of this class into code in a particular language. The routine specification covers all the features present in these languages. The CodeGen part must raise an exception when certain features are not present in the target language. For example, multiple return values are possible in Python, but not in C or Fortran. Another example: Fortran and Python support complex numbers, while C does not.

result_variables

Returns a list of OutputArgument, InOutArgument and Result.

If return values are present, they are at the end of the list.

variables

Returns a set of all variables possibly used in the routine.

For routines with unnamed return values, the dummies that may or may not be used will be included in the set.

class diofant.utilities.codegen.**DataType**(*cname*, *fname*, *pyname*, *octname*)

Holds strings for a certain datatype in different languages.

diofant.utilities.codegen.**get_default_datatype**(*expr*)

Derives an appropriate datatype based on the expression.

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.**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.CodeGen` (*project='project'*)

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*)

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.**CCodeGen**(*project='project'*)

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: http://en.wikipedia.org/wiki/Function_prototype

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: http://en.wikipedia.org/wiki/Function_prototype

class `diofant.utilities.codegen.OctaveCodeGen`(*project='project'*)

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*)

Specialized Routine creation for Octave.

```
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 preprocessor instructions. [default: "project"]

to_files : bool, optional

When True, the code will be written to one or more files with the given prefix, otherwise strings with the names and contents of these files are returned. [default: False]

header : bool, optional

When True, a header is written on top of each source file. [default: True]

empty : bool, optional

When True, empty lines are used to structure the code. [default: True]

argument_sequence : iterable, optional

Sequence of arguments for the routine in a preferred order. A CodeGenError is raised if required arguments are missing. Redundant arguments are used without warning. If omitted, arguments will be ordered alphabetically, but with all input arguments first, and then output or in-out arguments.

global_vars : iterable, optional

Sequence of global variables used by the routine. Variables listed here will not show up as function arguments.

Examples

```
>>> from diofant.utilities.codegen import codegen
>>> from diofant.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     ("f", x+y*z), "C", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
    double f_result;
    f_result = x + y*z;
    return f_result;
}
>>> print(h_name)
test.h
```

```
>>> print(c_header)
#ifdef 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
>>> from diofant import Eq
>>> [(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

```
>>> from diofant.utilities.codegen import codegen
>>> from diofant.abc import x, y, z
>>> [(f_name, f_code), header] = codegen(
...     ("f", x+y*z), "F95", header=False, empty=False,
...     argument_sequence=(x, y), global_vars=(z,))
>>> print(f_code)
REAL*8 function f(x, y)
implicit none
REAL*8, intent(in) :: x
REAL*8, intent(in) :: y
f = x + y*z
end function
```

`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.utilities.codegen import make_routine
>>> from diofant.abc import x, y, f, g
>>> from diofant import Eq
>>> r = make_routine('test', [Eq(f, 2*x), Eq(g, x + y)])
>>> [arg.result_var for arg in r.results]
[]
>>> [arg.name for arg in r.arguments]
[x, y, f, g]
>>> [arg.name for arg in r.result_variables]
[f, g]
>>> r.local_vars == set()
True
```

Another more complicated example with a mixture of specified and automatically-assigned names. Also has Matrix output.

```
>>> from diofant import Matrix
>>> r = make_routine('fcn', [x*y, Eq(f, 1), Eq(g, x + g), Matrix([[x, 2]])])
>>> [arg.result_var for arg in r.results]
[result_...]
>>> [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:

```
>>> from diofant.utilities.codegen import (InputArgument, OutputArgument,
...                                         InOutArgument)
>>> [a.name for a in r.arguments if isinstance(a, InputArgument)]
[x, y]
```

```
>>> [a.name for a in r.arguments if isinstance(a, OutputArgument)]
[f, out_...]
>>> [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]
```

3.29.3 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

```
>>> from diofant.utilities.decorator import no_attrs_in_subclass
```

```
>>> 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
```

`diofant.utilities.decorator.public`(*obj*)

Append *obj*'s name to global `__all__` variable (call site).

By using this decorator on functions or classes you achieve the same goal as by filling `__all__` variables manually, you just don't have to repeat yourself (object's name). You also know if object is public at definition site, not at some random location (where `__all__` was set).

Note that in multiple decorator setup (in almost all cases) `@public` decorator must be applied before any other decorators, because it relies on the pointer to object's global

namespace. If you apply other decorators first, `@public` may end up modifying the wrong namespace.

Examples

```
>>> from diofant.utilities.decorator import public
```

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

```
>>> @public
... def some_function():
...     pass
```

```
>>> __all__
['some_function']
```

3.29.4 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

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import multiset_partitions_taocp
>>> # variables components and multiplicities represent the multiset 'abb'
>>> components = 'ab'
>>> multiplicities = [1, 2]
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(list_visitor(state, components) for state in states)
[[['a', 'b', 'b']],
 [['a', 'b'], ['b']],
 [['a'], ['b', 'b']],
 [['a'], ['b'], ['b']]]
```

`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.

```
>>> from diofant.utilities.enumerative import factoring_visitor
>>> from diofant.utilities.enumerative import multiset_partitions_taocp
>>> from diofant import factorint
>>> primes, multiplicities = zip(*factorint(24).items())
>>> primes
(2, 3)
>>> multiplicities
(3, 1)
>>> states = multiset_partitions_taocp(multiplicities)
>>> list(factoring_visitor(state, primes) for state in states)
[[24], [8, 3], [12, 2], [4, 6], [4, 2, 3], [6, 2, 2], [2, 2, 2, 3]]
```

`diofant.utilities.enumerative.list_visitor(state, components)`

Return a list of lists to represent the partition.

Examples

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import multiset_partitions_taocp
>>> states = multiset_partitions_taocp([1, 2, 1])
>>> s = next(states)
>>> list_visitor(s, 'abc') # for multiset 'a b b c'
[['a', 'b', 'b', 'c']]
>>> s = next(states)
>>> list_visitor(s, [1, 2, 3]) # for multiset '1 2 2 3'
[[1, 2, 2], [3]]
```

The approach of the function `multiset_partitions_taocp` is extended and generalized by the class `MultisetPartitionTraverser`.

class 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 [AOCPL] (page 1277)."

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 1138)

References

[AOCPL] (page 1277), [Factorisatio] (page 1277), [Yorgey] (page 1277)

Examples

```
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([4,4,4,2])
127750
>>> m.count_partitions([3,3,3])
686
```

count_partitions(*multiplicities*)

Returns the number of partitions of a multiset whose components have the multiplicities given in *multiplicities*.

For larger counts, this method is much faster than calling one of the enumerators and counting the result. Uses dynamic programming to cut down on the number of nodes actually explored. The dictionary used in order to accelerate the counting process is stored in the `MultisetPartitionTraverser` object and persists across calls. If the the user does not expect to call `count_partitions` for any additional multisets, the object should be cleared to save memory. On the other hand, the cache built up from one count run can significantly speed up subsequent calls to `count_partitions`, so it may be advantageous not to clear the object.

Notes

If one looks at the workings of Knuth's algorithm M [AOCPL] (page 1277), it can be viewed as a traversal of a binary tree of parts. A part has (up to) two children, the left child resulting from the spread operation, and the right child from the decrement operation. The ordinary enumeration of multiset partitions is an in-order traversal of this tree, and with the partitions corresponding to paths from the root to the leaves. The mapping from paths to partitions is a little complicated, since the partition would contain only those parts which are leaves or the parents of a spread link, not those which are parents of a decrement link.

For counting purposes, it is sufficient to count leaves, and this can be done with a recursive in-order traversal. The number of leaves of a subtree rooted at a particular part is a function only of that part itself, so memoizing has the potential to speed up the counting dramatically.

This method follows a computational approach which is similar to the hypothetical memoized recursive function, but with two differences:

1. This method is iterative, borrowing its structure from the other enumerations and maintaining an explicit stack of parts which are in the process of being counted. (There may be multisets which can be counted reasonably quickly by this implementation, but which would overflow the default Python recursion limit with a recursive implementation.)
2. Instead of using the part data structure directly, a more compact key is constructed. This saves space, but more importantly coalesces some parts which would remain separate with physical keys.

Unlike the enumeration functions, there is currently no `_range` version of `count_partitions`. If someone wants to stretch their brain, it should be possible to construct one by memoizing with a histogram of counts rather than a single count, and combining the histograms.

Examples

```
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> m.count_partitions([9,8,2])
288716
>>> m.count_partitions([2,2])
9
>>> del m
```

`enum_all`(*multiplicities*)

Enumerate the partitions of a multiset.

See also:

[*multiset_partitions_taocp*](#) (page 1138) which provides the same result as this method, but is about twice as fast. Hence, `enum_all` is primarily useful for testing. Also see the function for a discussion of states and visitors.

Examples

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_all([2,2])
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
 [['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'a'], ['b'], ['b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']],
 [['a', 'b'], ['a'], ['b']],
 [['a'], ['a'], ['b', 'b']],
 [['a'], ['a'], ['b'], ['b']]]
```

`enum_large`(*multiplicities, lb*)

Enumerate the partitions of a multiset with $lb < \text{num}(\text{parts})$

Equivalent to `enum_range(multiplicities, lb, sum(multiplicities))`

Parameters multiplicities

list of multiplicities of the components of the multiset.

lb

Number of parts in the partition must be greater than this lower bound.

See also:

[enum_all](#) (page 1141), [enum_small](#) (page 1142), [enum_range](#) (page 1142)

Examples

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_large([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a'], ['b'], ['b']],
 [['a', 'b'], ['a'], ['b']],
 [['a'], ['a'], ['b', 'b']],
 [['a'], ['a'], ['b'], ['b']]]
```

enum_range(multiplicities, lb, ub)

Enumerate the partitions of a multiset with $lb < \text{num}(\text{parts}) \leq ub$.

In particular, if partitions with exactly k parts are desired, call with `(multiplicities, k - 1, k)`. This method generalizes `enum_all`, `enum_small`, and `enum_large`.

Examples

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_range([2,2], 1, 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']]]
```

enum_small(multiplicities, ub)

Enumerate multiset partitions with no more than `ub` parts.

Equivalent to `enum_range(multiplicities, 0, ub)`

Parameters multiplicities

list of multiplicities of the components of the multiset.

ub

Maximum number of parts

See also:

[enum_all](#) (page 1141), [enum_large](#) (page 1141), [enum_range](#) (page 1142)

Examples

```
>>> from diofant.utilities.enumerative import list_visitor
>>> from diofant.utilities.enumerative import MultisetPartitionTraverser
>>> m = MultisetPartitionTraverser()
>>> states = m.enum_small([2,2], 2)
>>> list(list_visitor(state, 'ab') for state in states)
[[['a', 'a', 'b', 'b']],
 [['a', 'a', 'b'], ['b']],
 [['a', 'a'], ['b', 'b']],
 [['a', 'b', 'b'], ['a']],
 [['a', 'b'], ['a', 'b']]]
```

The implementation is based, in part, on the answer given to exercise 69, in Knuth [\[AOCP\]](#) (page 1277).

3.29.5 Iterables

variations

variations(seq, n) Returns all the variations of the list of size n.

Has an optional third argument. Must be a boolean value and makes the method return the variations with repetition if set to True, or the variations without repetition if set to False.

Examples::

```
>>> from diofant.utilities.iterables import variations
>>> list(variations([1,2,3], 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> list(variations([1,2,3], 2, True))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

partitions

Although the combinatorics module contains Partition and IntegerPartition classes for investigation and manipulation of partitions, there are a few functions to generate partitions that can be used as low-level tools for routines: partitions and multiset_partitions. The former gives integer partitions, and the latter gives enumerated partitions of elements. There is also a routine kbins that will give a variety of permutations of partitions.

partitions:

```
>>> from 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
>>> [p for p in multiset_partitions(3, 2)]
[[[0, 1], [2]], [[0, 2], [1]], [[0], [1, 2]]]
>>> [p for p in multiset_partitions([1, 1, 1, 2], 2)]
[[[1, 1, 1], [2]], [[1, 1, 2], [1]], [[1, 1], [1, 2]]]
```

kbins:

```
>>> from diofant.utilities.iterables import kbins
>>> def show(k):
...     rv = []
...     for p in k:
...         rv.append(''.join([''.join(j) for j in p]))
...     return sorted(rv)
...
>>> show(kbins("ABCD", 2))
['A,BCD', 'AB,CD', 'ABC,D']
>>> show(kbins("ABC", 2))
['A,BC', 'AB,C']
>>> show(kbins("ABC", 2, ordered=0)) # same as multiset_partitions
['A,BC', 'AB,C', 'AC,B']
>>> show(kbins("ABC", 2, ordered=1))
['A,BC', 'A,CB',
 'B,AC', 'B,CA',
 'C,AB', 'C,BA']
>>> show(kbins("ABC", 2, ordered=10))
['A,BC', 'AB,C', 'AC,B',
 'B,AC', 'BC,A',
 'C,AB']
>>> show(kbins("ABC", 2, ordered=11))
['A,BC', 'A,CB', 'AB,C', 'AC,B',
 'B,AC', 'B,CA', 'BA,C', 'BC,A',
 'C,AB', 'C,BA', 'CA,B', 'CB,A']
```

Docstring

`diofant.utilities.iterables.binary_partitions(n)`

Generates the binary partition of n.

A binary partition consists only of numbers that are powers of two. Each step reduces a 2^{k+1} to 2^k and 2^k . Thus 16 is converted to 8 and 8.

References

[R584] (page 1277)

Examples

```
>>> from diofant.utilities.iterables import binary_partitions
>>> for i in binary_partitions(5):
...     print(i)
...
[4, 1]
[2, 2, 1]
[2, 1, 1, 1]
[1, 1, 1, 1, 1]
```

`diofant.utilities.iterables.bracelets(n, k)`

Wrapper to necklaces to return a free (unrestricted) necklace.

`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 islice, count
>>> 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.capture(func)`
 Return the printed output of `func()`.

`func` should be a function without arguments that produces output with print statements.

```
>>> from diofant.utilities.iterables import capture
>>> from diofant import pprint
>>> from diofant.abc import x
>>> def foo():
...     print('hello world!')
...
>>> 'hello' in capture(foo) # foo, not foo()
True
>>> capture(lambda: pprint(2/x, use_unicode=False))
'2\n-\nx\n'
```

`diofant.utilities.iterables.common_prefix(*seqs)`
 Return the subsequence that is a common start of sequences in `seqs`.

```
>>> from diofant.utilities.iterables import common_prefix
>>> common_prefix(list(range(3)))
[0, 1, 2]
>>> common_prefix(list(range(3)), list(range(4)))
[0, 1, 2]
>>> common_prefix([1, 2, 3], [1, 2, 5])
[1, 2]
>>> common_prefix([1, 2, 3], [1, 3, 5])
[1]
```

`diofant.utilities.iterables.common_suffix(*seqs)`
 Return the subsequence that is a common ending of sequences in `seqs`.

```
>>> from diofant.utilities.iterables import common_suffix
>>> common_suffix(list(range(3)))
[0, 1, 2]
>>> common_suffix(list(range(3)), list(range(4)))
[]
>>> common_suffix([1, 2, 3], [9, 2, 3])
[2, 3]
>>> common_suffix([1, 2, 3], [9, 7, 3])
[3]
```

`diofant.utilities.iterables.dict_merge(*dicts)`
 Merge dictionaries into a single dictionary.

`diofant.utilities.iterables.filter_symbols(iterator, exclude)`
 Only yield elements from `iterator` that do not occur in `exclude`.

Parameters `iterator`: iterable

iterator to take elements from

exclude : iterable

elements to exclude

Returns iterator : iterator

filtered iterator

`diofant.utilities.iterables.flatten(iterable, levels=None, cls=None)`

Recursively denest iterable containers.

```
>>> from diofant.utilities.iterables import flatten
```

```
>>> flatten([1, 2, 3])
[1, 2, 3]
>>> flatten([1, 2, [3]])
[1, 2, 3]
>>> flatten([1, [2, 3], [4, 5]])
[1, 2, 3, 4, 5]
>>> flatten([1.0, 2, (1, None)])
[1.0, 2, 1, None]
```

If you want to denest only a specified number of levels of nested containers, then set `levels` flag to the desired number of levels:

```
>>> ls = [[(-2, -1), (1, 2)], [(0, 0)]]
```

```
>>> flatten(ls, levels=1)
[(-2, -1), (1, 2), (0, 0)]
```

If `cls` argument is specified, it will only flatten instances of that class, for example:

```
>>> from diofant.core import Basic
>>> class MyOp(Basic):
...     pass
...
>>> flatten([MyOp(1, MyOp(2, 3))], cls=MyOp)
[1, 2, 3]
```

adapted from http://kogs-www.informatik.uni-hamburg.de/~meine/python_tricks

`diofant.utilities.iterables.generate_bell(n)`

Return permutations of $[0, 1, \dots, n - 1]$ such that each permutation differs from the last by the exchange of a single pair of neighbors. The $n!$ permutations are returned as an iterator. In order to obtain the next permutation from a random starting permutation, use the `next_trotterjohnson` method of the `Permutation` class (which generates the same sequence in a different manner).

See also:

[diofant.combinatorics.permutations.Permutation.next_trotterjohnson](#)
(page 179)

References

- http://en.wikipedia.org/wiki/Method_ringing
- <http://stackoverflow.com/questions/4856615/recursive-permutation/4857018>
- <http://programminggeeks.com/bell-algorithm-for-permutation/>

- http://en.wikipedia.org/wiki/Steinhaus%E2%80%93Johnson%E2%80%93Trotter_algorithm
- Generating involutions, derangements, and relatives by ECO Vincent Vajnovszki, DMTCS vol 1 issue 12, 2010

Examples

```
>>> from itertools import permutations
>>> from diofant.utilities.iterables import generate_bell
>>> from diofant import zeros, Matrix
```

This is the sort of permutation used in the ringing of physical bells, and does not produce permutations in lexicographical order. Rather, the permutations differ from each other by exactly one inversion, and the position at which the swapping occurs varies periodically in a simple fashion. Consider the first few permutations of 4 elements generated by `permutations` and `generate_bell`:

```
>>> list(permutations(range(4))[:5])
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 2, 1, 3), (0, 2, 3, 1), (0, 3, 1, 2)]
>>> list(generate_bell(4))[:5]
[(0, 1, 2, 3), (0, 1, 3, 2), (0, 3, 1, 2), (3, 0, 1, 2), (3, 0, 2, 1)]
```

Notice how the 2nd and 3rd lexicographical permutations have 3 elements out of place whereas each “bell” permutation always has only two elements out of place relative to the previous permutation (and so the signature (+/-1) of a permutation is opposite of the signature of the previous permutation).

How the position of inversion varies across the elements can be seen by tracing out where the largest number appears in the permutations:

```
>>> m = zeros(4, 24)
>>> for i, p in enumerate(generate_bell(4)):
...     m[:, i] = Matrix([j - 3 for j in list(p)]) # make largest zero
>>> m.print_nonzero('X')
[XXX XXXXXX XXXXXX XXX]
[XX XX XXXX XX XXXX XX XX]
[X XXXX XX XXXX XX XXXX X]
[ XXXXXX XXXXXX XXXXXX ]
```

`diofant.utilities.iterables.generate_derangements(perm)`

Routine to generate unique derangements.

TODO: This will be rewritten to use the ECO operator approach once the permutations branch is in master.

See also:

diofant.functions.combinatorial.factorials.subfactorial (page 346)

Examples

```
>>> from diofant.utilities.iterables import generate_derangements
>>> list(generate_derangements([0, 1, 2]))
[[1, 2, 0], [2, 0, 1]]
>>> list(generate_derangements([0, 1, 2, 3]))
```

```
[[1, 0, 3, 2], [1, 2, 3, 0], [1, 3, 0, 2], [2, 0, 3, 1], [2, 3, 0, 1], [2, 3,
→ 1, 0], [3, 0, 1, 2], [3, 2, 0, 1], [3, 2, 1, 0]]
>>> list(generate_derangements([0, 1, 1]))
[]
```

`diofant.utilities.iterables.generate_involutions(n)`

Generates involutions.

An involution is a permutation that when multiplied by itself equals the identity permutation. In this implementation the involutions are generated using Fixed Points.

Alternatively, an involution can be considered as a permutation that does not contain any cycles with a length that is greater than two.

References

[R585] (page 1277)

Examples

```
>>> from diofant.utilities.iterables import generate_involutions
>>> list(generate_involutions(3))
[(0, 1, 2), (0, 2, 1), (1, 0, 2), (2, 1, 0)]
>>> len(list(generate_involutions(4)))
10
```

`diofant.utilities.iterables.generate_oriented_forest(n)`

This algorithm generates oriented forests.

An oriented graph is a directed graph having no symmetric pair of directed edges. A forest is an acyclic graph, i.e., it has no cycles. A forest can also be described as a disjoint union of trees, which are graphs in which any two vertices are connected by exactly one simple path.

References

[R586] (page 1277), [R587] (page 1277)

Examples

```
>>> from diofant.utilities.iterables import generate_oriented_forest
>>> list(generate_oriented_forest(4))
[[0, 1, 2, 3], [0, 1, 2, 2], [0, 1, 2, 1], [0, 1, 2, 0], [0, 1, 1, 1], [0, 1,
→ 1, 0], [0, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 0]]
```

`diofant.utilities.iterables.group(seq, multiple=True)`

Splits a sequence into a list of lists of equal, adjacent elements.

See also:

`multiset` (page 1152)

Examples

```
>>> from diofant.utilities.iterables import group
```

```
>>> group([1, 1, 1, 2, 2, 3])
[[1, 1, 1], [2, 2], [3]]
>>> group([1, 1, 1, 2, 2, 3], multiple=False)
[(1, 3), (2, 2), (3, 1)]
>>> group([1, 1, 3, 2, 2, 1], multiple=False)
[(1, 2), (3, 1), (2, 2), (1, 1)]
```

`diofant.utilities.iterables.has_dups(seq)`

Return True if there are any duplicate elements in seq.

Examples

```
>>> from diofant.utilities.iterables import has_dups
>>> from diofant import Dict, Set
```

```
>>> has_dups((1, 2, 1))
True
>>> has_dups(range(3))
False
>>> all(has_dups(c) is False for c in (set(), Set(), dict(), Dict()))
True
```

`diofant.utilities.iterables.has_variety(seq)`

Return True if there are any different elements in seq.

Examples

```
>>> from diofant.utilities.iterables import has_variety
```

```
>>> has_variety((1, 2, 1))
True
>>> has_variety((1, 1, 1))
False
```

`diofant.utilities.iterables.ibin(n, bits=0, str=False)`

Return a list of length `bits` corresponding to the binary value of `n` with small bits to the right (last). If `bits` is omitted, the length will be the number required to represent `n`. If the bits are desired in reversed order, use the `[::-1]` slice of the returned list.

If a sequence of all bits-length lists starting from `[0, 0, ..., 0]` through `[1, 1, ..., 1]` are desired, pass a non-integer for `bits`, e.g. `'all'`.

If the bit *string* is desired pass `str=True`.

Examples

```
>>> from diofant.utilities.iterables import ibin
>>> ibin(2)
[1, 0]
>>> ibin(2, 4)
[0, 0, 1, 0]
>>> ibin(2, 4)[::-1]
[0, 1, 0, 0]
```

If all lists corresponding to 0 to $2^{**n} - 1$, pass a non-integer for bits:

```
>>> bits = 2
>>> for i in ibin(2, 'all'):
...     print(i)
(0, 0)
(0, 1)
(1, 0)
(1, 1)
```

If a bit string is desired of a given length, use `str=True`:

```
>>> n = 123
>>> bits = 10
>>> ibin(n, bits, str=True)
'0001111011'
>>> ibin(n, bits, str=True)[::-1] # small bits left
'1101111000'
>>> list(ibin(3, 'all', str=True))
['000', '001', '010', '011', '100', '101', '110', '111']
```

`diofant.utilities.iterables.interactive_traversal(expr)`
Traverse a tree asking a user which branch to choose.

`diofant.utilities.iterables.kbins(l, k, ordered=None)`
Return sequence `l` partitioned into `k` bins.

See also:

[partitions](#) (page 1155), [multiset_partitions](#) (page 1153)

Examples

```
>>> from diofant.utilities.iterables import kbins
```

The default is to give the items in the same order, but grouped into `k` partitions without any reordering:

```
>>> for p in kbins(list(range(5)), 2):
...     print(p)
...
[[0], [1, 2, 3, 4]]
[[0, 1], [2, 3, 4]]
[[0, 1, 2], [3, 4]]
[[0, 1, 2, 3], [4]]
```

The ordered flag which is either None (to give the simple partition of the the elements) or is a 2 digit integer indicating whether the order of the bins and the order of the items in the bins matters. Given:

```
A = [[0], [1, 2]]
B = [[1, 2], [0]]
C = [[2, 1], [0]]
D = [[0], [2, 1]]
```

the following values for ordered have the shown meanings:

```
00 means A == B == C == D
01 means A == B
10 means A == D
11 means A == A
```

```
>>> for ordered in [None, 0, 1, 10, 11]:
...     print('ordered = %s' % ordered)
...     for p in kbins(list(range(3)), 2, ordered=ordered):
...         print('      %s' % p)
...
ordered = None
  [[0], [1, 2]]
  [[0, 1], [2]]
ordered = 0
  [[0, 1], [2]]
  [[0, 2], [1]]
  [[0], [1, 2]]
ordered = 1
  [[0], [1, 2]]
  [[0], [2, 1]]
  [[1], [0, 2]]
  [[1], [2, 0]]
  [[2], [0, 1]]
  [[2], [1, 0]]
ordered = 10
  [[0, 1], [2]]
  [[2], [0, 1]]
  [[0, 2], [1]]
  [[1], [0, 2]]
  [[0], [1, 2]]
  [[1, 2], [0]]
ordered = 11
  [[0], [1, 2]]
  [[0, 1], [2]]
  [[0], [2, 1]]
  [[0, 2], [1]]
  [[1], [0, 2]]
  [[1, 0], [2]]
  [[1], [2, 0]]
  [[1, 2], [0]]
  [[2], [0, 1]]
  [[2, 0], [1]]
  [[2], [1, 0]]
  [[2, 1], [0]]
```

```
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

```
>>> from diofant.combinatorics.polyhedron import minlex
>>> minlex((1, 2, 0))
(0, 1, 2)
>>> minlex((1, 0, 2))
(0, 2, 1)
>>> minlex((1, 0, 2), directed=False)
(0, 1, 2)
```

```
>>> minlex('11010011000', directed=True)
'00011010011'
>>> minlex('11010011000', directed=False)
'00011001011'
```

`diofant.utilities.iterables.multiset(seq)`

Return the hashable sequence in multiset form with values being the multiplicity of the item in the sequence.

See also:

[group](#) (page 1148)

Examples

```
>>> from diofant.utilities.iterables import multiset
>>> multiset('mississippi') == {'i': 4, 'm': 1, 'p': 2, 's': 4}
True
```

`diofant.utilities.iterables.multiset_combinations(m, n, g=None)`

Return the unique combinations of size `n` from multiset `m`.

Examples

```
>>> from diofant.utilities.iterables import multiset_combinations
>>> from itertools import combinations
>>> [''.join(i) for i in multiset_combinations('baby', 3)]
['abb', 'aby', 'bby']
```

```
>>> def count(f, s): return len(list(f(s, 3)))
```

The number of combinations depends on the number of letters; the number of unique combinations depends on how the letters are repeated.

```

>>> s1 = 'abracadabra'
>>> s2 = 'banana tree'
>>> count(combinations, s1), count(multiset_combinations, s1)
(165, 23)
>>> count(combinations, s2), count(multiset_combinations, s2)
(165, 54)

```

`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:

```

>>> from diofant import bell
>>> len(list(multiset_partitions(5))) == bell(5) == 52
True

```

The number of partitions of length `k` from a set of size `n` is given by the Stirling Number of the 2nd kind:

```

>>> def S2(n, k):
...     from diofant import Dummy, binomial, factorial, Sum
...     if k > n:
...         return 0
...     j = Dummy()
...     arg = (-1)**(k-j)*j**n*binomial(k,j)
...     return 1/factorial(k)*Sum(arg,(j,0,k)).doit()
...
>>> S2(5, 2) == len(list(multiset_partitions(5, 2))) == 15
True

```

These comments on counting apply to *sets*, not multisets.

See also:

[partitions](#) (page 1155), [diofant.combinatorics.partitions.Partition](#) (page 158), [diofant.combinatorics.partitions.IntegerPartition](#) (page 159), [diofant.functions.combinatorial.numbers.nT](#) (page 356)

Notes

When all the elements are the same in the multiset, the order of the returned partitions is determined by the `partitions` routine. If one is counting partitions then it is better to use the `nT` function.

Examples

```

>>> from diofant.utilities.iterables import multiset_partitions
>>> list(multiset_partitions([1, 2, 3, 4], 2))
[[[1, 2, 3], [4]], [[1, 2, 4], [3]], [[1, 2], [3, 4]],
 [[1, 3, 4], [2]], [[1, 3], [2, 4]], [[1, 4], [2, 3]],
 [[1], [2, 3, 4]]]

```

```
>>> list(multiset_partitions([1, 2, 3, 4], 1))
[[[1, 2, 3, 4]]]
```

Only unique partitions are returned and these will be returned in a canonical order regardless of the order of the input:

```
>>> a = [1, 2, 2, 1]
>>> ans = list(multiset_partitions(a, 2))
>>> a.sort()
>>> list(multiset_partitions(a, 2)) == ans
True
>>> a = range(3, 1, -1)
>>> (list(multiset_partitions(a)) ==
...  list(multiset_partitions(sorted(a))))
True
```

If *m* is omitted then all partitions will be returned:

```
>>> list(multiset_partitions([1, 1, 2]))
[[[1, 1, 2]], [[1, 1], [2]], [[1, 2], [1]], [[1], [1], [2]]]
>>> list(multiset_partitions([1]*3))
[[[1, 1, 1]], [[1], [1, 1]], [[1], [1], [1]]]
```

`diofant.utilities.iterables.multiset_permutations(m, size=None, g=None)`

Return the unique permutations of multiset *m*.

Examples

```
>>> from diofant.utilities.iterables import multiset_permutations
>>> from diofant import factorial
>>> [''.join(i) for i in multiset_permutations('aab')]
['aab', 'aba', 'baa']
>>> factorial(len('banana'))
720
>>> len(list(multiset_permutations('banana')))
60
```

`diofant.utilities.iterables.necklaces(n, k, free=False)`

A routine to generate necklaces that may (*free=True*) or may not (*free=False*) be turned over to be viewed. The “necklaces” returned are comprised of *n* integers (beads) with *k* different values (colors). Only unique necklaces are returned.

References

<http://mathworld.wolfram.com/Necklace.html>

Examples

```
>>> from diofant.utilities.iterables import necklaces, bracelets
>>> def show(s, i):
...     return ''.join(s[j] for j in i)
```

The “unrestricted necklace” is sometimes also referred to as a “bracelet” (an object that can be turned over, a sequence that can be reversed) and the term “necklace” is used to imply a sequence that cannot be reversed. So $ACB == ABC$ for a bracelet (rotate and reverse) while the two are different for a necklace since rotation alone cannot make the two sequences the same.

(mnemonic: Bracelets can be viewed Backwards, but Not Necklaces.)

```
>>> B = [show('ABC', i) for i in bracelets(3, 3)]
>>> N = [show('ABC', i) for i in necklaces(3, 3)]
>>> set(N) - set(B) == {'ACB'}
True
```

```
>>> list(necklaces(4, 2))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 1),
 (0, 1, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1)]
```

```
>>> [show('.o', i) for i in bracelets(4, 2)]
['....', '...o', '..oo', '.o.o', '.ooo', 'oooo']
```

`diofant.utilities.iterables.numbered_symbols`(*prefix='x', cls=None, start=0, exclude=[], *args, **assumptions*)

Generate an infinite stream of Symbols consisting of a prefix and increasing subscripts provided that they do not occur in *exclude*.

Parameters *prefix* : str, optional

The prefix to use. By default, this function will generate symbols of the form “x0”, “x1”, etc.

cls : class, optional

The class to use. By default, it uses Symbol, but you can also use Wild or Dummy.

start : int, optional

The start number. By default, it is 0.

Returns *sym* : Symbol

The subscripted symbols.

`diofant.utilities.iterables.partitions`(*n, m=None, k=None, size=False*)

Generate all partitions of integer *n* (≥ 0).

Parameters “*m*” : integer (default gives partitions of all sizes)

limits number of parts in partition (mnemonic: *m*, maximum parts)

“*k*” : integer (default gives partitions number from 1 through *n*)

limits the numbers that are kept in the partition (mnemonic: *k*, keys)

“*size*” : bool (default False, only partition is returned)

when True then (*M*, *P*) is returned where *M* is the sum of the multiplicities and *P* is the generated partition.

Each partition is represented as a dictionary, mapping an integer to the number of copies of that integer in the partition. For example, the first partition of 4 returned is {4: 1}, “4: one of them”.

See also:

diofant.combinatorics.partitions.Partition (page 158), *diofant.combinatorics.partitions.IntegerPartition* (page 159)

Notes

Modified from Tim Peter's version [R588] (page 1277) to allow for k and m values.

References

[R588] (page 1277)

Examples

```
>>> from diofant.utilities.iterables import partitions
```

The numbers appearing in the partition (the key of the returned dict) are limited with k:

```
>>> for p in partitions(6, k=2):
...     print(p)
{2: 3}
{1: 2, 2: 2}
{1: 4, 2: 1}
{1: 6}
```

The maximum number of parts in the partition (the sum of the values in the returned dict) are limited with m:

```
>>> for p in partitions(6, m=2):
...     print(p)
...
{6: 1}
{1: 1, 5: 1}
{2: 1, 4: 1}
{3: 2}
```

Note that the `_same_` dictionary object is returned each time. This is for speed: generating each partition goes quickly, taking constant time, independent of n.

```
>>> [p for p in partitions(6, k=2)]
[{1: 6}, {1: 6}, {1: 6}, {1: 6}]
```

If you want to build a list of the returned dictionaries then make a copy of them:

```
>>> [p.copy() for p in partitions(6, k=2)]
[{2: 3}, {1: 2, 2: 2}, {1: 4, 2: 1}, {1: 6}]
>>> [(M, p.copy()) for M, p in partitions(6, k=2, size=True)]
[(3, {2: 3}), (4, {1: 2, 2: 2}), (5, {1: 4, 2: 1}), (6, {1: 6})]
```

`diofant.utilities.iterables.postfixes(seq)`
Generate all postfixes of a sequence.

Examples

```
>>> from diofant.utilities.iterables import postfixes
```

```
>>> list(postfixes([1,2,3,4]))
[[4], [3, 4], [2, 3, 4], [1, 2, 3, 4]]
```

`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.utilities.iterables import postorder_traversal
>>> from diofant.abc import w, x, y, z
```

The nodes are returned in the order that they are encountered unless key is given; simply passing `key=True` will guarantee that the traversal is unique.

```
>>> list(postorder_traversal(w + (x + y)*z, keys=True))
[w, z, x, y, x + y, z*(x + y), w + z*(x + y)]
```

`diofant.utilities.iterables.prefixes(seq)`

Generate all prefixes of a sequence.

Examples

```
>>> from diofant.utilities.iterables import prefixes
```

```
>>> list(prefixes([1,2,3,4]))
[[1], [1, 2], [1, 2, 3], [1, 2, 3, 4]]
```

`diofant.utilities.iterables.reshape(seq, how)`

Reshape the sequence according to the template in `how`.

Examples

```
>>> from diofant.utilities import reshape
>>> seq = list(range(1, 9))
```

```
>>> reshape(seq, [4]) # lists of 4
[[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
>>> reshape(seq, (4,)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, 2)) # tuples of 4
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

```
>>> reshape(seq, (2, [2])) # (i, i, [i, i])
[(1, 2, [3, 4]), (5, 6, [7, 8])]
```

```
>>> reshape(seq, ((2,), [2])) # etc....
[((1, 2), [3, 4]), ((5, 6), [7, 8])]
```

```
>>> reshape(seq, (1, [2], 1))
[(1, [2, 3], 4), (5, [6, 7], 8)]
```

```
>>> reshape(tuple(seq), ([[1], 1, (2,)],))
(((1, 2, (3, 4)),), ([5], 6, (7, 8)),)
```

```
>>> reshape(tuple(seq), ([1], 1, (2,)))
(((1, 2, (3, 4)), ([5], 6, (7, 8))))
```

```
>>> reshape(list(range(12)), [2, [3], {2}, (1, (3,)), 1]) == [[0, 1, [2, 3, 4],
↳ {5, 6}, (7, (8, 9, 10), 11)]]
True
```

`diofant.utilities.iterables.rotate_left(x, y)`
Left rotates a list `x` by the number of steps specified in `y`.

Examples

```
>>> from diofant.utilities.iterables import rotate_left
>>> 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

```
>>> from diofant.utilities.iterables import rotate_right
>>> 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

```
>>> from diofant.utilities.iterables import runs
>>> from operator import ge
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2])
[[0, 1, 2], [2], [1, 4], [3], [2], [2]]
>>> runs([0, 1, 2, 2, 1, 4, 3, 2, 2], op=ge)
[[0, 1, 2, 2], [1, 4], [3], [2, 2]]
```

`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.

See also:

[*diofant.core.compatibility.ordered*](#) (page 153)

Examples

```
>>> from collections import defaultdict
>>> from diofant.utilities import sift
>>> from diofant.abc import x, y
>>> from diofant import sqrt, exp, E
```

```
>>> 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.

`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 $\text{binomial}(n, k)$, whereas there are 2^n subsets all together. If k is None then all 2^n subsets will be returned from shortest to longest.

Examples

```
>>> from diofant.utilities.iterables import subsets
```

`subsets(seq, k)` will return the $n!/k!(n-k)!$ k-subsets (combinations) without repetition, i.e. once an item has been removed, it can no longer be “taken”:

```
>>> list(subsets([1, 2], 2))
[(1, 2)]
>>> list(subsets([1, 2]))
[(), (1,), (2,), (1, 2)]
>>> list(subsets([1, 2, 3], 2))
[(1, 2), (1, 3), (2, 3)]
```

`subsets(seq, k, repetition=True)` will return the $(n-1+k)!/k!(n-1)!$ combinations *with* repetition:

```
>>> list(subsets([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(subsets([0, 1], 3, repetition=False))
[]
>>> list(subsets([0, 1], 3, repetition=True))
[(0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1)]
```

`diofant.utilities.iterables.take(iter, n)`

Return n items from iter iterator.

`diofant.utilities.iterables.topological_sort(graph, key=None)`

Topological sort of graph’s vertices.

Parameters “`graph`”: tuple[list, list[tuple[T, T]]

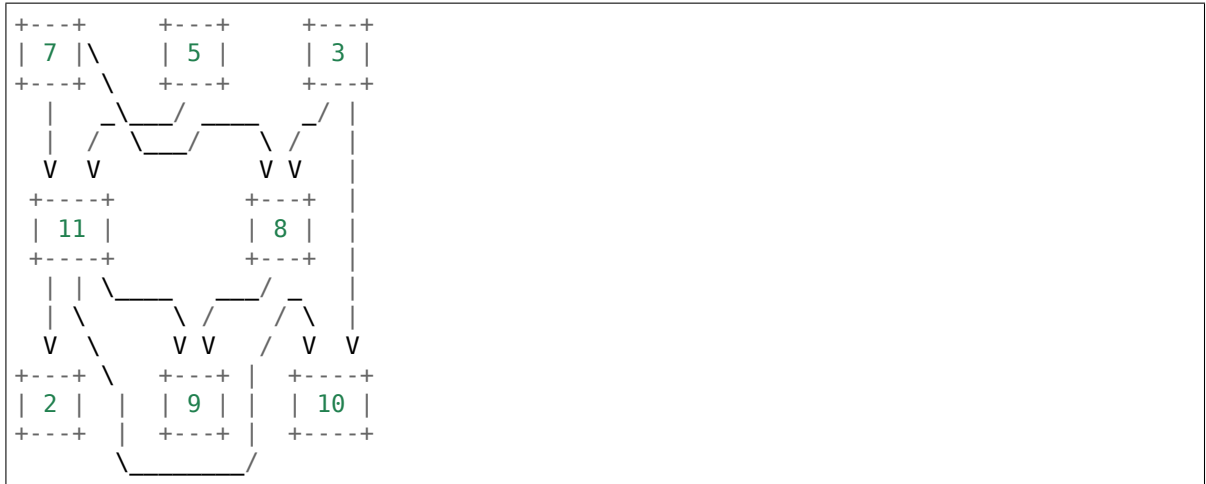
A tuple consisting of a list of vertices and a list of edges of a graph to be sorted topologically.

“`key`”: callable[T] (optional)

Ordering key for vertices on the same level. By default the natural (e.g. lexicographic) ordering is used (in this case the base type must implement ordering relations).

Examples

Consider a graph:



where vertices are integers. This graph can be encoded using elementary Python's data structures as follows:

```
>>> V = [2, 3, 5, 7, 8, 9, 10, 11]
>>> E = [(7, 11), (7, 8), (5, 11), (3, 8), (3, 10),
...      (11, 2), (11, 9), (11, 10), (8, 9)]
```

To compute a topological sort for graph (V,E) issue:

```
>>> from diofant.utilities.iterables import topological_sort
>>> topological_sort((V, E))
[3, 5, 7, 8, 11, 2, 9, 10]
```

If specific tie breaking approach is needed, use key parameter:

```
>>> topological_sort((V, E), key=lambda v: -v)
[7, 5, 11, 3, 10, 8, 9, 2]
```

Only acyclic graphs can be sorted. If the input graph has a cycle, then `ValueError` will be raised:

```
>>> topological_sort((V, E + [(10, 7)]))
Traceback (most recent call last):
...
ValueError: cycle detected
```

See also:

http://en.wikipedia.org/wiki/Topological_sorting

`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

```
>>> from diofant.utilities.iterables import uniq
>>> dat = [1, 4, 1, 5, 4, 2, 1, 2]
>>> type(uniq(dat)) in (list, tuple)
False
```

```
>>> list(uniq(dat))
[1, 4, 5, 2]
>>> list(uniq(x for x in dat))
[1, 4, 5, 2]
>>> list(uniq([[1], [2, 1], [1]]))
[[1], [2, 1]]
```

`diofant.utilities.iterables.variations(seq, n, repetition=False)`

Returns a generator of the n-sized variations of seq (size N). `repetition` controls whether items in seq can appear more than once;

Examples

`variations(seq, n)` will return $N! / (N - n)!$ permutations without repetition of seq's elements:

```
>>> from diofant.utilities.iterables import variations
>>> list(variations([1, 2], 2))
[(1, 2), (2, 1)]
```

`variations(seq, n, True)` will return the N^{*n} permutations obtained by allowing repetition of elements:

```
>>> list(variations([1, 2], 2, repetition=True))
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

If you ask for more items than are in the set you get the empty set unless you allow repetitions:

```
>>> list(variations([0, 1], 3, repetition=False))
[]
>>> list(variations([0, 1], 3, repetition=True))[:4]
[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1)]
```

3.29.6 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

```
>>> from diofant.abc import x
>>> from diofant.utilities.lambdify import lambdify, implemented_function
>>> from diofant import Function
>>> 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

```
>>> from diofant.abc import x, y, z
>>> from diofant.utilities.lambdify import lambdastr
>>> lambdastr(x, x**2)
'lambda x: (x**2)'
>>> lambdastr((x,y,z), [z,y,x])
'lambda x,y,z: ([z, y, x])'
```

Although tuples may not appear as arguments to lambda in Python 3, `lambdastr` will create a lambda function that will unpack the original arguments so that nested arguments can be handled:

```
>>> lambdastr((x, (y, z)), x + y)
'lambda _0,_1: (lambda x,y,z: (x + y))*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, Diofant functions are replaced as far as possible by either `python-math`, `numpy` (if available) or `mpmath` functions - exactly in this order. To change this behavior, the “modules” argument can be used. It accepts:

- the strings “math”, “mpmath”, “numpy”, “numexpr”, “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`.

For functions involving large array calculations, `numexpr` can provide a significant speedup over `numpy`. Please note that the available functions for `numexpr` are more limited than `numpy` but can be expanded with `implemented_function` and user defined subclasses of `Function`. If specified, `numexpr` may be the only option in modules. The official list of `numexpr` functions can be found at: <https://github.com/pydata/numexpr#supported-functions>

In previous releases `lambdify` replaced `Matrix` with `numpy.matrix` by default. As of release 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.

```
>>> from diofant import lambdify, Matrix
>>> from diofant.abc import x, y
>>> import numpy
>>> array2mat = [{'ImmutableMatrix': numpy.matrix}, 'numpy']
>>> f = lambdify((x, y), Matrix([x, y]), modules=array2mat)
>>> f(1, 2)
[[1]
 [2]]
```

1. Use one of the provided modules:

```
>>> from diofant import sin, tan, gamma
>>> from diofant.utilities.lambdify import lambdastr
>>> from diofant.abc import x, y
>>> f = lambdify(x, sin(x), "math")
```

Attention: Functions that are not in the math module will throw a name error when the 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 'sin(%s) is cool' % x
>>> myfuncs = {"sin" : my_cool_function}
>>> f = lambdify(x, sin(x), myfuncs); f(1)
'sin(1) is cool'
```

Examples

```
>>> from diofant.utilities.lambdify import implemented_function
>>> from diofant import sqrt, sin, Matrix
>>> from diofant import Function
>>> from diofant.abc import w, x, y, z
```

```
>>> f = lambdify(x, x**2)
>>> f(2)
4
>>> f = lambdify((x, y, z), [z, y, x])
>>> f(1,2,3)
[3, 2, 1]
>>> f = lambdify(x, sqrt(x))
>>> f(4)
2.0
>>> f = lambdify((x, y), sin(x*y)**2)
>>> f(0, 5)
0.0
>>> row = lambdify((x, y), Matrix((x, x + y)).T, modules='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:

```
>>> from diofant.utilities.iterables import flatten
>>> args = w, (x, (y, z))
>>> vals = 1, (2, (3, 4))
>>> f = lambdify(flatten(args), w + x + y + z)
>>> f(*flatten(vals))
10
```

Functions present in *expr* can also carry their own numerical implementations, in a callable attached to the `_imp_` attribute. Usually you attach this using the `implemented_function` factory:

```
>>> f = implemented_function(Function('f'), lambda x: x+1)
>>> func = lambdify(x, f(x))
>>> func(4)
5
```

`lambdify` always prefers `_imp_` implementations to implementations in other namespaces, unless the `use_imps` input parameter is `False`.

3.29.7 Memoization

`diofant.utilities.memoization.recurrence_memo(initial)`

Memo decorator for sequences defined by recurrence

See usage examples e.g. in the `specfun/combinatorial` module

3.29.8 Miscellaneous

Miscellaneous stuff that doesn't really fit anywhere else.

`diofant.utilities.misc.debug(*args)`

Print `*args` if `DIOFANT_DEBUG` is `True`, else do nothing.

`diofant.utilities.misc.fillededent(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.

3.29.9 Randomised Testing

Helpers for randomized testing

`diofant.utilities.randtest.random_complex_number(a=2, b=-1, c=3, d=1, rational=False)`

Return a random complex number.

To reduce chance of hitting branch cuts or anything, we guarantee $b \leq \text{Im } z \leq d$, $a \leq \text{Re } z \leq c$

`diofant.utilities.randtest.test_derivative_numerically(f, z, tol=1e-06, a=2, b=-1, c=3, d=1)`

Test numerically that the symbolically computed derivative of `f` with respect to `z` is correct.

This routine does not test whether there are Floats present with precision higher than 15 digits so if there are, your results may not be what you expect due to round-off errors.

Examples

```
>>> from diofant import sin
>>> from diofant.abc import x
>>> from diofant.utilities.randtest import test_derivative_numerically as td
>>> td(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

```
>>> from diofant import sin, cos
>>> from diofant.abc import x
>>> from diofant.utilities.randtest import verify_numerically as tn
>>> tn(sin(x)**2 + cos(x)**2, 1, x)
true
```

3.30 Parsing input

3.30.1 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)

See also:

[`diofant.parsing.sympy_parser.stringify_expr`](#) (page 1168),
[`diofant.parsing.sympy_parser.eval_expr`](#) (page 1168), [`diofant.parsing.sympy_parser.standard_transformations`](#) (page 1168), [`diofant.parsing.sympy_parser.implicit_multiplication_application`](#) (page 1170)

Examples

```
>>> from diofant.parsing.sympy_parser import parse_expr
>>> parse_expr("1/2")
1/2
>>> type(_)
<class 'diofant.core.numbers.Half'>
>>> from diofant.parsing.sympy_parser import standard_transformations,\
... implicit_multiplication_application
>>> transformations = (standard_transformations +
... (implicit_multiplication_application,))
>>> parse_expr("2x", transformations=transformations)
2*x
```

When `evaluate=False`, some automatic simplifications will not occur:

```
>>> parse_expr("2**3"), parse_expr("2**3", evaluate=False)
(8, 2**3)
```

In addition the order of the arguments will not be made canonical. This feature allows one to tell exactly how the expression was entered:

```
>>> a = parse_expr('1 + x', evaluate=False)
>>> b = parse_expr('x + 1', evaluate=0)
>>> a == b
False
>>> a.args
(1, x)
>>> b.args
(x, 1)
```

`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)`

3.30.2 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 1167) and are applied in the order given.

`diofant.parsing.sympy_parser.standard_transformations = (<function lambda_notation>, <function`

Standard transformations for `parse_expr()` (page 1167). Inserts calls to `Symbol` (page 81), `Integer` (page 91), 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:

```
>>> from diofant.parsing.sympy_parser import (parse_expr, _tokenSplittable,
... standard_transformations, implicit_multiplication,
... split_symbols_custom)
>>> def can_split(symbol):
...     if symbol not in ('list', 'of', 'unsplittable', 'names'):
...         return _tokenSplittable(symbol)
...     return False
...
>>> transformation = split_symbols_custom(can_split)
>>> parse_expr('unsplittable', transformations=standard_transformations +
... (transformation, implicit_multiplication))
unsplittable
```

`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 1169), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from diofant.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication)
>>> transformations = standard_transformations + (implicit_multiplication,)
>>> parse_expr('3 x y', transformations=transformations)
3*x*y
```

`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 1169), otherwise expressions like `sin 2x` will be parsed as `x * sin(2)` rather than `sin(2*x)`.

Examples

```
>>> from diofant.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_application)
>>> transformations = standard_transformations + (implicit_application,)
>>> parse_expr('cot z + csc z', transformations=transformations)
cot(z) + csc(z)
```

`diofant.parsing.sympy_parser.function_exponentiation(tokens, local_dict, global_dict)`

Allows functions to be exponentiated, e.g. `cos**2(x)`.

Examples

```
>>> from diofant.parsing.sympy_parser import (parse_expr,
... standard_transformations, function_exponentiation)
>>> transformations = standard_transformations + (function_exponentiation,)
>>> parse_expr('sin**4(x)', transformations=transformations)
sin(x)**4
```

`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

```
>>> from diofant.parsing.sympy_parser import (parse_expr,
... standard_transformations, implicit_multiplication_application)
>>> parse_expr("10sin**2 x**2 + 3xyz + tan theta",
... transformations=(standard_transformations +
... (implicit_multiplication_application,)))
3*x*y*z + 10*sin(x**2)**2 + tan(theta)
```

`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` 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`.

3.31 Calculus

Some calculus-related methods waiting to find a better place in the Diofant modules tree.

`diofant.calculus.euler.euler_equations`(*L*, *funcs=()*, *vars=()*)
Find the Euler-Lagrange equations [R1] (page 1277) for a given Lagrangian.

Parameters *L* : Expr

The Lagrangian that should be a function of the functions listed in the second argument and their derivatives.

For example, in the case of two functions $f(x, y)$, $g(x, y)$ and two independent variables x, y the Lagrangian would have the form:

$$L\left(f(x, y), g(x, y), \frac{\partial f(x, y)}{\partial x}, \frac{\partial f(x, y)}{\partial y}, \frac{\partial g(x, y)}{\partial x}, \frac{\partial g(x, y)}{\partial y}, x, y\right)$$

In many cases it is not necessary to provide anything, except the Lagrangian, it will be auto-detected (and an error raised if this couldn't be done).

funcs : Function or an iterable of Functions

The functions that the Lagrangian depends on. The Euler equations are differential equations for each of these functions.

vars : Symbol or an iterable of Symbols

The Symbols that are the independent variables of the functions.

Returns eqns : list of Eq

The list of differential equations, one for each function.

References

[R1] (page 1277)

Examples

```
>>> from diofant import Symbol, Function
```

```
>>> x = Function('x')
>>> t = Symbol('t')
>>> L = (x(t).diff(t))**2/2 - x(t)**2/2
>>> euler_equations(L, x(t), t)
[Eq(-x(t) - Derivative(x(t), t, t), 0)]
>>> u = Function('u')
>>> x = Symbol('x')
>>> L = (u(t, x).diff(t))**2/2 - (u(t, x).diff(x))**2/2
>>> euler_equations(L, u(t, x), [t, x])
[Eq(-Derivative(u(t, x), t, t) + Derivative(u(t, x), x, x), 0)]
```

`diofant.calculus.singularities.singularities(f, x)`

Find singularities of real-valued function f with respect to x .

Notes

Removable singularities are not supported now.

References

[R2] (page 1277)

Examples

```
>>> from diofant import Symbol, exp, log
>>> from diofant.abc import x
```

```
>>> singularities(1/(1 + x), x) == {-1}
True
```

```
>>> singularities(exp(1/x) + log(x + 1), x) == {-1, 0}
True
```

```
>>> singularities(exp(1/log(x + 1)), x) == {0}
True
```

`diofant.calculus.optimization.minimize(f, *v)`
Minimizes *f* with respect to given variables *v*.

See also:

[maximize](#) (page 1172)

Examples

```
>>> from diofant.calculus import minimize
>>> from diofant.abc import x
>>> 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})
```

`diofant.calculus.optimization.maximize(f, *v)`
Maximizes *f* with respect to given variables *v*.

See also:

[minimize](#) (page 1172)

3.31.1 Finite difference weights

This module implements an algorithm for efficient generation of finite difference weights for ordinary differentials of functions for derivatives from 0 (interpolation) up to arbitrary order. The core algorithm is provided in the finite difference weight generating function (`finite_diff_weights`), and two convenience functions are provided for:

- **estimating a derivative (or interpolate) directly from a series of points** is also provided (`apply_finite_diff`).

- **making a finite difference approximation of a Derivative instance** (`as_finite_diff`).

```
diofant.calculus.finite_diff.apply_finite_diff(order, x_list, y_list,
                                              x0=Integer(0))
```

Calculates the finite difference approximation of the derivative of requested order at `x0` from points provided in `x_list` and `y_list`.

Parameters order: int

order of derivative to approximate. 0 corresponds to interpolation.

x_list: sequence

Sequence of (unique) values for the independent variable.

y_list: sequence

The function value at corresponding values for the independent variable in `x_list`.

x0: Number or Symbol

At what value of the independent variable the derivative should be evaluated. Defaults to `Integer(0)`.

Returns `diofant.core.add.Add` or `diofant.core.numbers.Number`

The finite difference expression approximating the requested derivative order at `x0`.

See also:

[diofant.calculus.finite_diff.finite_diff_weights](#) (page 1175)

Notes

Order = 0 corresponds to interpolation. Only supply so many points you think makes sense to around `x0` when extracting the derivative (the function need to be well behaved within that region). Also beware of Runge's phenomenon.

References

Fortran 90 implementation with Python interface for numerics: [fnitediff](#)

Examples

```
>>> from diofant.calculus import apply_finite_diff
>>> cube = lambda arg: (1.0*arg)**3
>>> xlist = range(-3,3+1)
>>> apply_finite_diff(2, xlist, list(map(cube, xlist)), 2) - 12
-3.55271367880050e-15
```

we see that the example above only contain rounding errors. `apply_finite_diff` can also be used on more abstract objects:

```

>>> from diofant import IndexedBase, Idx
>>> from diofant.calculus import apply_finite_diff
>>> x, y = map(IndexedBase, 'xy')
>>> i = Idx('i')
>>> x_list, y_list = zip(*[(x[i+j], y[i+j]) for j in range(-1,2)])
>>> apply_finite_diff(1, x_list, y_list, x[i])
(-1 + (x[i + 1] - x[i])/(-x[i - 1] + x[i]))*y[i]/(x[i + 1] - x[i]) + (-x[i - 1]
↳ + x[i])*y[i + 1]/((-x[i - 1] + x[i + 1])*(x[i + 1] - x[i])) - (x[i + 1] -
↳ x[i])*y[i - 1]/((-x[i - 1] + x[i + 1])*(-x[i - 1] + x[i]))

```

`diofant.calculus.finite_diff.as_finite_diff(derivative, points=1, x0=None, wrt=None)`

Returns an approximation of a derivative of a function in the form of a finite difference formula. The expression is a weighted sum of the function at a number of discrete values of (one of) the independent variable(s).

Parameters derivative: a Derivative instance (needs to have an variables

and expr attribute).

points: sequence or coefficient, optional

If sequence: discrete values (length \geq order+1) of the independent variable used for generating the finite difference weights. If it is a coefficient, it will be used as the step-size for generating an equidistant sequence of length order+1 centered around x0. default: 1 (step-size 1)

x0: number or Symbol, optional

the value of the independent variable (wrt) at which the derivative is to be approximated. default: same as wrt

wrt: Symbol, optional

“with respect to” the variable for which the (partial) derivative is to be approximated for. If not provided it is required that the Derivative is ordinary. default: None

See also:

[diofant.calculus.finite_diff.apply_finite_diff](#) (page 1173), [diofant.calculus.finite_diff.finite_diff_weights](#) (page 1175)

Examples

```

>>> from diofant import symbols, Function, exp, sqrt, Symbol, as_finite_diff
>>> x, h = symbols('x h')
>>> f = Function('f')
>>> as_finite_diff(f(x).diff(x))
-f(x - 1/2) + f(x + 1/2)

```

The default step size and number of points are 1 and order + 1 respectively. We can change the step size by passing a symbol as a parameter:

```

>>> as_finite_diff(f(x).diff(x), h)
-f(-h/2 + x)/h + f(h/2 + x)/h

```

We can also specify the discretized values to be used in a sequence:

```
>>> as_finite_diff(f(x).diff(x), [x, x+h, x+2*h])
-3*f(x)/(2*h) + 2*f(h + x)/h - f(2*h + x)/(2*h)
```

The algorithm is not restricted to use equidistant spacing, nor do we need to make the approximation around x_0 , but we can get an expression estimating the derivative at an offset:

```
>>> e, sq2 = exp(1), sqrt(2)
>>> x1 = [x-h, x+h, x+e*h]
>>> as_finite_diff(f(x).diff(x, 1), x1, x+h*sq2)
2*h*f(E*h + x)*((h + sqrt(2)*h)/(2*h) - (-sqrt(2)*h + h)/(2*h))/((-h +
↳ E*h)*(h + E*h)) + f(-h + x)*(-(-sqrt(2)*h + h)/(2*h) - (-sqrt(2)*h + E*h)/
↳ (2*h))/(h + E*h) + f(h + x)*(-(h + sqrt(2)*h)/(2*h) + (-sqrt(2)*h +
↳ E*h)/(2*h))/(-h + E*h)
```

Partial derivatives are also supported:

```
>>> y = Symbol('y')
>>> d2fdxdy=f(x,y).diff(x,y)
>>> as_finite_diff(d2fdxdy, wrt=x)
-f(x - 1/2, y) + f(x + 1/2, y)
```

`diofant.calculus.finite_diff.finite_diff_weights(order, x_list, x0=Integer(0))`

Calculates the finite difference weights for an arbitrarily spaced one-dimensional grid (*x_list*) for derivatives at '*x0*' of order 0, 1, ..., up to '*order*' using a recursive formula. Order of accuracy is at least $\text{len}(x_list) - \text{order}$, if *x_list* is defined accurately.

Parameters *order*: **int**

Up to what derivative order weights should be calculated. 0 corresponds to interpolation.

x_list: **sequence**

Sequence of (unique) values for the independent variable. It is useful (but not necessary) to order *x_list* from nearest to farthest from *x0*; see examples below.

x0: **Number or Symbol**

Root or value of the independent variable for which the finite difference weights should be generated. Defaults to Integer(0).

Returns *list*

A list of sublists, each corresponding to coefficients for increasing derivative order, and each containing lists of coefficients for increasing subsets of *x_list*.

See also:

[diofant.calculus.finite_diff.apply_finite_diff](#) (page 1173)

Notes

If weights for a finite difference approximation of 3rd order derivative is wanted, weights for 0th, 1st and 2nd order are calculated "for free", so are formulae using subsets of *x_list*. This is something one can take advantage of to save computational cost. Be aware that one should define *x_list* from nearest to farthest from *x0*. If not, subsets of

`x_list` will yield poorer approximations, which might not grand an order of accuracy of $\text{len}(x_list) - \text{order}$.

References

[R3] (page 1278)

Examples

```
>>> from diofant import Integer, Rational
>>> from diofant.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [-Rational(1, 2), Rational(1, 2), Rational(3, 2),
↳ Rational(5, 2)], 0)
>>> res
[[[1, 0, 0, 0],
  [1/2, 1/2, 0, 0],
  [3/8, 3/4, -1/8, 0],
  [5/16, 15/16, -5/16, 1/16]],
 [[0, 0, 0, 0],
  [-1, 1, 0, 0],
  [-1, 1, 0, 0],
  [-23/24, 7/8, 1/8, -1/24]]]
>>> res[0][-1] # FD weights for 0th derivative, using full x_list
[5/16, 15/16, -5/16, 1/16]
>>> res[1][-1] # FD weights for 1st derivative
[-23/24, 7/8, 1/8, -1/24]
>>> res[1][-2] # FD weights for 1st derivative, using x_list[:-1]
[-1, 1, 0, 0]
>>> res[1][-1][0] # FD weight for 1st deriv. for x_list[0]
-23/24
>>> res[1][-1][1] # FD weight for 1st deriv. for x_list[1], etc.
7/8
```

Each sublist contains the most accurate formula at the end. Note, that in the above example `res[1][1]` is the same as `res[1][2]`. Since `res[1][2]` has an order of accuracy of $\text{len}(x_list[:3]) - \text{order} = 3 - 1 = 2$, the same is true for `res[1][1]`!

```
>>> from diofant.calculus import finite_diff_weights
>>> res = finite_diff_weights(1, [Integer(0), Integer(1), -Integer(1), Integer(2),
↳ -Integer(2)], 0)[1]
>>> res
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [0, 1/2, -1/2, 0, 0],
 [-1/2, 1, -1/3, -1/6, 0],
 [0, 2/3, -2/3, -1/12, 1/12]]
>>> res[0] # no approximation possible, using x_list[0] only
[0, 0, 0, 0, 0]
>>> res[1] # classic forward step approximation
[-1, 1, 0, 0, 0]
>>> res[2] # classic centered approximation
[0, 1/2, -1/2, 0, 0]
>>> res[3:] # higher order approximations
[[-1/2, 1, -1/3, -1/6, 0], [0, 2/3, -2/3, -1/12, 1/12]]
```

Let us compare this to a differently defined `x_list`. Pay attention to `foo[i][k]` corresponding to the gridpoint defined by `x_list[k]`.

```
>>> from diofant.calculus import finite_diff_weights
>>> foo = finite_diff_weights(1, [-Integer(2), -Integer(1), Integer(0),
↳ Integer(1), Integer(2)], 0)[1]
>>> foo
[[0, 0, 0, 0, 0],
 [-1, 1, 0, 0, 0],
 [1/2, -2, 3/2, 0, 0],
 [1/6, -1, 1/2, 1/3, 0],
 [1/12, -2/3, 0, 2/3, -1/12]]
>>> foo[1] # not the same and of lower accuracy as res[1]!
[-1, 1, 0, 0, 0]
>>> foo[2] # classic double backward step approximation
[1/2, -2, 3/2, 0, 0]
>>> foo[4] # the same as res[4]
[1/12, -2/3, 0, 2/3, -1/12]
```

Note that, unless you plan on using approximations based on subsets of `x_list`, the order of gridpoints does not matter.

The capability to generate weights at arbitrary points can be used e.g. to minimize Runge's phenomenon by using Chebyshev nodes:

```
>>> from diofant import cos, symbols, pi, simplify
>>> from diofant.calculus import finite_diff_weights
>>> N, (h, x) = 4, symbols('h x')
>>> x_list = [x+h*cos(i*pi/(N)) for i in range(N,-1,-1)] # chebyshev nodes
>>> x_list
[-h + x, -sqrt(2)*h/2 + x, x, sqrt(2)*h/2 + x, h + x]
>>> mycoeffs = finite_diff_weights(1, x_list, 0)[1][4]
>>> [simplify(c) for c in mycoeffs]
[(h**3/2 + h**2*x - 3*h*x**2 - 4*x**3)/h**4,
 (-sqrt(2)*h**3 - 4*h**2*x + 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 6*x/h**2 - 8*x**3/h**4,
 (sqrt(2)*h**3 - 4*h**2*x - 3*sqrt(2)*h*x**2 + 8*x**3)/h**4,
 (-h**3/2 + h**2*x + 3*h*x**2 - 4*x**3)/h**4]
```

3.32 Differential Geometry

3.32.1 Introduction

3.32.2 Base Class Reference

class diofant.diffgeom.Manifold

Object representing a mathematical manifold.

The only role that this object plays is to keep a list of all patches defined on the manifold. It does not provide any means to study the topological characteristics of the manifold that it represents.

class diofant.diffgeom.Patch

Object representing a patch on a manifold.

On a manifold one can have many patches that do not always include the whole manifold. On these patches coordinate charts can be defined that permit the parametrization of any point on the patch in terms of a tuple of real numbers (the coordinates).

This object serves as a container/parent for all coordinate system charts that can be defined on the patch it represents.

Examples

Define a Manifold and a Patch on that Manifold:

```
>>> from diofant.diffgeom import Manifold, Patch
>>> m = Manifold('M', 3)
>>> p = Patch('P', m)
>>> p in m.patches
True
```

class diofant.diffgeom.CoordSystem

Contains all coordinate transformation logic.

Examples

Define a Manifold and a Patch, and then define two coord systems on that patch:

```
>>> from diofant import symbols, sin, cos, pi
>>> from diofant.diffgeom import Manifold, Patch, CoordSystem
>>> from diofant.simplify import simplify
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
>>> patch = Patch('P', m)
>>> rect = CoordSystem('rect', patch)
>>> polar = CoordSystem('polar', patch)
>>> rect in patch.coord_systems
True
```

Connect the coordinate systems. An inverse transformation is automatically found by solve when possible:

```
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
>>> polar.coord_tuple_transform_to(rect, [0, 2])
Matrix([
[0],
[0]])
>>> polar.coord_tuple_transform_to(rect, [2, pi/2])
Matrix([
[0],
[2]])
>>> rect.coord_tuple_transform_to(polar, [1, 1]).applyfunc(simplify)
Matrix([
[sqrt(2)],
[ pi/4]])
```

Calculate the jacobian of the polar to cartesian transformation:

```
>>> polar.jacobian(rect, [r, theta])
Matrix([
[cos(theta), -r*sin(theta)],
[sin(theta),  r*cos(theta)]])
```

Define a point using coordinates in one of the coordinate systems:

```
>>> p = polar.point([1, 3*pi/4])
>>> rect.point_to_coords(p)
Matrix([
[-sqrt(2)/2],
[ sqrt(2)/2]])
```

Define a basis scalar field (i.e. a coordinate function), that takes a point and returns its coordinates. It is an instance of `BaseScalarField`.

```
>>> rect.coord_function(0)(p)
-sqrt(2)/2
>>> rect.coord_function(1)(p)
sqrt(2)/2
```

Define a basis vector field (i.e. a unit vector field along the coordinate line). Vectors are also differential operators on scalar fields. It is an instance of `BaseVectorField`.

```
>>> v_x = rect.base_vector(0)
>>> x = rect.coord_function(0)
>>> v_x(x)
1
>>> v_x(v_x(x))
0
```

Define a basis oneform field:

```
>>> dx = rect.base_oneform(0)
>>> dx(v_x)
1
```

If you provide a list of names the fields will print nicely: - without provided names:

```
>>> x, v_x, dx
(rect_0, e_rect_0, drect_0)
```

- with provided names

```
>>> rect = CoordSystem('rect', patch, ['x', 'y'])
>>> rect.coord_function(0), rect.base_vector(0), rect.base_oneform(0)
(x, e_x, dx)
```

base_oneform(*coord_index*)

Return a basis 1-form field.

The basis one-form field for this coordinate system. It is also an operator on vector fields.

See the docstring of `CoordSystem` for examples.

base_oneforms()

Returns a list of all base oneforms.

For more details see the `base_oneform` method of this class.

base_vector(*coord_index*)

Return a basis vector field.

The basis vector field for this coordinate system. It is also an operator on scalar fields.

See the docstring of `CoordSystem` for examples.

base_vectors()

Returns a list of all base vectors.

For more details see the `base_vector` method of this class.

connect_to(*to_sys, from_coords, to_exprs, inverse=True, fill_in_gaps=False*)

Register the transformation used to switch to another coordinate system.

Parameters to_sys

another instance of `CoordSystem`

from_coords

list of symbols in terms of which `to_exprs` is given

to_exprs

list of the expressions of the new coordinate tuple

inverse

try to deduce and register the inverse transformation

fill_in_gaps

try to deduce other transformation that are made possible by composing the present transformation with other already registered transformation

coord_function(*coord_index*)

Return a `BaseScalarField` that takes a point and returns one of the coords.

Takes a point and returns its coordinate in this coordinate system.

See the docstring of `CoordSystem` for examples.

coord_functions()

Returns a list of all coordinate functions.

For more details see the `coord_function` method of this class.

coord_tuple_transform_to(*to_sys, coords*)

Transform `coords` to coord system `to_sys`.

See the docstring of `CoordSystem` for examples.

jacobian(*to_sys, coords*)

Return the jacobian matrix of a transformation.

point(*coords*)

Create a `Point` with coordinates given in this coord system.

See the docstring of `CoordSystem` for examples.

point_to_coords(*point*)

Calculate the coordinates of a point in this coord system.

See the docstring of `CoordSystem` for examples.

class diofant.diffgeom.**Point**(*coord_sys, coords*)

Point in a Manifold object.

To define a point you must supply coordinates and a coordinate system.

The usage of this object after its definition is independent of the coordinate system that was used in order to define it, however due to limitations in the simplification routines you can arrive at complicated expressions if you use inappropriate coordinate systems.

Examples

Define the boilerplate Manifold, Patch and coordinate systems:

```
>>> from diofant import symbols, sin, cos, pi
>>> from diofant.diffgeom import (
...     Manifold, Patch, CoordSystem, Point)
>>> r, theta = symbols('r, theta')
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r, theta], [r*cos(theta), r*sin(theta)])
```

Define a point using coordinates from one of the coordinate systems:

```
>>> p = Point(polar, [r, 3*pi/4])
>>> p.coords()
Matrix([
 [   r],
 [3*pi/4]])
>>> p.coords(rect)
Matrix([
 [-sqrt(2)*r/2],
 [ sqrt(2)*r/2]])
```

coords(*to_sys=None*)

Coordinates of the point in a given coordinate system.

If *to_sys* is None it returns the coordinates in the system in which the point was defined.

class diofant.diffgeom.**BaseScalarField**

Base Scalar Field over a Manifold for a given Coordinate System.

A scalar field takes a point as an argument and returns a scalar.

A base scalar field of a coordinate system takes a point and returns one of the coordinates of that point in the coordinate system in question.

To define a scalar field you need to choose the coordinate system and the index of the coordinate.

The use of the scalar field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use unappropriate coordinate systems.

You can build complicated scalar fields by just building up Diofant expressions containing **BaseScalarField** instances.

Examples

Define boilerplate Manifold, Patch and coordinate systems:

```
>>> from diofant import symbols, sin, cos, pi, Function
>>> from diofant.diffgeom import (
...     Manifold, Patch, CoordSystem, Point, BaseScalarField)
>>> r0, theta0 = symbols('r0, theta0')
>>> m = Manifold('M', 2)
>>> p = Patch('P', m)
>>> rect = CoordSystem('rect', p)
>>> polar = CoordSystem('polar', p)
>>> polar.connect_to(rect, [r0, theta0], [r0*cos(theta0), r0*sin(theta0)])
```

Point to be used as an argument for the field:

```
>>> point = polar.point([r0, 0])
```

Examples of fields:

```
>>> fx = BaseScalarField(rect, 0)
>>> fy = BaseScalarField(rect, 1)
>>> (fx**2+fy**2).rcall(point)
r0**2
```

```
>>> g = Function('g')
>>> ftheta = BaseScalarField(polar, 1)
>>> fg = g(ftheta-pi)
>>> fg.rcall(point)
g(-pi)
```

class diofant.diffgeom.BaseVectorField

Vector Field over a Manifold.

A vector field is an operator taking a scalar field and returning a directional derivative (which is also a scalar field).

A base vector field is the same type of operator, however the derivation is specifically done with respect to a chosen coordinate.

To define a base vector field you need to choose the coordinate system and the index of the coordinate.

The use of the vector field after its definition is independent of the coordinate system in which it was defined, however due to limitations in the simplification routines you may arrive at more complicated expression if you use inappropriate coordinate systems.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from diofant import symbols, pi, Function
>>> from diofant.diffgeom.rn import R2, R2_p, R2_r
>>> from diofant.diffgeom import BaseVectorField
>>> from diofant import pprint
>>> x0, y0, r0, theta0 = symbols('x0, y0, r0, theta0')
```

Points to be used as arguments for the field:

```
>>> point_p = R2_p.point([r0, theta0])
>>> point_r = R2_r.point([x0, y0])
```

Scalar field to operate on:

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
>>> s_field.rcall(point_r)
g(x0, y0)
>>> s_field.rcall(point_p)
g(r0*cos(theta0), r0*sin(theta0))
```

Vector field:

```
>>> v = BaseVectorField(R2_r, 1)
>>> pprint(v(s_field), use_unicode=False)
/ d          \|
|------(g(x, xi_2))||
\dx_i_2          /|xi_2=y
>>> pprint(v(s_field).rcall(point_r).doit(), use_unicode=False)
d
---(g(x0, y0))
dy0
>>> pprint(v(s_field).rcall(point_p).doit(), use_unicode=False)
/ d          \|
|------(g(r0*cos(theta0), xi_2))||
\dx_i_2          /|xi_2=r0*sin(theta0)
```

class diofant.diffgeom.Commutator(v_1, v_2)

Commutator of two vector fields.

The commutator of two vector fields v_1 and v_2 is defined as the vector field $[v_1, v_2]$ that evaluated on each scalar field f is equal to $v_1(v_2(f)) - v_2(v_1(f))$.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import Commutator
>>> from diofant import pprint
>>> from diofant.simplify import simplify
```

Vector fields:

```
>>> e_x, e_y, e_r = R2.e_x, R2.e_y, R2.e_r
>>> c_xy = Commutator(e_x, e_y)
>>> c_xr = Commutator(e_x, e_r)
>>> c_xy
0
```

Unfortunately, the current code is not able to compute everything:

```
>>> c_xr
Commutator(e_x, e_r)
```

```
>>> simplify(c_xr(R2.y**2).doit())
-2*cos(theta)*y**2/(x**2 + y**2)
```

class diofant.diffgeom.Differential(*form_field*)

Return the differential (exterior derivative) of a form field.

The differential of a form (i.e. the exterior derivative) has a complicated definition in the general case.

The differential df of the 0-form f is defined for any vector field v as $df(v) = v(f)$.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from diofant import Function
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import Differential
>>> from diofant import pprint
```

Scalar field (0-forms):

```
>>> g = Function('g')
>>> s_field = g(R2.x, R2.y)
```

Vector fields:

```
>>> e_x, e_y, = R2.e_x, R2.e_y
```

Differentials:

```
>>> dg = Differential(s_field)
>>> dg
d(g(x, y))
>>> pprint(dg(e_x), use_unicode=False)
/ d      \
|------(g(xi_1, y))|
\dx_i_1   /|xi_1=x
>>> pprint(dg(e_y), use_unicode=False)
/ d      \
|------(g(x, xi_2))|
\dx_i_2   /|xi_2=y
```

Applying the exterior derivative operator twice always results in:

```
>>> Differential(dg)
0
```

class diofant.diffgeom.TensorProduct(*args)

Tensor product of forms.

The tensor product permits the creation of multilinear functionals (i.e. higher order tensors) out of lower order forms (e.g. 1-forms). However, the higher tensors thus created lack the interesting features provided by the other type of product, the wedge product, namely they are not antisymmetric and hence are not form fields.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from diofant import Function
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import TensorProduct
>>> from diofant import pprint
```

```
>>> TensorProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> TensorProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
0
>>> TensorProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
```

You can nest tensor products.

```
>>> tp1 = TensorProduct(R2.dx, R2.dy)
>>> TensorProduct(tp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
1
```

You can make partial contraction for instance when ‘raising an index’. Putting None in the second argument of rcall means that the respective position in the tensor product is left as it is.

```
>>> TP = TensorProduct
>>> metric = TP(R2.dx, R2.dx) + 3*TP(R2.dy, R2.dy)
>>> metric.rcall(R2.e_y, None)
3*dy
```

Or automatically pad the args with None without specifying them.

```
>>> metric.rcall(R2.e_y)
3*dy
```

class diofant.diffgeom.WedgeProduct(*args)

Wedge product of forms.

In the context of integration only completely antisymmetric forms make sense. The wedge product permits the creation of such forms.

Examples

Use the predefined R2 manifold, setup some boilerplate.

```
>>> from diofant import Function
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import WedgeProduct
>>> from diofant import pprint
```

```
>>> WedgeProduct(R2.dx, R2.dy)(R2.e_x, R2.e_y)
1
>>> WedgeProduct(R2.dx, R2.dy)(R2.e_y, R2.e_x)
-1
```

```
>>> WedgeProduct(R2.dx, R2.x*R2.dy)(R2.x*R2.e_x, R2.e_y)
x**2
```

You can nest wedge products.

```
>>> wp1 = WedgeProduct(R2.dx, R2.dy)
>>> WedgeProduct(wp1, R2.dx)(R2.e_x, R2.e_y, R2.e_x)
0
```

class diofant.diffgeom.LieDerivative(*v_field, expr*)

Lie derivative with respect to a vector field.

The transport operator that defines the Lie derivative is the pushforward of the field to be derived along the integral curve of the field with respect to which one derives.

Examples

```
>>> from diofant.diffgeom import (LieDerivative, TensorProduct)
>>> from diofant.diffgeom.rn import R2
>>> LieDerivative(R2.e_x, R2.y)
0
>>> LieDerivative(R2.e_x, R2.x)
1
>>> LieDerivative(R2.e_x, R2.e_x)
0
```

The Lie derivative of a tensor field by another tensor field is equal to their commutator:

```
>>> LieDerivative(R2.e_x, R2.e_r)
Commutator(e_x, e_r)
>>> LieDerivative(R2.e_x + R2.e_y, R2.x)
1
>>> tp = TensorProduct(R2.dx, R2.dy)
>>> LieDerivative(R2.e_x, tp)
LieDerivative(e_x, TensorProduct(dx, dy))
>>> LieDerivative(R2.e_x, tp).doit()
LieDerivative(e_x, TensorProduct(dx, dy))
```

class diofant.diffgeom.BaseCovarDerivative0p(*coord_sys, index, christoffel*)

Covariant derivative operator with respect to a base vector.

Examples

```
>>> from diofant.diffgeom.rn import R2, R2_r
>>> from diofant.diffgeom import BaseCovarDerivative0p
>>> from diofant.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = BaseCovarDerivative0p(R2_r, 0, ch)
>>> cvd(R2.x)
1
>>> cvd(R2.x*R2.e_x)
e_x
```

```
class diofant.diffgeom.CovarDerivative0p(wrt, christoffel)
    Covariant derivative operator.
```

Examples

```
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import CovarDerivative0p
>>> from diofant.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> ch = metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
>>> ch
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> cvd = CovarDerivative0p(R2.x*R2.e_x, ch)
>>> cvd(R2.x)
x
>>> cvd(R2.x*R2.e_x)
x*e_x
```

```
diofant.diffgeom.intcurve_series(vector_field, param, start_point, n=6, coord_sys=None, coeffs=False)
```

Return the series expansion for an integral curve of the field.

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as V . This holds for any value t for the parameter and any scalar field f .

This equation can also be decomposed of a basis of coordinate functions

$$V(f_i)(\gamma(t)) = \frac{d}{dt}f_i(\gamma(t)) \quad \forall i$$

This function returns a series expansion of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Parameters `vector_field`

the vector field for which an integral curve will be given

`param`

the argument of the function γ from R to the curve

`start_point`

the point which corresponds to $\gamma(0)$

`n`

the order to which to expand

`coord_sys`

the coordinate system in which to expand `coeffs` (default `False`) - if `True` return a list of elements of the expansion

See also:

[`intcurve_diffegu`](#) (page 1188)

Examples

Use the predefined R2 manifold:

```
>>> from diofant.abc import t, x, y
>>> from diofant.diffgeom.rn import R2, R2_p, R2_r
>>> from diofant.diffgeom import intcurve_series
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([x, y])
>>> vector_field = R2_r.e_x
```

Calculate the series:

```
>>> intcurve_series(vector_field, t, start_point, n=3)
Matrix([
[t + x],
[ y]])
```

Or get the elements of the expansion in a list:

```
>>> series = intcurve_series(vector_field, t, start_point, n=3, coeffs=True)
>>> series[0]
Matrix([
[x],
[y]])
>>> series[1]
Matrix([
[t],
[0]])
>>> series[2]
Matrix([
[0],
[0]])
```

The series in the polar coordinate system:

```
>>> series = intcurve_series(vector_field, t, start_point,
...                          n=3, coord_sys=R2_p, coeffs=True)
>>> series[0]
Matrix([
[sqrt(x**2 + y**2)],
[ atan2(y, x)]])
>>> series[1]
Matrix([
[t*x/sqrt(x**2 + y**2)],
[ -t*y/(x**2 + y**2)]])
>>> series[2]
Matrix([
[t**2*(-x**2/(x**2 + y**2)**(3/2) + 1/sqrt(x**2 + y**2))/2],
[ t**2*x*y/(x**2 + y**2)**2]])
```

`diofant.diffgeom.intcurve_diffequ`(*vector_field*, *param*, *start_point*, *coord_sys=None*)

Return the differential equation for an integral curve of the field.

Integral curve is a function γ taking a parameter in R to a point in the manifold. It verifies the equation:

$$V(f)(\gamma(t)) = \frac{d}{dt}f(\gamma(t))$$

where the given `vector_field` is denoted as V . This holds for any value t for the parameter and any scalar field f .

This function returns the differential equation of $\gamma(t)$ in terms of the coordinate system `coord_sys`. The equations and expansions are necessarily done in coordinate-system-dependent way as there is no other way to represent movement between points on the manifold (i.e. there is no such thing as a difference of points for a general manifold).

Parameters `vector_field`

the vector field for which an integral curve will be given

`param`

the argument of the function γ from \mathbb{R} to the curve

`start_point`

the point which corresponds to $\gamma(0)$

`coord_sys`

the coordinate system in which to give the equations

Returns a tuple of (equations, initial conditions)

See also:

[intcurve_series](#) (page 1187)

Examples

Use the predefined `R2` manifold:

```
>>> from diofant.abc import t
>>> from diofant.diffgeom.rn import R2, R2_p, R2_r
>>> from diofant.diffgeom import intcurve_diffequ
```

Specify a starting point and a vector field:

```
>>> start_point = R2_r.point([0, 1])
>>> vector_field = -R2.y*R2.e_x + R2.x*R2.e_y
```

Get the equation:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point)
>>> equations
[f_1(t) + Derivative(f_0(t), t), -f_0(t) + Derivative(f_1(t), t)]
>>> init_cond
[f_0(0), f_1(0) - 1]
```

The series in the polar coordinate system:

```
>>> equations, init_cond = intcurve_diffequ(vector_field, t, start_point, R2_p)
>>> equations
[Derivative(f_0(t), t), Derivative(f_1(t), t) - 1]
>>> init_cond
[f_0(0) - 1, f_1(0) - pi/2]
```

`diofant.diffgeom.vectors_in_basis(expr, to_sys)`

Transform all base vectors in base vectors of a specified coord basis.

While the new base vectors are in the new coordinate system basis, any coefficients are kept in the old system.

Examples

```
>>> from diofant.diffgeom import vectors_in_basis
>>> from diofant.diffgeom.rn import R2_r, R2_p
>>> vectors_in_basis(R2_r.e_x, R2_p)
-y*e_theta/(x**2 + y**2) + x*e_r/sqrt(x**2 + y**2)
>>> vectors_in_basis(R2_p.e_r, R2_r)
sin(theta)*e_y + cos(theta)*e_x
```

`diofant.diffgeom.twoform_to_matrix(expr)`

Return the matrix representing the twoform.

For the twoform w return the matrix M such that $M[i, j] = w(e_i, e_j)$, where e_i is the i -th base vector field for the coordinate system in which the expression of w is given.

Examples

```
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import twoform_to_matrix, TensorProduct
>>> TP = TensorProduct
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[1, 0],
[0, 1]])
>>> twoform_to_matrix(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
Matrix([
[x, 0],
[0, 1]])
>>> twoform_to_matrix(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy) - TP(R2.dx, R2.dy)/2)
Matrix([
[ 1, 0],
[-1/2, 1]])
```

`diofant.diffgeom.metric_to_Christoffel_1st(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of first kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import metric_to_Christoffel_1st, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_1st(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_1st(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/2, 0], [0, 0]], [[0, 0], [0, 0]]]
```

`diofant.diffgeom.metric_to_Christoffel_2nd(expr)`

Return the nested list of Christoffel symbols for the given metric.

This returns the Christoffel symbol of second kind that represents the Levi-Civita connection for the given metric.

Examples

```
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import metric_to_Christoffel_2nd, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Christoffel_2nd(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]
>>> metric_to_Christoffel_2nd(R2.x*TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[1/(2*x), 0], [0, 0]], [[0, 0], [0, 0]]]
```

`diofant.diffgeom.metric_to_Riemann_components(expr)`

Return the components of the Riemann tensor expressed in a given basis.

Given a metric it calculates the components of the Riemann tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from diofant import pprint, exp
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import metric_to_Riemann_components, TensorProduct
>>> TP = TensorProduct
>>> metric_to_Riemann_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[[[0, 0], [0, 0]], [[0, 0], [0, 0]]], [[[[0, 0], [0, 0]], [[0, 0], [0, 0]]]]]
```

```
>>> non_trivial_metric = exp(2*R2.r)*TP(R2.dr, R2.dr) + R2.r**2*TP(R2.
↳ dtheta, R2.dtheta)
>>> non_trivial_metric
E**(2*r)*TensorProduct(dr, dr) + r**2*TensorProduct(dtheta, dtheta)
>>> riemann = metric_to_Riemann_components(non_trivial_metric)
>>> riemann[0, :, :, :]
[[[0, 0], [0, 0]], [[0, E**(-2*r)*r], [-E**(-2*r)*r, 0]]]
>>> riemann[1, :, :, :]
[[[0, -1/r], [1/r, 0]], [[0, 0], [0, 0]]]
```

`diofant.diffgeom.metric_to_Ricci_components(expr)`

Return the components of the Ricci tensor expressed in a given basis.

Given a metric it calculates the components of the Ricci tensor in the canonical basis of the coordinate system in which the metric expression is given.

Examples

```
>>> from diofant import pprint, exp
>>> from diofant.diffgeom.rn import R2
>>> from diofant.diffgeom import metric_to_Ricci_components, TensorProduct
>>> TP = TensorProduct
```

```
>>> metric_to_Ricci_components(TP(R2.dx, R2.dx) + TP(R2.dy, R2.dy))
[[0, 0], [0, 0]]
```

```
>>> non_trivial_metric = (exp(2*R2.r)*TP(R2.dr, R2.dr) +
...                       R2.r**2*TP(R2.dtheta, R2.dtheta))
>>> non_trivial_metric
E**(2*r)*TensorProduct(dr, dr) + r**2*TensorProduct(dtheta, dtheta)
>>> metric_to_Ricci_components(non_trivial_metric)
[[1/r, 0], [0, E**(-2*r)*r]]
```

3.33 Vector Module

The vector module provides tools for basic vector math and differential calculus with respect to 3D Cartesian coordinate systems. This documentation provides an overview of all the features offered, and relevant API.

3.33.1 Guide to Vector

Introduction

This page gives a brief conceptual overview of the functionality present in *diofant.vector* (page 1193).

Vectors and Scalars

In vector math, we deal with two kinds of quantities - scalars and vectors.

A **scalar** is an entity which only has a magnitude - no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A **vector**, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector \mathbf{V} , the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$ m, where, as per standard convention, $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ represent unit vectors along the \mathbf{X} , \mathbf{Y} and \mathbf{Z} axes respectively. Therefore, it can be concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\|$ m = $5\sqrt{2}$ m. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Coordinate Systems

A **coordinate system** is an abstract mathematical entity used to define the notion of directions and locations in n-dimensional spaces. This module deals with 3-dimensional spaces, with the conventional X , Y and Z axes defined with respect to each coordinate system.

Each coordinate system also has a special reference point called the ‘origin’ defined for it. This point is used either while referring to locations in 3D space, or while calculating the coordinates of pre-defined points with respect to the system.

It is a pretty well-known concept that there is no absolute notion of location or orientation in space. Any given coordinate system defines a unique ‘perspective’ of quantifying positions and directions. Therefore, even if we assume that all systems deal with the same units of measurement, the expression of vectorial and scalar quantities differs according to the coordinate system a certain observer deals with.

Consider two points P and Q in space. Assuming units to be common throughout, the distance between these points remains the same regardless of the coordinate system in which the measurements are being made. However, the 3-D coordinates of each of the two points, as well as the position vector of any of the points with respect to the other, do not. In fact, these two quantities don’t make sense at all, unless they are being measured keeping in mind a certain location and orientation of the measurer (essentially the coordinate system).

Therefore, it is quite clear that the orientation and location (of the origin) of a coordinate system define the way different quantities will be expressed with respect to it. Neither of the two properties can be measured on an absolute scale, but rather with respect to another coordinate system. The orientation of one system with respect to another is measured using the rotation matrix, while the relative position can be quantified via the position vector of one system’s origin with respect to the other.

Fields

A **field** is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). Since we only deal with 3D spaces in this module, a field is defined as a function of the x , y and z coordinates corresponding to a location in the coordinate system. Here, x , y and z act as scalar variables defining the position of a general point.

For example, temperature in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ - a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $\vec{g}(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{i} + 2x^2\hat{j}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Basic Implementation details

Coordinate Systems and Vectors

As of now, *diofant.vector* (page 1193) only deals with the Cartesian (also called rectangular) coordinate systems. A 3D Cartesian coordinate system can be initialized in *diofant.vector* (page 1193) as

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
```

The string parameter to the constructor denotes the name assigned to the system, and will primarily be used for printing purposes.

Once a coordinate system (in essence, a `CoordSysCartesian` instance) has been defined, we can access the orthonormal unit vectors (i.e. the \hat{i} , \hat{j} and \hat{k} vectors) and coordinate variables/base scalars (i.e. the x , y and z variables) corresponding to it. We will talk about coordinate variables in the later sections.

The basis vectors for the X , Y and Z axes can be accessed using the `i`, `j` and `k` properties respectively.

```
>>> N.i
N.i
>>> type(N.i)
<class 'diofant.vector.vector.BaseVector'>
```

As seen above, the basis vectors are all instances of a class called `BaseVector`.

When a `BaseVector` is multiplied by a scalar (essentially any Diofant Expr), we get a `VectorMul` - the product of a base vector and a scalar.

```
>>> 3*N.i
3*N.i
>>> type(3*N.i)
<class 'diofant.vector.vector.VectorMul'>
```

Addition of `VectorMul` and `BaseVectors` gives rise to formation of `VectorAdd` - except for special cases, ofcourse.

```
>>> v = 2*N.i + N.j
>>> type(v)
<class 'diofant.vector.vector.VectorAdd'>
>>> v - N.j
2*N.i
>>> type(v - N.j)
<class 'diofant.vector.vector.VectorMul'>
```

What about a zero vector? It can be accessed using the `zero` attribute assigned to class `Vector`. Since the notion of a zero vector remains the same regardless of the coordinate system in consideration, we use `Vector.zero` wherever such a quantity is required.

```
>>> from diofant.vector import Vector
>>> Vector.zero
0
>>> type(Vector.zero)
<class 'diofant.vector.vector.VectorZero'>
>>> N.i + Vector.zero
N.i
>>> Vector.zero == 2*Vector.zero
True
```

All the classes shown above - `BaseVector`, `VectorMul`, `VectorAdd` and `VectorZero` are subclasses of `Vector`.

You should never have to instantiate objects of any of the subclasses of `Vector`. Using the `BaseVector` instances assigned to a `CoordSysCartesian` instance and (if needed) `Vector.zero`

as building blocks, any sort of vectorial expression can be constructed with the basic mathematical operators $+$, $-$, $*$, and $/$.

```
>>> v = N.i - 2*N.j
>>> v/3
1/3*N.i + (-2/3)*N.j
>>> v + N.k
N.i + (-2)*N.j + N.k
>>> Vector.zero/2
0
>>> (v/3)*4
4/3*N.i + (-8/3)*N.j
```

In addition to the elementary mathematical operations, the vector operations of dot and cross can also be performed on Vector.

```
>>> v1 = 2*N.i + 3*N.j - N.k
>>> v2 = N.i - 4*N.j + N.k
>>> v1.dot(v2)
-11
>>> v1.cross(v2)
(-1)*N.i + (-3)*N.j + (-11)*N.k
>>> v2.cross(v1)
N.i + 3*N.j + 11*N.k
```

The $&$ and $^$ operators have been overloaded for the dot and cross methods respectively.

```
>>> v1 & v2
-11
>>> v1 ^ v2
(-1)*N.i + (-3)*N.j + (-11)*N.k
```

However, this is not the recommended way of performing these operations. Using the original methods makes the code clearer and easier to follow.

In addition to these operations, it is also possible to compute the outer products of Vector instances in [diofant.vector](#) (page 1193). More on that in a little bit.

Diofant operations on Vectors

The Diofant operations of `simplify`, `trigsimp`, `diff`, and `factor` work on Vector objects, with the standard Diofant API.

In essence, the methods work on the measure numbers(The coefficients of the basis vectors) present in the provided vectorial expression.

```
>>> from diofant.abc import a, b, c
>>> from diofant import sin, cos, trigsimp, diff
>>> v = (a*b + a*c + b**2 + b*c)*N.i + N.j
>>> v.factor()
((a + b)*(b + c))*N.i + N.j
>>> v = (sin(a)**2 + cos(a)**2)*N.i - (2*cos(b)**2 - 1)*N.k
>>> trigsimp(v)
N.i + (-cos(2*b))*N.k
>>> v.simplify()
N.i + (-cos(2*b))*N.k
>>> diff(v, b)
(4*sin(b)*cos(b))*N.k
```

```
>>> from diofant import Derivative
>>> Derivative(v, b).doit()
(4*sin(b)*cos(b))*N.k
```

Integral also works with Vector instances, similar to Derivative.

```
>>> from diofant import Integral
>>> v1 = a*N.i + sin(a)*N.j - N.k
>>> Integral(v1, a)
(Integral(a, a))*N.i + (Integral(sin(a), a))*N.j + (Integral(-1, a))*N.k
>>> Integral(v1, a).doit()
a**2/2*N.i + (-cos(a))*N.j + (-a)*N.k
```

Points

As mentioned before, every coordinate system corresponds to a unique origin point. Points, in general, have been implemented in *diofant.vector* (page 1193) in the form of the `Point` class.

To access the origin of system, use the `origin` property of the `CoordSysCartesian` class.

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> N.origin
N.origin
>>> type(N.origin)
<class 'diofant.vector.point.Point'>
```

You can instantiate new points in space using the `locate_new` method of `Point`. The arguments include the name(string) of the new `Point`, and its position vector with respect to the 'parent' `Point`.

```
>>> from diofant.abc import a, b, c
>>> P = N.origin.locate_new('P', a*N.i + b*N.j + c*N.k)
>>> Q = P.locate_new('Q', -b*N.j)
```

Like `Vector`, a user never has to expressly instantiate an object of `Point`. This is because any location in space (albeit relative) can be pointed at by using the origin of a `CoordSysCartesian` as the reference, and then using `locate_new` on it and subsequent `Point` instances.

The position vector of a `Point` with respect to another `Point` can be computed using the `position_wrt` method.

```
>>> P.position_wrt(Q)
b*N.j
>>> Q.position_wrt(N.origin)
a*N.i + c*N.k
```

Additionally, it is possible to obtain the *X*, *Y* and *Z* coordinates of a `Point` with respect to a `CoordSysCartesian` in the form of a tuple. This is done using the `express_coordinates` method.

```
>>> Q.express_coordinates(N)
(a, 0, c)
```


Dyadics

A dyadic, or dyadic tensor, is a second-order tensor formed by the juxtaposition of pairs of vectors. Therefore, the outer products of vectors give rise to the formation of dyadics. Dyadic tensors have been implemented in *diofant.vector* (page 1193) in the `Dyadic` class.

Once again, you never have to instantiate objects of `Dyadic`. The outer products of vectors can be computed using the `outer` method of `Vector`. The `|` operator has been overloaded for `outer`.

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> N.i.outer(N.j)
(N.i|N.j)
>>> N.i|N.j
(N.i|N.j)
```

Similar to `Vector`, `Dyadic` also has subsequent subclasses like `BaseDyadic`, `DyadicMul`, `DyadicAdd`. As with `Vector`, a zero dyadic can be accessed from `Dyadic.zero`.

All basic mathematical operations work with `Dyadic` too.

```
>>> dyad = N.i.outer(N.k)
>>> dyad*3
3*(N.i|N.k)
>>> dyad - dyad
0
>>> dyad + 2*(N.j|N.i)
(N.i|N.k) + 2*(N.j|N.i)
```

`dot` and `cross` also work among `Dyadic` instances as well as between a `Dyadic` and `Vector` (and also vice versa) - as per the respective mathematical definitions. As with `Vector`, `&` and `^` have been overloaded for `dot` and `cross`.

```
>>> d = N.i.outer(N.j)
>>> d.dot(N.j|N.j)
(N.i|N.j)
>>> d.dot(N.i)
0
>>> d.dot(N.j)
N.i
>>> N.i.dot(d)
N.j
>>> N.k ^ d
(N.j|N.j)
```

More about Coordinate Systems

We will now look at how we can initialize new coordinate systems in *diofant.vector* (page 1193), positioned and oriented in user-defined ways with respect to already-existing systems.

Locating new systems

We already know that the `origin` property of a `CoordSysCartesian` corresponds to the `Point` instance denoting its origin reference point.

Consider a coordinate system N . Suppose we want to define a new system M , whose origin is located at $3\hat{i} + 4\hat{j} + 5\hat{k}$ from N 's origin. In other words, the coordinates of M 's origin from N 's perspective happen to be $(3, 4, 5)$. Moreover, this would also mean that the coordinates of N 's origin with respect to M would be $(-3, -4, -5)$.

This can be achieved programatically as follows -

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> M = N.locate_new('M', 3*N.i + 4*N.j + 5*N.k)
>>> M.position_wrt(N)
3*N.i + 4*N.j + 5*N.k
>>> N.origin.express_coordinates(M)
(-3, -4, -5)
```

It is worth noting that M 's orientation is the same as that of N . This means that the rotation matrix of N with respect to M , and also vice versa, is equal to the identity matrix of dimensions 3×3 . The `locate_new` method initializes a `CoordSysCartesian` that is only translated in space, not re-oriented, relative to the 'parent' system.

Orienting new systems

Similar to 'locating' new systems, `diofant.vector` (page 1193) also allows for initialization of new `CoordSysCartesian` instances that are oriented in user-defined ways with respect to existing systems.

Suppose you have a coordinate system A .

```
>>> from diofant.vector import CoordSysCartesian
>>> A = CoordSysCartesian('A')
```

You want to initialize a new coordinate system B , that is rotated with respect to A 's Z-axis by an angle θ .

```
>>> from diofant import Symbol
>>> theta = Symbol('theta')
```

There are two ways to achieve this.

Using a method of `CoordSysCartesian` directly

This is the easiest, cleanest, and hence the recommended way of doing it.

```
>>> B = A.orient_new_axis('B', theta, A.k)
```

This initializes B with the required orientation information with respect to A .

`CoordSysCartesian` provides the following direct orientation methods in its API-

1. `orient_new_axis`
2. `orient_new_body`

3. `orient_new_space`
4. `orient_new_quaternion`

Please look at the `CoordSysCartesian` class API given in the docs of this module, to know their functionality and required arguments in detail.

Using Orienter(s) and the `orient_new` method

You would first have to initialize an `AxisOrienter` instance for storing the rotation information.

```
>>> from diofant.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(theta, A.k)
```

And then apply it using the `orient_new` method, to obtain B .

```
>>> B = A.orient_new('B', axis_orienter)
```

`orient_new` also lets you orient new systems using multiple `Orienter` instances, provided in an iterable. The rotations/orientations are applied to the new system in the order the `Orienter` instances appear in the iterable.

```
>>> from diofant.vector import BodyOrienter
>>> from diofant.abc import a, b, c
>>> body_orienter = BodyOrienter(a, b, c, 'XYZ')
>>> C = A.orient_new('C', (axis_orienter, body_orienter))
```

The `diofant.vector` (page 1193) API provides the following four `Orienter` classes for orientation purposes-

1. `AxisOrienter`
2. `BodyOrienter`
3. `SpaceOrienter`
4. `QuaternionOrienter`

Please refer to the API of the respective classes in the docs of this module to know more.

In each of the above examples, the origin of the new coordinate system coincides with the origin of the 'parent' system.

```
>>> B.position_wrt(A)
0
```

To compute the rotation matrix of any coordinate system with respect to another one, use the `rotation_matrix` method.

```
>>> B = A.orient_new_axis('B', a, A.k)
>>> B.rotation_matrix(A)
Matrix([
 [ cos(a), sin(a), 0],
 [-sin(a), cos(a), 0],
 [      0,      0, 1]])
>>> B.rotation_matrix(B)
Matrix([
 [1, 0, 0],
```

```
[0, 1, 0],  
[0, 0, 1]])
```

Orienting AND Locating new systems

What if you want to initialize a new system that is not only oriented in a pre-defined way, but also translated with respect to the parent?

Each of the `orient_new_<method of orientation>` methods, as well as the `orient_new` method, support a `location` keyword argument.

If a `Vector` is supplied as the value for this `kwarg`, the new system's origin is automatically defined to be located at that position vector with respect to the parent coordinate system.

Thus, the orientation methods also act as methods to support orientation+ location of the new systems.

```
>>> C = A.orient_new_axis('C', a, A.k, location=2*A.j)  
>>> C.position_wrt(A)  
2*A.j  
>>> from diofant.vector import express  
>>> express(A.position_wrt(C), C)  
(-2*sin(a))*C.i + (-2*cos(a))*C.j
```

More on the `express` function in a bit.

Expression of quantities in different coordinate systems

Vectors and Dyadics

As mentioned earlier, the same vector attains different expressions in different coordinate systems. In general, the same is true for scalar expressions and dyadic tensors.

`diofant.vector` (page 1193) supports the expression of vector/scalar quantities in different coordinate systems using the `express` function.

For purposes of this section, assume the following initializations-

```
>>> from diofant.vector import CoordSysCartesian, express  
>>> from diofant.abc import a, b, c  
>>> N = CoordSysCartesian('N')  
>>> M = N.orient_new_axis('M', a, N.k)
```

Vector instances can be expressed in user defined systems using `express`.

```
>>> v1 = N.i + N.j + N.k  
>>> express(v1, M)  
(sin(a) + cos(a))*M.i + (-sin(a) + cos(a))*M.j + M.k  
>>> v2 = N.i + M.j  
>>> express(v2, N)  
(-sin(a) + 1)*N.i + (cos(a))*N.j
```

Apart from `Vector` instances, `express` also supports reexpression of scalars (general Diofant Expr) and Dyadic objects.

`express` also accepts a second coordinate system for re-expressing Dyadic instances.

```
>>> d = 2*(M.i | N.j) + 3*(M.j | N.k)
>>> express(d, M)
(2*sin(a))*(M.i|M.i) + (2*cos(a))*(M.i|M.j) + 3*(M.j|M.k)
>>> express(d, M, N)
2*(M.i|N.j) + 3*(M.j|N.k)
```

Coordinate Variables

The location of a coordinate system's origin does not affect the re-expression of `BaseVector` instances. However, it does affect the way `BaseScalar` instances are expressed in different systems.

`BaseScalar` instances, are coordinate 'symbols' meant to denote the variables used in the definition of vector/scalar fields in *diofant.vector* (page 1193).

For example, consider the scalar field $\mathbf{T}_N(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x} + \mathbf{y} + \mathbf{z}$ defined in system N . Thus, at a point with coordinates (a, b, c) , the value of the field would be $a + b + c$. Now consider system R , whose origin is located at $(1, 2, 3)$ with respect to N (no change of orientation). A point with coordinates (a, b, c) in R has coordinates $(a + 1, b + 2, c + 3)$ in N . Therefore, the expression for \mathbf{T}_N in R becomes $\mathbf{T}_R(x, y, z) = x + y + z + 6$.

Coordinate variables, if present in a vector/scalar/dyadic expression, can also be re-expressed in a given coordinate system, by setting the `variables` keyword argument of `express` to `True`.

The above mentioned example, done programatically, would look like this -

```
>>> R = N.locate_new('R', N.i + 2*N.j + 3*N.k)
>>> T_N = N.x + N.y + N.z
>>> express(T_N, R, variables=True)
R.x + R.y + R.z + 6
```

Other expression-dependent methods

The `to_matrix` method of `Vector` and `express_coordinates` method of `Point` also return different results depending on the coordinate system being provided.

```
>>> P = R.origin.locate_new('P', a*R.i + b*R.j + c*R.k)
>>> P.express_coordinates(N)
(a + 1, b + 2, c + 3)
>>> P.express_coordinates(R)
(a, b, c)
>>> v = N.i + N.j + N.k
>>> v.to_matrix(M)
Matrix([
[ sin(a) + cos(a)],
[-sin(a) + cos(a)],
[                1]])
>>> v.to_matrix(N)
Matrix([
[1],
[1],
[1]])
```

Scalar and Vector Field Functionality

Implementation in diofant.vector

Scalar and vector fields

In *diofant.vector* (page 1193), every `CoordSysCartesian` instance is assigned basis vectors corresponding to the X , Y and Z axes. These can be accessed using the properties named `i`, `j` and `k` respectively. Hence, to define a vector \mathbf{v} of the form $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$ with respect to a given frame \mathbf{R} , you would do

```
>>> from diofant.vector import CoordSysCartesian
>>> R = CoordSysCartesian('R')
>>> v = 3*R.i + 4*R.j + 5*R.k
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in the earlier section of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented in a special class called `BaseScalar`, and are assigned to every coordinate system, one for each axis from X , Y and Z . These coordinate variables are used to form the expressions of vector or scalar fields in 3D space. For a system \mathbf{R} , the X , Y and Z `BaseScalar`s instances can be accessed using the `R.x`, `R.y` and `R.z` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from diofant.vector import CoordSysCartesian
>>> R = CoordSysCartesian('R')
>>> electric_potential = 2*R.x**2*R.y
>>> electric_potential
2*R.x**2*R.y
```

It is to be noted that `BaseScalar` instances can be used just like any other `Diofant Symbol`, except that they store the information about the coordinate system and axis they correspond to.

Scalar fields can be treated just as any other `Diofant expression`, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to x (i.e. `R.x`), you would use the `diff` method.

```
>>> from diofant.vector import CoordSysCartesian
>>> R = CoordSysCartesian('R')
>>> electric_potential = 2*R.x**2*R.y
>>> from diofant import diff
>>> diff(electric_potential, R.x)
4*R.x*R.y
```

It is worth noting that having a `BaseScalar` in the expression implies that a 'field' changes with position, in 3D space. Technically speaking, a simple `Expr` with no `BaseScalar`s is still a field, though constant.

Like scalar fields, vector fields that vary with position can also be constructed using `BaseScalar`s in the measure-number expressions.

```
>>> from diofant.vector import CoordSysCartesian
>>> R = CoordSysCartesian('R')
>>> v = R.x**2*R.i + 2*R.x*R.z*R.k
```

The Del operator

The Del, or ‘Nabla’ operator - written as ∇ is commonly known as the vector differential operator. Depending on its usage in a mathematical expression, it may denote the gradient of a scalar field, or the divergence of a vector field, or the curl of a vector field.

Essentially, ∇ is not technically an ‘operator’, but a convenient mathematical notation to denote any one of the aforementioned field operations.

In *diofant.vector* (page 1193), ∇ has been implemented as the `delop` property of the `CoordSysCartesian` class. Hence, assuming `C` is a coordinate system, the ∇ operator corresponding to the vector differentials wrt `C`’s coordinate variables and basis vectors would be accessible as `C.delop`.

Given below is an example of usage of the `delop` object.

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> gradient_field = C.delop(C.x*C.y*C.z)
>>> gradient_field
(Derivative(C.x*C.y*C.z, C.x))*C.i + (Derivative(C.x*C.y*C.z, C.y))*C.j +
↳(Derivative(C.x*C.y*C.z, C.z))*C.k
```

The above expression can be evaluated using the `Diofant.doit()` routine.

```
>>> gradient_field.doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Usage of the ∇ notation in *diofant.vector* (page 1193) has been described in greater detail in the subsequent subsections.

Field operators and related functions

Here we describe some basic field-related functionality implemented in *diofant.vector* (page 1193).

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector \mathbf{F} , denoted by $\nabla \times \mathbf{F}$ is given by:

$$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right) \hat{\mathbf{i}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right) \hat{\mathbf{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right) \hat{\mathbf{k}}$$

where F_x denotes the X component of vector \mathbf{F} .

Computing the curl of a vector field in *diofant.vector* (page 1193) can be accomplished in two ways.

One, by using the `delop` property

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.delop.cross(C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

```
>>> (C.delop ^ C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Or by using the dedicated function

```
>>> from diofant.vector import curl
>>> curl(C.x*C.y*C.z*C.i, C)
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector \mathbf{F} , denoted by $\nabla \cdot \mathbf{F}$ is given by:

$$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$$

where U , V and W denote the X , Y and Z components of \mathbf{F} respectively.

Computing the divergence of a vector field in *diofant.vector* (page 1193) can be accomplished in two ways.

One, by using the `delop` property

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.delop.dot(C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
>>> (C.delop & C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

Or by using the dedicated function

```
>>> from diofant.vector import divergence
>>> divergence(C.x*C.y*C.z*(C.i + C.j + C.k), C)
C.x*C.y + C.x*C.z + C.y*C.z
```

Gradient

Consider a scalar field $f(x, y, z)$ in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of f with respect to x , y and z in the X , Y and Z axes respectively.

In the 3D Cartesian system, the divergence of a scalar field f , denoted by ∇f is given by -

$$\nabla f = \frac{\partial f}{\partial x} \hat{\mathbf{i}} + \frac{\partial f}{\partial y} \hat{\mathbf{j}} + \frac{\partial f}{\partial z} \hat{\mathbf{k}}$$

Computing the divergence of a vector field in *diofant.vector* (page 1193) can be accomplished in two ways.

One, by using the `delop` property

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.delop.gradient(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```



```
>>> C.delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Or by using the dedicated function

```
>>> from diofant.vector import gradient
>>> gradient(C.x*C.y*C.z, C)
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Directional Derivative

Apart from the above three common applications of ∇ , it is also possible to compute the directional derivative of a field wrt a Vector in *diofant.vector* (page 1193).

By definition, the directional derivative of a field \mathbf{F} along a vector v at point x represents the instantaneous rate of change of \mathbf{F} moving through x with the velocity v . It is represented mathematically as: $(\vec{v} \cdot \nabla) \mathbf{F}(x)$.

Directional derivatives of vector and scalar fields can be computed in *diofant.vector* (page 1193) using the `delop` property of `CoordSysCartesian`.

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> vel = C.i + C.j + C.k
>>> scalar_field = C.x*C.y*C.z
>>> vector_field = C.x*C.y*C.z*C.i
>>> (vel.dot(C.delop))(scalar_field)
C.x*C.y + C.x*C.z + C.y*C.z
>>> (vel & C.delop)(vector_field)
(C.x*C.y + C.x*C.z + C.y*C.z)*C.i
```

Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path travelled. A conservative vector field is also said to be 'irrotational', since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in *diofant.vector* (page 1193), the `is_conservative` function can be used.

```
>>> from diofant.vector import CoordSysCartesian, is_conservative
>>> R = CoordSysCartesian('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_conservative(field)
True
>>> curl(field, R)
0
```

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in *diofant.vector* (page 1193), the `is_solenoidal` function can be used.

```
>>> from diofant.vector import CoordSysCartesian, is_solenoidal
>>> R = CoordSysCartesian('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_solenoidal(field)
True
>>> divergence(field, R)
0
```

Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the ‘scalar potential field’ corresponding to the aforementioned conservative field.

The `scalar_potential` function in *diofant.vector* (page 1193) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from diofant.vector import CoordSysCartesian, scalar_potential
>>> R = CoordSysCartesian('R')
>>> conservative_field = 4*R.x*R.y*R.z*R.i + 2*R.x**2*R.z*R.j + 2*R.x**2*R.y*R.k
>>> scalar_potential(conservative_field, R)
2*R.x**2*R.y*R.z
```

Providing a non-conservative vector field as an argument to `scalar_potential` raises a `ValueError`.

The scalar potential difference, or simply ‘potential difference’, corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in *diofant.vector* (page 1193).

```
>>> from diofant.vector import CoordSysCartesian, Point
>>> from diofant.vector import scalar_potential_difference
>>> R = CoordSysCartesian('R')
>>> P = R.origin.locate_new('P', 1*R.i + 2*R.j + 3*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
4
```

If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

General examples of usage

This section details the solution of two basic problems in vector math/calculus using the *diofant.vector* (page 1193) package.

Quadrilateral problem

The Problem

OABC is any quadrilateral in 3D space. P is the midpoint of OA, Q is the midpoint of AB, R is the midpoint of BC and S is the midpoint of OC. Prove that PQ is parallel to SR

Solution

The solution to this problem demonstrates the usage of Point, and basic operations on Vector.

Define a coordinate system

```
>>> from diofant.vector import CoordSysCartesian
>>> Sys = CoordSysCartesian('Sys')
```

Define point O to be Sys' origin. We can do this without loss of generality

```
>>> 0 = Sys.origin
```

Define point A with respect to O

```
>>> from diofant import symbols
>>> a1, a2, a3 = symbols('a1 a2 a3')
>>> A = 0.locate_new('A', a1*Sys.i + a2*Sys.j + a3*Sys.k)
```

Similarly define points B and C

```
>>> b1, b2, b3 = symbols('b1 b2 b3')
>>> B = 0.locate_new('B', b1*Sys.i + b2*Sys.j + b3*Sys.k)
>>> c1, c2, c3 = symbols('c1 c2 c3')
>>> C = 0.locate_new('C', c1*Sys.i + c2*Sys.j + c3*Sys.k)
```

P is the midpoint of OA. Lets locate it with respect to O (you could also define it with respect to A).

```
>>> P = 0.locate_new('P', A.position_wrt(0) + (0.position_wrt(A) / 2))
```

Similarly define points Q, R and S as per the problem definitions.

```
>>> Q = A.locate_new('Q', B.position_wrt(A) / 2)
>>> R = B.locate_new('R', C.position_wrt(B) / 2)
>>> S = 0.locate_new('R', C.position_wrt(0) / 2)
```

Now compute the vectors in the directions specified by PQ and SR.

```
>>> PQ = Q.position_wrt(P)
>>> SR = R.position_wrt(S)
```

Compute cross product

```
>>> PQ.cross(SR)
0
```

Since the cross product is a zero vector, the two vectors have to be parallel, thus proving that $PQ \parallel SR$.

Third product rule for Del operator

See

The Problem

Prove the third rule - $\nabla \cdot (f\vec{v}) = f(\nabla \cdot \vec{v}) + \vec{v} \cdot (\nabla f)$

Solution

Start with a coordinate system

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
```

The scalar field f and the measure numbers of the vector field \vec{v} are all functions of the coordinate variables of the coordinate system in general. Hence, define Diofant functions that way.

```
>>> from diofant import symbols, Function
>>> v1, v2, v3, f = symbols('v1 v2 v3 f', cls=Function)
```

v_1, v_2 and v_3 are the X, Y and Z components of the vector field respectively.

Define the vector field as `vfield` and the scalar field as `sfield`.

```
>>> vfield = v1(C.x, C.y, C.z)*C.i + v2(C.x, C.y, C.z)*C.j + v3(C.x, C.y, C.z)*C.k
>>> ffield = f(C.x, C.y, C.z)
```

Construct the expression for the LHS of the equation using `C.delop`.

```
>>> lhs = (C.delop.dot(ffield * vfield)).doit()
```

Similarly, the RHS would be defined.

```
>>> rhs = ((vfield.dot(C.delop(ffield))) + (ffield * (C.delop.dot(vfield))))).doit()
```

Now, to prove the product rule, we would just need to equate the expanded and simplified versions of the lhs and the rhs, so that the Diofant expressions match.

```
>>> lhs.expand().simplify() == rhs.expand().simplify()
True
```

Thus, the general form of the third product rule mentioned above can be proven using *diofant.vector* (page 1193).

3.33.2 Vector API

Essential Classes in `diofant.vector` (doctrings)

`CoordSysCartesian`

```
class diofant.vector.coordsysrect.CoordSysCartesian(name, location=None,
                                                    rotation_matrix=None,
                                                    parent=None, vector_names=None, variable_names=None,
                                                    latex_vects=None, pretty_vects=None,
                                                    latex_scalars=None, pretty_scalars=None)
```

Represents a coordinate system in 3-D space.

```
__init__(name, location=None, rotation_matrix=None, parent=None,
          vector_names=None, variable_names=None, latex_vects=None,
          pretty_vects=None, latex_scalars=None, pretty_scalars=None)
```

The orientation/location parameters are necessary if this system is being defined at a certain orientation or location wrt another.

Parameters `name` : str

The name of the new `CoordSysCartesian` instance.

location : Vector

The position vector of the new system's origin wrt the parent instance.

rotation_matrix : Diofant ImmutableMatrix

The rotation matrix of the new coordinate system with respect to the parent. In other words, the output of `new_system.rotation_matrix(parent)`.

parent : `CoordSysCartesian`

The coordinate system wrt which the orientation/location (or both) is being defined.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

```
locate_new(name, position, vector_names=None, variable_names=None)
```

Returns a `CoordSysCartesian` with its origin located at the given position wrt this coordinate system's origin.

Parameters `name` : str

The name of the new `CoordSysCartesian` instance.

position : Vector

The position vector of the new system's origin wrt this one.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> A = CoordSysCartesian('A')
>>> B = A.locate_new('B', 10 * A.i)
>>> B.origin.position_wrt(A.origin)
10*A.i
```

orient_new(*name*, *orienters*, *location=None*, *vector_names=None*, *variable_names=None*)

Creates a new `CoordSysCartesian` oriented in the user-specified way with respect to this system.

Please refer to the documentation of the orienter classes for more information about the orientation procedure.

Parameters *name* : str

The name of the new `CoordSysCartesian` instance.

orienters : iterable/Orienter

An Orienter or an iterable of Orienters for orienting the new coordinate system. If an Orienter is provided, it is applied to get the new system. If an iterable is provided, the orienters will be applied in the order in which they appear in the iterable.

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSysCartesian('N')
```

Using an `AxisOrienter`

```
>>> from diofant.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> A = N.orient_new('A', (axis_orienter, ))
```

Using a `BodyOrienter`

```
>>> from diofant.vector import BodyOrienter
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> B = N.orient_new('B', (body_orienter, ))
```

Using a SpaceOrienter

```
>>> from diofant.vector import SpaceOrienter
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> C = N.orient_new('C', (space_orienter, ))
```

Using a QuaternionOrienter

```
>>> from diofant.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> D = N.orient_new('D', (q_orienter, ))
```

orient_new_axis(*name*, *angle*, *axis*, *location=None*, *vector_names=None*, *variable_names=None*)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a Diofant expr scalar, and the axis is supplied as a Vector.

Parameters **name** : string

The name of the new coordinate system

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q1 = symbols('q1')
>>> N = CoordSysCartesian('N')
>>> B = N.orient_new_axis('B', q1, N.i + 2 * N.j)
```

orient_new_body(*name*, *angle1*, *angle2*, *angle3*, *rotation_order*, *location=None*, *vector_names=None*, *variable_names=None*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see http://en.wikipedia.org/wiki/Euler_angles.

Parameters **name** : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSysCartesian('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> D = N.orient_new_body('D', q1, q2, q3, '123')
```

is same as

```
>>> D = N.orient_new_axis('D', q1, N.i)
>>> D = D.orient_new_axis('D', q2, D.j)
>>> D = D.orient_new_axis('D', q3, D.k)
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> B = N.orient_new_body('B', q1, q2, q3, '123')
>>> B = N.orient_new_body('B', q1, q2, 0, 'ZXZ')
>>> B = N.orient_new_body('B', 0, 0, 0, 'YXX')
```

orient_new_quaternion(*name, q0, q1, q2, q3, location=None, vector_names=None, variable_names=None*)

Quaternion orientation orients the new CoordSysCartesian with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q_0 = \cos(\theta/2)$

$q_1 = \lambda_x \sin(\theta/2)$

$q_2 = \lambda_y \sin(\theta/2)$

$q_3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters name : string

The name of the new coordinate system

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSysCartesian('N')
>>> B = N.orient_new_quaternion('B', q0, q1, q2, q3)
```

orient_new_space(name, angle1, angle2, angle3, rotation_order, location=None, vector_names=None, variable_names=None)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters name : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

See also:

[diofant.vector.coordsysrect.CoordSysCartesian.orient_new_body](#) (page 1211)
method to orient via Euler angles

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSysCartesian('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> D = N.orient_new_space('D', q1, q2, q3, '312')
```

is same as

```
>>> B = N.orient_new_axis('B', q1, N.i)
>>> C = B.orient_new_axis('C', q2, N.j)
>>> D = C.orient_new_axis('D', q3, N.k)
```

position_wrt(*other*)

Returns the position vector of the origin of this coordinate system with respect to another Point/CoordSysCartesian.

Parameters *other* : Point/CoordSysCartesian

If *other* is a Point, the position of this system's origin wrt it is returned. If its an instance of CoordSyRect, the position wrt its origin is returned.

Examples

```
>>> from diofant.vector import Point, CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> N1 = N.locate_new('N1', 10 * N.i)
>>> N.position_wrt(N1)
(-10)*N.i
```

rotation_matrix(*other*)

Returns the direction cosine matrix(DCM), also known as the 'rotation matrix' of this coordinate system with respect to another system.

If v_a is a vector defined in system 'A' (in matrix format) and v_b is the same vector defined in system 'B', then $v_a = A.rotation_matrix(B) * v_b$.

A Diofant Matrix is returned.

Parameters *other* : CoordSysCartesian

The system which the DCM is generated to.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q1 = symbols('q1')
>>> N = CoordSysCartesian('N')
```

```
>>> A = N.orient_new_axis('A', q1, N.i)
>>> N.rotation_matrix(A)
Matrix([
[1,      0,      0],
[0, cos(q1), -sin(q1)],
[0, sin(q1),  cos(q1)])])
```

scalar_map(*other*)

Returns a dictionary which expresses the coordinate variables (base scalars) of this frame in terms of the variables of *other* frame.

Parameters *otherframe* : CoordSysCartesian

The other system to map the variables to.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import Symbol, cos, sin
>>> A = CoordSysCartesian('A')
>>> q = Symbol('q')
>>> B = A.orient_new_axis('B', q, A.k)
>>> A.scalar_map(B) == {A.x: B.x*cos(q) - B.y*sin(q), A.y: B.x*sin(q) + B.
  ↪ y*cos(q), A.z: B.z}
True
```

Vector**class diofant.vector.vector.Vector**

Super class for all Vector classes. Ideally, neither this class nor any of its subclasses should be instantiated by the user.

components

Returns the components of this vector in the form of a Python dictionary mapping BaseVector instances to the corresponding measure numbers.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.components == {C.i: 3, C.j: 4, C.k: 5}
True
```

cross(*other*)

Returns the cross product of this Vector with another Vector or Dyadic instance. The cross product is a Vector, if 'other' is a Vector. If 'other' is a Dyadic, this returns a Dyadic instance.

Parameters *other*: Vector/Dyadic

The Vector or Dyadic we are crossing with.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.i.cross(C.j)
C.k
>>> C.i ^ C.i
0
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v ^ C.i
5*C.j + (-4)*C.k
>>> d = C.i.outer(C.i)
>>> C.j.cross(d)
(-1)*(C.k|C.i)
```

dot(*other*)

Returns the dot product of this Vector, either with another Vector, or a Dyadic, or a Del operator. If 'other' is a Vector, returns the dot product scalar (Diofant expression). If 'other' is a Dyadic, the dot product is returned as a Vector. If 'other' is an instance of Del, returns the directional derivate operator as a Python function. If this function is applied to a scalar expression, it returns the directional derivative of the scalar field wrt this Vector.

Parameters other: Vector/Dyadic/Del

The Vector or Dyadic we are dotting with, or a Del operator .

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.i.dot(C.j)
0
>>> C.i & C.i
1
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.dot(C.k)
5
>>> (C.i & C.delop)(C.x*C.y*C.z)
C.y*C.z
>>> d = C.i.outer(C.i)
>>> C.i.dot(d)
C.i
```

magnitude()

Returns the magnitude of this vector.

normalize()

Returns the normalized version of this vector.

outer(*other*)

Returns the outer product of this vector with another, in the form of a Dyadic instance.

Parameters other : Vector

The Vector with respect to which the outer product is to be computed.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```

`separate()`

The constituents of this vector in different coordinate systems, as per its definition. Returns a dict mapping each `CoordSysCartesian` to the corresponding constituent Vector.

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> R1 = CoordSysCartesian('R1')
>>> R2 = CoordSysCartesian('R2')
>>> v = R1.i + R2.i
>>> v.separate() == {R1: R1.i, R2: R2.i}
True
```

`to_matrix(system)`

Returns the matrix form of this vector with respect to the specified coordinate system.

Parameters `system` : `CoordSysCartesian`

The system wrt which the matrix form is to be computed

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> from diofant.abc import a, b, c
>>> v = a*C.i + b*C.j + c*C.k
>>> v.to_matrix(C)
Matrix([
[a],
[b],
[c]])
```

Dyadic

class `diofant.vector.dyadic.Dyadic`

Super class for all Dyadic-classes.

References

[R589] (page 1278), [R590] (page 1278)

components

Returns the components of this dyadic in the form of a Python dictionary mapping BaseDyadic instances to the corresponding measure numbers.

cross(*other*)

Returns the cross product between this Dyadic, and a Vector, as a Vector instance.

Parameters **other** : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> d = N.i.outer(N.i)
>>> d.cross(N.j)
(N.i|N.k)
```

dot(*other*)

Returns the dot product(also called inner product) of this Dyadic, with another Dyadic or Vector. If 'other' is a Dyadic, this returns a Dyadic. Else, it returns a Vector (unless an error is encountered).

Parameters **other** : Dyadic/Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> D1 = N.i.outer(N.j)
>>> D2 = N.j.outer(N.j)
>>> D1.dot(D2)
(N.i|N.j)
>>> D1.dot(N.j)
N.i
```

to_matrix(*system, second_system=None*)

Returns the matrix form of the dyadic with respect to one or two coordinate systems.

Parameters **system** : CoordSysCartesian

The coordinate system that the rows and columns of the matrix correspond to. If a second system is provided, this only corresponds to the rows of the matrix.

second_system : CoordSysCartesian, optional, default=None

The coordinate system that the columns of the matrix correspond to.

Examples

```

>>> from diofant.vector import CoordSysCartesian
>>> N = CoordSysCartesian('N')
>>> v = N.i + 2*N.j
>>> d = v.outer(N.i)
>>> d.to_matrix(N)
Matrix([
[1, 0, 0],
[2, 0, 0],
[0, 0, 0]])
>>> from diofant import Symbol
>>> q = Symbol('q')
>>> P = N.orient_new_axis('P', q, N.k)
>>> d.to_matrix(N, P)
Matrix([
[ cos(q),  -sin(q),  0],
[2*cos(q), -2*sin(q), 0],
[      0,          0, 0]])

```

Del

class diofant.vector.deloperator.Del

Represents the vector differential operator, usually represented in mathematical expressions as the ‘nabla’ symbol.

cross(*vect*, *doit*=False)

Represents the cross product between this operator and a given vector - equal to the curl of the vector field.

Parameters *vect* : Vector

The vector whose curl is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```

>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> C.delop.cross(v, doit = True)
(-C.x*C.y + C.x*C.z)*C.i + (C.x*C.y - C.y*C.z)*C.j + (-C.x*C.z + C.y*C.z)*C.k
>>> (C.delop ^ C.i).doit()
0

```

dot(*vect*, *doit*=False)

Represents the dot product between this operator and a given vector - equal to the divergence of the vector field.

Parameters *vect* : Vector

The vector whose divergence is to be calculated.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.delop.dot(C.x*C.i)
Derivative(C.x, C.x)
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> (C.delop & v).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

gradient(*scalar_field*, *doit=False*)

Returns the gradient of the given scalar field, as a Vector instance.

Parameters **scalar_field** : Diofant expression

The scalar field to calculate the gradient of.

doit : bool

If True, the result is returned after calling `.doit()` on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> C = CoordSysCartesian('C')
>>> C.delop.gradient(9)
(Derivative(9, C.x))*C.i + (Derivative(9, C.y))*C.j + (Derivative(9, C.z))*C.k
>>> C.delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Orienter classes (docstrings)

Orienter

class `diofant.vector.orienters.Orienter`

Super-class for all orienter classes.

rotation_matrix()

The rotation matrix corresponding to this orienter instance.

AxisOrienter

class `diofant.vector.orienters.AxisOrienter`(*angle*, *axis*)

Class to denote an axis orienter.

__init__(*angle*, *axis*)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a Diofant expr scalar, and the axis is supplied as a Vector.

Parameters **angle** : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q1 = symbols('q1')
>>> N = CoordSysCartesian('N')
>>> from diofant.vector import AxisOrienter
>>> orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> B = N.orient_new('B', (orienter, ))
```

rotation_matrix(*system*)

The rotation matrix corresponding to this orienter instance.

Parameters **system** : CoordSysCartesian

The coordinate system wrt which the rotation matrix is to be computed

BodyOrienter

class diofant.vector.orienters.**BodyOrienter**(*angle1, angle2, angle3, rot_order*)

Class to denote a body-orienter.

__init__(*angle1, angle2, angle3, rot_order*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see http://en.wikipedia.org/wiki/Euler_angles.

Parameters **angle1, angle2, angle3** : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from diofant.vector import CoordSysCartesian, BodyOrienter
>>> from diofant import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSysCartesian('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> body_orienter = BodyOrientor(q1, q2, q3, '123')
>>> D = N.orient_new('D', (body_orienter, ))
```

is same as

```
>>> from diofant.vector import AxisOrientor
>>> axis_orienter1 = AxisOrientor(q1, N.i)
>>> D = N.orient_new('D', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrientor(q2, D.j)
>>> D = D.orient_new('D', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrientor(q3, D.k)
>>> D = D.orient_new('D', (axis_orienter3, ))
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> body_orienter1 = BodyOrientor(q1, q2, q3, '123')
>>> body_orienter2 = BodyOrientor(q1, q2, 0, 'ZXZ')
>>> body_orienter3 = BodyOrientor(0, 0, 0, 'YXX')
```

SpaceOrientor

class `diofant.vector.orienters.SpaceOrientor`(*angle1, angle2, angle3, rot_order*)

Class to denote a space-orienter.

__init__(*angle1, angle2, angle3, rot_order*)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters *angle1, angle2, angle3* : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

See also:

[BodyOrientor](#) (page 1221) Orientor to orient systems wrt Euler angles.

Examples

```
>>> from diofant.vector import CoordSysCartesian, SpaceOrientor
>>> from diofant import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSysCartesian('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> space_orienter = SpaceOrientor(q1, q2, q3, '312')
>>> D = N.orient_new('D', (space_orienter, ))
```

is same as

```
>>> from diofant.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> B = N.orient_new('B', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, N.j)
>>> C = B.orient_new('C', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, N.k)
>>> D = C.orient_new('C', (axis_orienter3, ))
```

QuaternionOrienter

```
class diofant.vector.orienters.QuaternionOrienter(angle1, angle2, angle3,
                                                  rot_order)
```

Class to denote a quaternion-orienter.

__init__ (*angle1, angle2, angle3, rot_order*)

Quaternion orientation orients the new CoordSysCartesian with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation is described by four parameters:

$q_0 = \cos(\theta/2)$

$q_1 = \lambda_x \sin(\theta/2)$

$q_2 = \lambda_y \sin(\theta/2)$

$q_3 = \lambda_z \sin(\theta/2)$

Quaternion does not take in a rotation order.

Parameters q_0, q_1, q_2, q_3 : Expr

The quaternions to rotate the coordinate system by

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSysCartesian('N')
>>> from diofant.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> B = N.orient_new('B', (q_orienter, ))
```

Essential Functions in diofant.vector (docstrings)

matrix_to_vector

matrix_to_vector (*matrix, system*)

Converts a vector in matrix form to a Vector instance.

It is assumed that the elements of the Matrix represent the measure numbers of the components of the vector along basis vectors of 'system'.

Parameters *matrix* : Diofant Matrix, Dimensions: (3, 1)

The matrix to be converted to a vector

system : CoordSysCartesian

The coordinate system the vector is to be defined in

Examples

```
>>> from diofant import ImmutableMatrix as Matrix
>>> m = Matrix([1, 2, 3])
>>> from diofant.vector import CoordSysCartesian, matrix_to_vector
>>> C = CoordSysCartesian('C')
>>> v = matrix_to_vector(m, C)
>>> v
C.i + 2*C.j + 3*C.k
>>> v.to_matrix(C) == m
True
```

express

`diofant.vector.express`(*expr*, *system*, *system2=None*, *variables=False*)

Global function for 'express' functionality.

Re-expresses a Vector, Dyadic or scalar(diofant-able) in the given coordinate system.

If 'variables' is True, then the coordinate variables (base scalars) of other coordinate systems present in the vector/scalar field or dyadic are also substituted in terms of the base scalars of the given system.

Parameters **expr** : Vector/Dyadic/scalar(diofant-able)

The expression to re-express in CoordSysCartesian 'system'

system: CoordSysCartesian

The coordinate system the expr is to be expressed in

system2: CoordSysCartesian

The other coordinate system required for re-expression (only for a Dyadic Expr)

variables : boolean

Specifies whether to substitute the coordinate variables present in expr, in terms of those of parameter system

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant import Symbol, cos, sin
>>> N = CoordSysCartesian('N')
>>> q = Symbol('q')
>>> B = N.orient_new_axis('B', q, N.k)
>>> from diofant.vector import express
>>> express(B.i, N)
(cos(q))*N.i + (sin(q))*N.j
>>> express(N.x, B, variables=True)
B.x*cos(q) - B.y*sin(q)
```

```
>>> d = N.i.outer(N.i)
>>> express(d, B, N) == (cos(q))*(B.i|N.i) + (-sin(q))*(B.j|N.i)
True
```

curl

`diofant.vector.curl`(*vect*, *coord_sys*)

Returns the curl of a vector field computed wrt the base scalars of the given coordinate system.

Parameters *vect* : Vector

The vector operand

coord_sys : CoordSysCartesian

The coordinate system to calculate the curl in

Examples

```
>>> from diofant.vector import CoordSysCartesian, curl
>>> R = CoordSysCartesian('R')
>>> v1 = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> curl(v1, R)
0
>>> v2 = R.x*R.y*R.z*R.i
>>> curl(v2, R)
R.x*R.y*R.j + (-R.x*R.z)*R.k
```

divergence

`diofant.vector.divergence`(*vect*, *coord_sys*)

Returns the divergence of a vector field computed wrt the base scalars of the given coordinate system.

Parameters *vect* : Vector

The vector operand

coord_sys : CoordSysCartesian

The coordinate system to calculate the divergence in

Examples

```
>>> from diofant.vector import CoordSysCartesian, divergence
>>> R = CoordSysCartesian('R')
>>> v1 = R.x*R.y*R.z * (R.i+R.j+R.k)
>>> divergence(v1, R)
R.x*R.y + R.x*R.z + R.y*R.z
>>> v2 = 2*R.y*R.z*R.j
>>> divergence(v2, R)
2*R.z
```

gradient

`diofant.vector.gradient`(*scalar*, *coord_sys*)

Returns the vector gradient of a scalar field computed wrt the base scalars of the given coordinate system.

Parameters *scalar* : Diofant Expr

The scalar field to compute the gradient of

coord_sys : CoordSysCartesian

The coordinate system to calculate the gradient in

Examples

```
>>> from diofant.vector import CoordSysCartesian, gradient
>>> R = CoordSysCartesian('R')
>>> s1 = R.x*R.y*R.z
>>> gradient(s1, R)
R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> s2 = 5*R.x**2*R.z
>>> gradient(s2, R)
10*R.x*R.z*R.i + 5*R.x**2*R.k
```

is_conservative

`diofant.vector.is_conservative`(*field*)

Checks if a field is conservative.

Parameters *field* : Vector

The field to check for conservative property

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant.vector import is_conservative
>>> R = CoordSysCartesian('R')
>>> is_conservative(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_conservative(R.z*R.j)
False
```

is_solenoidal

`diofant.vector.is_solenoidal`(*field*)

Checks if a field is solenoidal.

Parameters *field* : Vector

The field to check for solenoidal property

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant.vector import is_solenoidal
>>> R = CoordSysCartesian('R')
>>> is_solenoidal(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_solenoidal(R.y * R.j)
False
```

scalar_potential

`diofant.vector.scalar_potential`(*field*, *coord_sys*)

Returns the scalar potential function of a field in a given coordinate system (without the added integration constant).

Parameters **field** : Vector

The vector field whose scalar potential function is to be calculated

coord_sys : CoordSysCartesian

The coordinate system to do the calculation in

Examples

```
>>> from diofant.vector import CoordSysCartesian
>>> from diofant.vector import scalar_potential, gradient
>>> R = CoordSysCartesian('R')
>>> scalar_potential(R.k, R) == R.z
True
>>> scalar_field = 2*R.x**2*R.y*R.z
>>> grad_field = gradient(scalar_field, R)
>>> scalar_potential(grad_field, R)
2*R.x**2*R.y*R.z
```

scalar_potential_difference

`diofant.vector.scalar_potential_difference`(*field*, *coord_sys*, *point1*, *point2*)

Returns the scalar potential difference between two points in a certain coordinate system, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at point2) - (potential at point1)

The position vectors of the two Points are calculated wrt the origin of the coordinate system provided.

Parameters **field** : Vector/Expr

The field to calculate wrt

coord_sys : CoordSysCartesian

The coordinate system to do the calculations in

point1 : Point

The initial Point in given coordinate system

position2 : Point

The second Point in the given coordinate system

Examples

```
>>> from diofant.vector import CoordSysCartesian, Point
>>> from diofant.vector import scalar_potential_difference
>>> R = CoordSysCartesian('R')
>>> P = R.origin.locate_new('P', R.x*R.i + R.y*R.j + R.z*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
2*R.x**2*R.y
>>> Q = R.origin.locate_new('Q', 3*R.i + R.j + 2*R.k)
>>> scalar_potential_difference(vectfield, R, P, Q)
-2*R.x**2*R.y + 18
```

3.33.3 References for Vector

DEVELOPER'S GUIDE

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.

4.1 Guidelines for Contributing

This project adheres to [No Code of Conduct](#). Contributions will be judged by their technical merit. Nothing else matters.

4.1.1 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 reproducible test case as an example.
4. Include any relevant details of your local setup (i.e. Python version, installed libraries).

4.1.2 Contributing Code

All work should be submitted via [Pull Requests \(PR\)](#).

1. PR can be submitted as soon as there is code worth discussing.
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.
4. Please conform to [PEP 8](#) for code formatting, check code quality with [flake8](#) tool:

```
$ python setup.py flake8
```

5. PR should include tests:
 - (a) Bugfixes should include regression tests.
 - (b) All new functionality should be tested, every new line should be covered by tests.

- (c) All new public interfaces (methods, functions or classes) should have doctests showing how to use them.
 - (d) Keep in mind, doctests are not tests. Think of them as examples that happen to be tested.
6. Usually, it's good idea to be sure that all existing tests pass and you don't break anything, so please run:

```
$ python setup.py test
```

4.1.3 License

By submitting a PR, you agree to license your code under the BSD license and [LGPL](#).

4.2 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 [Diofant's issues](#). Functions for regression tests should be named like [test_sympyissue_123](#) and [test_diofantissue_123](#), respectively.

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 history - does mean [issue sympy/sympy#3222](#) on Github.

4.3 Versioning and Release Procedure

We use standard [Semantic Versioning](#) MAJOR.MINOR.MAINTENANCE numbering scheme, but adopt [PEP 440](#) for alpha ("aN" suffix), beta ("bN" suffix) and release candidates ("rcN" suffix).

Releasing a new version is done as follows:

1. Increase `__version__` in `diofant/__init__.py`.
2. Commit version update:

```
$ git commit -am 'Bump version to X.Y.Z'
```

3. Merge commit to the master branch.

4. Tag merge commit and push release tag:

```
$ git pull
$ git tag -s vX.Y.Z
$ git push origin vX.Y.Z
```

4.4 How to Build Documentation

To make the html documentation, first install the Diofant *from sources* (page 1) with docs extra requirement, then enter the project directory and run:

```
python setup.py build_sphinx
```

Documentation tree will be located in the build/sphinx/html/ directory.

ABOUT

The [Diofant](#) project 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, when development was in svn.

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.

5.1 License

Unless stated otherwise, all files in the Diofant project are licensed using the new BSD license:

```
Copyright (c) 2006-2016 SymPy Development Team,  
2013-2016 Sergey B Kirpichev
```

```
All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions are met:
```

- a. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- b. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- c. Neither the name of Diofant, or SymPy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"  
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  
ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR  
ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL  
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER  
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT  
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY  
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH  
DAMAGE.
```

5.2 SymPy Development Team

SymPy is a team project and it was developed by a lot of people.

All people who contributed in the Diofant project via SymPy by sending at least a patch or more (in the order of the date of their first contribution) are listed here. This list was generated automatically from the git history by SymPy tools (people with a * next to their names are not found in the metadata of the git history).

Please note that this list is incomplete intentionally, because some former SymPy developers don't want to be mentioned in the context of the SymPy or derived projects.

- Ondřej Čertík
- Fabian Pedregosa
- Mateusz Paprocki
- Jason Gedge
- Marc-Etienne M. Leveille*
- Brian Jorgensen
- Robert Schwarz
- Fredrik Johansson
- Chris Wu
- Pearu Peterson
- Ulrich Hecht*
- Sebastian Krämer
- Goutham Lakshminarayan
- David Lawrence
- Jaroslav Tworek
- David Marek
- Bernhard R. Link
- Andrej Tokarčík
- Or Dvory
- Saroj Adhikari
- Pauli Virtanen
- Robert Kern
- James Aspnes
- Jurjen N.E. Bos
- Nimish Telang
- Abderrahim Kitouni
- Pan Peng
- Friedrich Hagedorn
- Elrond der Elbenfuerst
- Rizgar Mella

- Felix Kaiser
- Roberto Nobrega
- David Roberts
- Vinzent Steinberg
- Case Van Horses*
- Riccardo Gori
- Štěpán Roučka
- Ali Raza Syed
- Stefano Maggiolo
- Robert Cimrman
- Bastian Weber
- Sebastian Krause
- Dan*
- Sebastian Kreft
- Alan Bromborsky
- Boris Timokhin
- Robert (average.programmer)
- Andy R. Terrel
- Hubert Tsang
- Konrad Meyer
- Henrik Johansson
- Priit Laes
- Freddie Witherden
- Brian E. Granger
- Andrew Straw
- Kaifeng Zhu
- Ted Horst
- Aaron Meurer
- Akshay Srinivasan
- Andrew Docherty
- Barry Wardell
- Tomasz Buchert
- Vinay Kumar
- Johann Cohen-Tanugi
- Jochen Voss
- Luke Peterson
- Thomas Sidoti
- Chris Smith

- Florian Mickler
- Nicolas Pourcelot
- Ben Goodrich
- Toon Verstraelen
- Ronan Lamy
- James Abbatiello
- Ryan Krauss
- Bill Flynn
- Jorn Baayen
- Eh Tan
- Renato Coutinho
- Kevin Goodsell
- Oscar Benjamin
- Øyvind Jensen
- Julio Idichekop Filho
- Chu-Ching Huang*
- Łukasz Pankowski
- Fernando Perez
- Raffaele De Feo
- Christian Muise
- Matt Curry
- Kazuo Thow
- Christian Schubert
- Jezreel Ng
- Matthew Brett
- James Pearson
- Addison Cugini
- Nicholas J.S. Kinar
- Harold Erbin
- Thomas Dixon
- Cristóvão Sousa
- Andre de Fortier Smit
- Mark Dewing
- Alexey U. Gudchenko
- Gary Kerr
- Sherjil Ozair
- Oleksandr Gituliar
- Sean Vig

- Prafullkumar P. Tale
- Vladimir Perić
- Tom Bachmann
- Yuri Karadzhov
- Vladimir Lagunov
- Matthew Rocklin
- Saptarshi Mandal
- Gilbert Gede
- Anatolii Koval
- Tomo Lazovich
- Pavel Fedotov
- Jack McCaffery
- Jeremias Yehdegheo
- Kibeom Kim
- Gregory Ksionda
- Tomáš Bambas
- Raymond Wong
- Luca Weihs
- Shai ‘Deshe’ Wyborski
- Thomas Wiecki
- Óscar Nájera
- Mario Pernici
- Benjamin McDonald
- Sam Magura
- Stefan Krastanov
- Bradley Froehle
- Min Ragan-Kelley
- Emma Hogan
- Nikhil Sarda
- Julien Rioux
- Roberto Colistete, Jr.
- Raoul Bourquin
- Gert-Ludwig Ingold
- Srinivas Vasudevan
- Jason Moore
- Miha Marolt
- Tim Lahey
- Luis Garcia

- Matt Rajca
- David Li
- Alexandr Gudulin
- Bilal Akhtar
- Grzegorz Świrski
- Matt Habel
- David Ju
- Nichita Utiu
- Nikolay Lazarov
- Steve Anton
- Imran Ahmed Manzoor
- Ljubiša Močić
- Piotr Korgul
- Jim Zhang
- Sam Sleight
- tsmars15
- Chancellor Arkantos
- Stepan Simsa
- Tobias Lenz
- Siddhanathan Shanmugam
- Tiffany Zhu
- Tristan Hume
- Alexey Subach
- Joan Creus
- Geoffry Song
- Puneeth Chaganti
- Marcin Kostrzewa
- Natalia Nawara
- vishal
- Shruti Mangipudi
- Davy Mao
- Swapnil Agarwal
- Kendhia
- jerryma1121
- Joachim Durchholz
- Martin Povišer
- Siddhant Jain
- Kevin Hunter

- Michael Mayorov
- Nathan Alison
- Christian Bühler
- Carsten Knoll
- M R Bharath
- Matthias Toews
- Sergiu Ivanov
- Jorge E. Cardona
- Sanket Agarwal
- Manoj Babu K.
- Sai Nikhil
- Aleksandar Makelov
- Sachin Irukula
- Raphael Michel
- Ashwini Oruganti
- Andreas Kloeckner
- Prateek Papriwal
- Arpit Goyal
- Angadh Nanjangud
- Comer Duncan
- Jens H. Nielsen
- Joseph Dougherty
- marshall2389
- Guru Devanla
- George Waksman
- Alexandr Popov
- Tarun Gaba
- Takafumi Arakaki
- Saurabh Jha
- Rom le Clair
- Angus Griffith
- Timothy Reluga
- Brian Stephanik
- Alexander Eberspächer
- Sachin Joglekar
- Tyler Pirtle
- Vasily Povalyaev
- Colleen Lee

- Matthew Hoff
- Niklas Thörne
- Huijun Mai
- Marek Šuppa
- Ramana Venkata
- Prasoon Shukla
- Stefen Yin
- Thomas Hisch
- Madeleine Ball
- Case Van Horsen
- Mary Clark
- Rishabh Dixit
- Manoj Kumar
- Akshit Agarwal
- CJ Carey
- Patrick Lacasse
- Ananya H
- Tarang Patel
- Christopher Dembia
- Benjamin Fishbein
- Sean Ge
- Amit Jamadagni
- Ankit Agrawal
- Björn Dahlgren
- Christophe Saint-Jean
- Demian Wassermann
- Khagesh Patel
- Stephen Loo
- hm
- Patrick Poitras
- Katja Sophie Hotz
- Varun Joshi
- Chetna Gupta
- Thilina Rathnayake
- Max Hutchinson
- Shravas K Rao
- Matthew Tadd
- Alexander Hirzel

- Randy Heydon
- Oliver Lee
- Seshagiri Prabhu
- Pradyumna
- Erik Welch
- Eric Nelson
- Roland Puntaier
- Chris Conley
- Tim Swast
- Dmitry Batkovich
- Francesco Bonazzi
- Yuriy Demidov
- Rick Muller
- Manish Gill
- Markus Müller
- Amit Saha
- Jeremy
- QuaBoo
- Stefan van der Walt
- David Joyner
- Lars Buitinck
- Alkiviadis G. Akritas
- Vinit Ravishankar
- Mike Boyle
- Heiner Kirchhoffer
- Pablo Puente
- James Fiedler
- Harsh Gupta
- Tuomas Airaksinen
- Paul Strickland
- James Goppert
- rathmann
- Avichal Dayal
- Paul Scott
- Shipra Banga
- Pramod Ch
- Akshay
- Buck Shlegeris

- Jonathan Miller
- Edward
- Rajath S
- Zamrath Nizam
- Aditya Shah
- Rajat Aggarwal
- Sambuddha Basu
- Zeel Shah
- Abhinav Chanda
- Jim Crist
- Sudhanshu Mishra
- Anurag Sharma
- Soumya Dipta Biswas
- Sushant Hiray
- Ben Lucato
- Kunal Arora
- Henry Gebhardt
- Dammina Sahabandu
- Shukla
- Ralph Bean
- richierichrawr
- John Connor
- Juan Luis Cano Rodríguez
- Sahil Shekhawat
- Kundan Kumar
- Stas Kelvich
- sevaader
- Dhruvesh Vijay Parikh
- Venkatesh Halli
- Lennart Fricke
- Vlad Seghete
- shashank-agg
- carstimon
- Pierre Haessig
- Maciej Baranski
- Benjamin Gudehus
- Faisal Anees
- Mark Shoulson

- Robert Johansson
- Kalevi Suominen
- Kaushik Varanasi
- Fawaz Alazemi
- Ambar Mehrotra
- David P. Sanders
- Peter Brady
- John V. Siratt
- Sarwar Chahal
- Nathan Woods
- Colin B. Macdonald
- Marcus Näslund
- Clemens Novak
- Mridul Seth
- Craig A. Stoudt
- Raj
- Mihai A. Ionescu
- immerrr
- Chai Wah Wu
- Leonid Blouvshtein
- Peleg Michaeli
- ck Lux
- zsc347
- Hamish Dickson
- Michael Gallaspy
- Roman Infiaskas
- Duane Nykamp
- Ted Dokos
- Sunny Aggarwal
- Victor Brebenar
- Akshat Jain
- Shivam Vats
- WANG Longqi
- Juan Felipe Osorio
- GitRay
- Lukas Zorich
- Eric Miller
- Cody Herbst

- Nishith Shah
- AMiT Kumar
- Yury G. Kudryashov
- Guillaume Gay
- Ray Cathcart
- Mihir Wadwekar
- Tuan Manh Lai
- Asish Panda
- Darshan Chaudhary
- Alec Kalinin
- Ralf Stephan
- Aaditya Nair
- Jayesh Lahori
- harshil goel
- Luv Agarwal
- Jason Ly
- Lokesh Sharma
- Sartaj Singh
- Chris Swierczewski
- vizietto
- Juha Remes
- Peter Schmidt
- Jiaxing Liang
- Lucas Jones
- Greg Ashton - LianLi
- jennifercw
- Michael Boyle
- Mathew Chong
- Govind Sahai
- Adam Bloomston
- Kyle McDaniel
- Nguyen Truong Duy
- Gaurav Dhingra
- Kevin Ventullo
- mao8
- Richard Otis
- Rich LaSota
- Anton Akhmerov

- David T
- Jason Siefken
- Matthew Thomas
- Sumith1896
- Akshay Nagar
- Yu Kobayashi
- Shashank Kumar
- Tschijnmo TSCHAU
- Jens Jørgen Mortensen
- Dzhelil Rufat
- Prashant Tyagi
- Matthew Davis
- Jacob Garber
- Justin Blythe
- Moo VI
- Keval Shah
- Kshitij Saraogi
- Isuru Fernando
- Meghana Madhyastha
- Tanu Hari Dixit
- Dustin Gadal

5.3 Brief History

[SymPy](#) was started by Ondřej Čertík in 2005, he wrote some code during the summer, then he wrote some more code during the summer 2006. In February 2007, Fabian Pedregosa joined the project and helped fixed many things, contributed documentation and made it alive again. 5 students (Mateusz Paprocki, Brian Jorgensen, Jason Gedge, Robert Schwarz and Chris Wu) improved SymPy incredibly during the summer 2007 as part of the Google Summer of Code (GSoC). Pearu Peterson joined the development during the summer 2007 and he has made SymPy much more competitive and fast (from 10x to 100x) by rewriting the core from scratch. Jurjen N.E. Bos has contributed pretty printing and other patches. Fredrik Johansson has wrote mpmath and contributed a lot of patches.

SymPy has participated in every GSoC since 2007. Moderate amount of SymPy's development has come from GSoC students.

In 2011, Ondřej Čertík stepped down as lead developer, with Aaron Meurer, who also started as a GSoC student, taking his place.

Ondřej Čertík is still active in the community, but is too busy with work and family to play a lead development role. Unfortunately, his remaining activity neither constructive nor productive anymore and SymPy just slowly dies now: most former contributors are inactive now or explicitly leaving this “friendly and welcoming” project.

This unfortunate situation was major reason to fork the SymPy as the [Diofant](#) project. Development in the new project will be open and public, without hidden double standards, centered about good, proved code and not project popularity counters. Here we believe that mathematical correctness is more important than political one.

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.

6.1 Diofant 0.8

7 Nov 2016

6.1.1 New features

- MrvAsympt algorithm to find asymptotic expansion, see *aseries()* (page 67) method and #6. Thanks to Avichal Dayal.
- *findrecur()* (page 283) method to find recurrence relations (with Sister Celine's algorithm), see #15.
- Support for *Pow* (page 100)/*log* (page 331) branch-cuts in limits, see #140.
- Added basic optimization package, see *minimize()* (page 1172) and #108.
- Cartesian product of iterables using Cantor pairing, see *cantor_product()* (page 1144) and #276.
- *Rationals* (page 921) 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 994) now support initial conditions for ODEs, see #307. Thanks to Aaron Meurer.

6.1.2 Major changes

- Depend on *setuptools*, see #44.
- *The Gruntz Algorithm* (page 905) reimplemented correctly, see #68.
- Replaced $\exp(x)$ with E^{**x} internally, see #79.
- Used *srepr()* (page 878) instead of *sstr()* (page 879) 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.

6.1.3 Backwards-incompatible changes

- 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.

6.1.4 Minor changes

- Add support for bidirectional limits, see #10.
- Reimplement *cot* (page 315), see #113.
- A better implementation of *singularities()* (page 1171), 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 1070), 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](#).

6.1.5 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](#).

6.1.6 Issues closed

- [#3](#) Set up documentation on the readthedocs
- [#20](#) Add CONTRIBUTING.rst
- [#24](#) Remove support for some python versions
- [#46](#) Use rtd theme locally
- [#55](#) $\lim_{x \rightarrow -\infty} \frac{x + \exp(x)}{x - 1}$, $x, -\infty$ should be 1
- [#56](#) $\text{gruntz}(\ln(x) - 1, 1 - \sqrt{x}, x, E)$ should be ∞
- [sympy/sympy#9351](#) order-1 series wrong with non-zero expansion point
- [#16](#) `solveset(sinh(x))` doesn't returns all solutions
- [#22](#) Use `py.test` for testing
- [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)

- [#21](#) Remove unsupported and obsoleted modules
- [#124](#) `exp(n*x).subs({exp(x): x})` doesn't work for integer symbol `n`
- [sympy/sympy#9832](#) `x**2 < oo` returns True but `x < oo` un-evaluated for real `x`
- [sympy/sympy#9053](#) `MatMul(2, Matrix(...)).doit()` doesn't do it
- [sympy/sympy#9052](#) `trace(2*A) != 2*Trace(A)` because LHS still has an `MatMul`
- [sympy/sympy#9053](#) `MatMul(2, Matrix(...)).doit()` doesn't do it
- [sympy/sympy#9052](#) `trace(2*A) != 2*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
- [#138](#) Wrong `polylog.eval` for `z=-1`
- [sympy/sympy#10195](#) Simplification bug on alternating series.
- [#143](#) `powsimp((-1)**(odd/2)) != ImaginaryUnit`
- [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
- [#74](#) Trivial limit's of sign fails
- [#31](#) Wrong automatical cancelation of `expr` with `O` terms
- [sympy/sympy#10258](#) Relational involving `Piecewise` evaluates incorrectly as True
- [sympy/sympy#10205](#) 10203: handle `Eq` and `Ne` with `_solve_inequality`
- [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
- [#148](#) Drop py2 support?
- [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
- [#32](#) repr printing oddness
- [sympy/sympy#8241](#) Wrong assumption/result in a parametric limit
- [sympy/sympy#3539](#) Symbol.__call__ should not create a Function
- [#203](#) Wrong hyperexpand(hyper((-6, -7, -5), (-6, -6), 1))
- [sympy/sympy#7216](#) Limits involving branch cuts of elementary functions not handled
- [#19](#) Remove obsoleted/redundant docs
- [sympy/sympy#10503](#) Series return an incorrect result
- [#210](#) Incorrect nseries for cos(x**6)
- [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
- [#238](#) Wrong coeff in *_factor_list with RR domain
- [#236](#) simplify(summation(n/((n+2)*(n+4)*(n+8)), (n, 1, oo))) returns 521/25200
- [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 rightharrow term
- [#241](#) Wrong hyperexpand(hyper((2, 3, 5, 9, 1), (1, 4, 6, 10), 1))
- [#172](#) limit(sin(x)**15,x,0,'-') is slow
- [sympy/sympy#9558](#) Bug with limit
- [#251](#) Random MemoryError in test_gruntz_eval_special
- [sympy/sympy#4949](#) solve_linear_system contains duplicate rref algorithm
- [#213](#) Consolidate all code for solving linear systems
- [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](#) $\operatorname{acot}(-x)$ evaluation
- [sympy/sympy#11011](#) Scientific notation should be delimited for LaTeX
- [#263](#) Workaround decreased coverage due to randomness
- [sympy/sympy#11062](#) Error while simplifying equations containing csc and sec using `trigsimp_groebner`
- [sympy/sympy#10804](#) $1/\lim_{x \rightarrow \infty} (\operatorname{airybi}(x) \cdot \sqrt[4]{x} \cdot \exp(-2x^{3/2}/3))$ is wrong
- [sympy/sympy#11063](#) Some wrong answers from `rsolve`
- [#282](#) Random test failure in master (minimize tests)
- [sympy/sympy#9480](#) `Matrix.rank()` incorrect results
- [#288](#) Wrong rank for matrix with $\det = 0$
- [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
- [#296](#) limit produces bad results with Floats
- [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
- [#294](#) Wrong `rsolve(f(n)-f(n-1)-2*f(n-2)-2*n, f(n))`
- [sympy/sympy#11313](#) Series of Derivative
- [#293](#) `classify_sysode` should be modified to support mass matrix case in LODE
- [#65](#) Docs todo
- [#215](#) Replace `test_code_quality.py` with flake8/pep8 tests
- [sympy/sympy#11290](#) `1st_exact_Integral` wrong result
- [sympy/sympy#10761](#) $(1/(x^{-2} + x^{-3})).\operatorname{series}(x, 0)$ gives wrong result
- [#312](#) `Mod(-x, 2*x)` should be x , not $-x$
- [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](#) `oo**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
- [#135](#) Rename project and adapt imports (sympy -> diofant)
- [#129](#) Use gmpy2 in travis, get rid of gmpy support
- [#133](#) Test regressions with cache on/off
- [#220](#) Update docs/aboutus.rst with more actual info (and move this file?)
- [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)
- [#34](#) assumptions todo
- [#203](#) Add changelog (in sphinx docs)
- [sympy/sympy#11553](#) Polynomial solve with GoldenRatio causes Traceback
- [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
- [#333](#) Expose docs for diofant.interactive (both entry-level and api)
- [#218](#) Remove manualintegrate?
- [sympy/sympy#6572](#) Remove “#doctest: +SKIP” comments on valid docstrings
- [sympy/sympy#10134](#) Remove “raise StopIteration”
- [#329](#) Drop examples/
- [sympy/sympy#11672](#) limit(Rational(-1,2)**k, k, oo) fails
- [#338](#) Rosetta stone for dev's
- [#351](#) Test on CPython 3.6
- [#352](#) Enable testing for DeprecationWarning's
- [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

- [#364](#) Random test failure in combinatorics
- [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](#) $\text{Sum}(1/(n^{**3} - 1), (n, -\infty, -2)).\text{doit}()$ 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 ∞ gives wrong result
- [sympy/sympy#4231](#) add a test for complex integral from wikipedia
- [sympy/sympy#8634](#) `limit(x^{**n}, x, -\infty)` is sometimes wrong
- [sympy/sympy#8481](#) Wrong error raised trying to calculate limit of Poisson PMF
- [sympy/sympy#9956](#) `Union(Interval(-\infty, \infty), FiniteSet(1))` not evaluated
- [sympy/sympy#9747](#) `test_piecewise_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, -\infty)` gives $-\infty$

- [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#7872](#) Substitution in Order fails
- [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 $t^k \cdot \log(t)$
- [#347](#) Search & mention more closed SymPy issues
- [sympy/sympy#8804](#) series expansion of 1/x ignores order parameter
- [sympy/sympy#10728](#) Dummy(commutative=False).is_zero -> False
- [#360](#) Fix "short" references to issues
- [#328](#) Final proofreading of entry-level sphinx docs

See also [full list of closed issues](#) and [full list of merged pull requests](#) in the Diofant repository.

BIBLIOGRAPHY

- [R31] http://en.wikipedia.org/wiki/Algebraic_number
- [R32] http://en.wikipedia.org/wiki/Composite_number
- [R33] http://en.wikipedia.org/wiki/Parity_%28mathematics%29
- [R34] http://en.wikipedia.org/wiki/Extended_real_number_line
- [R35] <http://en.wikipedia.org/wiki/Finite>
- [R36] http://en.wikipedia.org/wiki/Imaginary_number
- [R37] <https://docs.python.org/3/library/math.html#math.isfinite>
- [R38] <http://docs.scipy.org/doc/numpy/reference/generated/numpy.isfinite.html>
- [R39] http://en.wikipedia.org/wiki/Irrational_number
- [R40] http://en.wikipedia.org/wiki/Negative_number
- [R41] http://en.wikipedia.org/wiki/Negative_number
- [R42] http://en.wikipedia.org/wiki/Parity_%28mathematics%29
- [R43] http://en.wikipedia.org/wiki/Prime_number
- [R44] http://en.wikipedia.org/wiki/Real_number
- [R45] http://en.wikipedia.org/wiki/Transcendental_number
- [R46] A New Algorithm for Computing Asymptotic Series - Dominik Gruntz
- [R47] Gruntz thesis - p90
- [R48] http://en.wikipedia.org/wiki/Asymptotic_expansion
- [R49] http://en.wikipedia.org/wiki/Algebraic_expression
- [R50] http://en.wikipedia.org/wiki/Composite_number
- [R51] http://en.wikipedia.org/wiki/Parity_%28mathematics%29
- [R52] http://en.wikipedia.org/wiki/Imaginary_number
- [R53] http://en.wikipedia.org/wiki/Parity_%28mathematics%29
- [R54] http://en.wikipedia.org/wiki/Prime_number
- [R55] <http://en.wikipedia.org/wiki/Zero>
- [R56] http://en.wikipedia.org/wiki/1_%28number%29
- [R57] http://en.wikipedia.org/wiki/%E2%88%921_%28number%29
- [R58] http://en.wikipedia.org/wiki/One_half
- [R59] <http://en.wikipedia.org/wiki/NaN>

[R60] <http://en.wikipedia.org/wiki/Infinity>

[R61] http://en.wikipedia.org/wiki/E_%28mathematical_constant%29

[R62] http://en.wikipedia.org/wiki/Imaginary_unit

[R63] <http://en.wikipedia.org/wiki/Pi>

[R64] http://en.wikipedia.org/wiki/Euler%E2%80%93Mascheroni_constant

[R65] http://en.wikipedia.org/wiki/Catalan%27s_constant

[R66] http://en.wikipedia.org/wiki/Golden_ratio

[R67] <http://en.wikipedia.org/wiki/Exponentiation>

[R68] http://en.wikipedia.org/wiki/Exponentiation#Zero_to_the_power_of_zero

[R69] http://en.wikipedia.org/wiki/Indeterminate_forms

[R70] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <https://docs.python.org/3/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because Diofant cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with x , y , and z being `Symbols`, Python converts the statement (roughly) into these steps:

1. $x > y > z$
2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__bool__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R71] For more information, see these two bug reports:

“Separate boolean and symbolic relationals” [Issue sympy/sympy#4986](#)

“It right $0 < x < 1$?” [Issue sympy/sympy#6059](#)

[R72] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <https://docs.python.org/3/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The and operator coerces each side into a bool, returning the object itself when it short-circuits. The bool of the `-Than` operators will raise `TypeError` on purpose, because Diofant cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with x , y , and z being `Symbols`, Python converts the statement (roughly) into these steps:

1. $x > y > z$

2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__bool__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R73] For more information, see these two bug reports:

“Separate boolean and symbolic relationals” [Issue sympy/sympy#4986](#)

“It right $0 < x < 1$?” [Issue sympy/sympy#6059](#)

[R74] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <https://docs.python.org/3/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The `and` operator coerces each side into a `bool`, returning the object itself when it short-circuits. The `bool` of the `-Than` operators will raise `TypeError` on purpose, because Diofant cannot determine the mathematical ordering of symbolic expressions. Thus, if we were to compute $x > y > z$, with x , y , and z being `Symbols`, Python converts the statement (roughly) into these steps:

1. $x > y > z$
2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__bool__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the `and` operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R75] For more information, see these two bug reports:

“Separate boolean and symbolic relationals” [Issue sympy/sympy#4986](#)

“It right $0 < x < 1$?” [Issue sympy/sympy#6059](#)

[R76] This implementation detail is that Python provides no reliable method to determine that a chained inequality is being built. Chained comparison operators are evaluated pairwise, using “and” logic (see <https://docs.python.org/3/reference/expressions.html#not-in>). This is done in an efficient way, so that each object being compared is only evaluated once and the comparison can short-circuit. For example, $1 > 2 > 3$ is evaluated by Python as $(1 > 2)$ and $(2 > 3)$. The `and` operator coerces each side into a `bool`, returning the object itself when it short-circuits. The `bool` of the `-Than` operators will raise `TypeError` on purpose, because Diofant cannot determine the mathematical ordering of symbolic

expressions. Thus, if we were to compute $x > y > z$, with x , y , and z being Symbols, Python converts the statement (roughly) into these steps:

1. $x > y > z$
2. $(x > y)$ and $(y > z)$
3. `(GreaterThanObject)` and $(y > z)$
4. `(GreaterThanObject.__bool__())` and $(y > z)$
5. `TypeError`

Because of the “and” added at step 2, the statement gets turned into a weak ternary statement, and the first object’s `__bool__` method will raise `TypeError`. Thus, creating a chained inequality is not possible.

In Python, there is no way to override the and operator, or to control how it short circuits, so it is impossible to make something like $x > y > z$ work. There was a PEP to change this, [PEP 335](#), but it was officially closed in March, 2012.

[R77] For more information, see these two bug reports:

“Separate boolean and symbolic relationals” [Issue sympy/sympy#4986](#)

“It right $0 < x < 1$?” [Issue sympy/sympy#6059](#)

[R78] http://reference.wolfram.com/legacy/v5_2/Built-inFunctions/AlgebraicComputation/Calculus/D.html

[R79] <http://mathworld.wolfram.com/Multiple-AngleFormulas.html>

[R8] http://en.wikipedia.org/wiki/Partition_%28number_theory%29

[R12] Skiena, S. ‘Permutations.’ 1.1 in *Implementing Discrete Mathematics Combinatorics and Graph Theory with Mathematica*. Reading, MA: Addison-Wesley, pp. 3-16, 1990.

[R13] Knuth, D. E. *The Art of Computer Programming, Vol. 4: Combinatorial Algorithms*, 1st ed. Reading, MA: Addison-Wesley, 2011.

[R14] 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

[R15] D. L. Kreher, D. R. Stinson ‘*Combinatorial Algorithms*’ CRC Press, 1999

[R16] Graham, R. L.; Knuth, D. E.; and Patashnik, O. *Concrete Mathematics: A Foundation for Computer Science*, 2nd ed. Reading, MA: Addison-Wesley, 1994.

[R17] http://en.wikipedia.org/wiki/Permutation#Product_and_inverse

[R18] http://en.wikipedia.org/wiki/Lehmer_code

[R19] <http://mathworld.wolfram.com/LabeledTree.html>

[R4] Nijenhuis, A. and Wilf, H.S. (1978). *Combinatorial Algorithms*. Academic Press.

[R5] Knuth, D. (2011). *The Art of Computer Programming, Vol 4* Addison Wesley

[R6] <http://statweb.stanford.edu/~susan/courses/s208/node12.html>

[R7] http://groupprops.subwiki.org/wiki/Structure_theorem_for_finitely_generated_abelian_groups

[R20] Holt, D., Eick, B., O’Brien, E. “*Handbook of computational group theory*”

[R368] <http://primes.utm.edu/glossary/xpage/BertrandsPostulate.html>

[R369] http://en.wikipedia.org/wiki/Prime_number

[R370] <http://primes.utm.edu/notes/gaps.html>

- [R371] http://en.wikipedia.org/wiki/Bertrand's_postulate
- [R372] http://en.wikipedia.org/wiki/Cycle_detection.
- [R373] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 229-231
- [R374] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 236-238
- [R375] <http://modular.math.washington.edu/edu/2007/spring/ent/ent-html/node81.html>
- [R376] <http://www.cs.toronto.edu/~yuvalf/Factorization.pdf>
- [R377] <http://stackoverflow.com/questions/1010381/python-factorization>
- [R378] <http://www.mayer.dial.pipex.com/maths/formulae.htm>
- [R379] http://en.wikipedia.org/wiki/Square-free_integer#Squarefree_core
- [R380] <http://mathworld.wolfram.com/PartitionFunctionP.html>
- [R381] Richard Crandall & Carl Pomerance (2005), "Prime Numbers: A Computational Perspective", Springer, 2nd edition, 135-138
- [R382] A list of thresholds and the bases they require are here: http://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Deterministic_variants_of_the_test
- [R383] 23. Stein "Elementary Number Theory" (2011), page 44
- [R384] 16. Hackman "Elementary Number Theory" (2009), Chapter C
- [R385] 16. Hackman "Elementary Number Theory" (2009), page 76
- [R386] https://en.wikipedia.org/wiki/Legendre_symbol
- [R387] http://en.wikipedia.org/wiki/Continued_fraction
- [R388] http://en.wikipedia.org/wiki/Periodic_continued_fraction
- [R389] K. Rosen. Elementary Number theory and its applications. Addison-Wesley, 3 Sub edition, pages 379-381, January 1992.
- [R390] http://en.wikipedia.org/wiki/M%C3%B6bius_function
- [R391] Thomas Koshy "Elementary Number Theory with Applications"
- [R392] http://en.wikipedia.org/wiki/Egyptian_fraction
- [R393] https://en.wikipedia.org/wiki/Greedy_algorithm_for_Egyptian_fractions
- [R394] <http://www.ics.uci.edu/~eppstein/numth/egypt/conflict.html>
- [R395] http://ami.ektf.hu/uploads/papers/finalpdf/AMI_42_from129to134.pdf
- [R21] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R22] http://en.wikipedia.org/wiki/Summation#Capital-sigma_notation
- [R23] http://en.wikipedia.org/wiki/Empty_sum
- [R24] 13. Petkovšek, H. S. Wilf, D. Zeilberger, A = B, 1996, Ch. 4.
- [R25] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R26] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doid=322248.322255>
- [R27] http://en.wikipedia.org/wiki/Multiplication#Capital_Pi_notation
- [R28] http://en.wikipedia.org/wiki/Empty_product

- [R29] Michael Karr, "Summation in Finite Terms", Journal of the ACM, Volume 28 Issue 2, April 1981, Pages 305-350 <http://dl.acm.org/citation.cfm?doi=322248.322255>
- [R30] Marko Petkovšek, Herbert S. Wilf, Doron Zeilberger, A = B, AK Peters, Ltd., Wellesley, MA, USA, 1997, pp. 73-100
- [R111] http://en.wikipedia.org/wiki/Complex_conjugation
- [R112] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R113] <http://dlmf.nist.gov/4.14>
- [R114] <http://functions.wolfram.com/ElementaryFunctions/Sin>
- [R115] <http://mathworld.wolfram.com/TrigonometryAngles.html>
- [R116] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R117] <http://dlmf.nist.gov/4.14>
- [R118] <http://functions.wolfram.com/ElementaryFunctions/Cos>
- [R119] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R120] <http://dlmf.nist.gov/4.14>
- [R121] <http://functions.wolfram.com/ElementaryFunctions/Tan>
- [R122] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R123] <http://dlmf.nist.gov/4.14>
- [R124] <http://functions.wolfram.com/ElementaryFunctions/Cot>
- [R125] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R126] <http://dlmf.nist.gov/4.14>
- [R127] <http://functions.wolfram.com/ElementaryFunctions/Sec>
- [R128] http://en.wikipedia.org/wiki/Trigonometric_functions
- [R129] <http://dlmf.nist.gov/4.14>
- [R130] <http://functions.wolfram.com/ElementaryFunctions/Csc>
- [R131] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R132] <http://dlmf.nist.gov/4.23>
- [R133] <http://functions.wolfram.com/ElementaryFunctions/ArcSin>
- [R134] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R135] <http://dlmf.nist.gov/4.23>
- [R136] <http://functions.wolfram.com/ElementaryFunctions/ArcCos>
- [R137] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R138] <http://dlmf.nist.gov/4.23>
- [R139] <http://functions.wolfram.com/ElementaryFunctions/ArcSec>
- [R140] <http://reference.wolfram.com/language/ref/ArcSec.html>
- [R141] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R142] <http://dlmf.nist.gov/4.23>
- [R143] <http://functions.wolfram.com/ElementaryFunctions/ArcTan>
- [R144] <http://dlmf.nist.gov/4.23>
- [R145] <http://functions.wolfram.com/ElementaryFunctions/ArcCot>

- [R146] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R147] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R148] <http://dlmf.nist.gov/4.23>
- [R149] <http://functions.wolfram.com/ElementaryFunctions/ArcSec>
- [R150] <http://reference.wolfram.com/language/ref/ArcSec.html>
- [R151] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R152] <http://dlmf.nist.gov/4.23>
- [R153] <http://functions.wolfram.com/ElementaryFunctions/ArcCsc>
- [R154] http://en.wikipedia.org/wiki/Inverse_trigonometric_functions
- [R155] <http://en.wikipedia.org/wiki/Atan2>
- [R156] <http://functions.wolfram.com/ElementaryFunctions/ArcTan2>
- [R157] “Concrete mathematics” by Graham, pp. 87
- [R158] <http://mathworld.wolfram.com/CeilingFunction.html>
- [R159] “Concrete mathematics” by Graham, pp. 87
- [R160] <http://mathworld.wolfram.com/FloorFunction.html>
- [R161] http://en.wikipedia.org/wiki/Lambert_W_function
- [R162] http://en.wikipedia.org/wiki/Directed_complete_partial_order
- [R163] http://en.wikipedia.org/wiki/Lattice_%28order%29
- [R164] http://en.wikipedia.org/wiki/Square_root
- [R165] http://en.wikipedia.org/wiki/Principal_value
- [R80] http://en.wikipedia.org/wiki/Bell_number
- [R81] <http://mathworld.wolfram.com/BellNumber.html>
- [R82] <http://mathworld.wolfram.com/BellPolynomial.html>
- [R83] http://en.wikipedia.org/wiki/Bernoulli_number
- [R84] http://en.wikipedia.org/wiki/Bernoulli_polynomial
- [R85] <http://mathworld.wolfram.com/BernoulliNumber.html>
- [R86] <http://mathworld.wolfram.com/BernoulliPolynomial.html>
- [R87] http://en.wikipedia.org/wiki/Catalan_number
- [R88] <http://mathworld.wolfram.com/CatalanNumber.html>
- [R89] <http://functions.wolfram.com/GammaBetaErf/CatalanNumber/>
- [R90] <http://geometer.org/mathcircles/catalan.pdf>
- [R91] http://en.wikipedia.org/wiki/Euler_numbers
- [R92] <http://mathworld.wolfram.com/EulerNumber.html>
- [R93] http://en.wikipedia.org/wiki/Alternating_permutation
- [R94] <http://mathworld.wolfram.com/AlternatingPermutation.html>
- [R95] <http://en.wikipedia.org/wiki/Subfactorial>
- [R96] <http://mathworld.wolfram.com/Subfactorial.html>
- [R97] https://en.wikipedia.org/wiki/Double_factorial

- [R98] http://en.wikipedia.org/wiki/Fibonacci_number
- [R99] <http://mathworld.wolfram.com/FibonacciNumber.html>
- [R100] http://en.wikipedia.org/wiki/Harmonic_number
- [R101] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber/>
- [R102] <http://functions.wolfram.com/GammaBetaErf/HarmonicNumber2/>
- [R103] http://en.wikipedia.org/wiki/Lucas_number
- [R104] <http://mathworld.wolfram.com/LucasNumber.html>
- [R105] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_first_kind
- [R106] http://en.wikipedia.org/wiki/Stirling_numbers_of_the_second_kind
- [R107] <http://en.wikipedia.org/wiki/Combination>
- [R108] <http://tinyurl.com/cep849r>
- [R109] <http://en.wikipedia.org/wiki/Permutation>
- [R110] <http://undergraduate.csse.uwa.edu.au/units/CITS7209/partition.pdf>
- [R166] <http://mathworld.wolfram.com/DeltaFunction.html>
- [R167] https://en.wikipedia.org/wiki/Heaviside_step_function
- [R168] http://en.wikipedia.org/wiki/Gamma_function
- [R169] <http://dlmf.nist.gov/5>
- [R170] <http://mathworld.wolfram.com/GammaFunction.html>
- [R171] <http://functions.wolfram.com/GammaBetaErf/Gamma/>
- [R172] http://en.wikipedia.org/wiki/Gamma_function
- [R173] <http://dlmf.nist.gov/5>
- [R174] <http://mathworld.wolfram.com/LogGammaFunction.html>
- [R175] <http://functions.wolfram.com/GammaBetaErf/LogGamma/>
- [R176] http://en.wikipedia.org/wiki/Polygamma_function
- [R177] <http://mathworld.wolfram.com/PolygammaFunction.html>
- [R178] <http://functions.wolfram.com/GammaBetaErf/PolyGamma/>
- [R179] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R180] http://en.wikipedia.org/wiki/Digamma_function
- [R181] <http://mathworld.wolfram.com/DigammaFunction.html>
- [R182] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R183] http://en.wikipedia.org/wiki/Trigamma_function
- [R184] <http://mathworld.wolfram.com/TrigammaFunction.html>
- [R185] <http://functions.wolfram.com/GammaBetaErf/PolyGamma2/>
- [R186] http://en.wikipedia.org/wiki/Incomplete_gamma_function#Upper_incomplete_Gamma_function
- [R187] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R188] <http://dlmf.nist.gov/8>
- [R189] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>

- [R190] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R191] http://en.wikipedia.org/wiki/Exponential_integral#Relation_with_other_functions
- [R192] http://en.wikipedia.org/wiki/Incomplete_gamma_function#Lower_incomplete_Gamma_function
- [R193] Abramowitz, Milton; Stegun, Irene A., eds. (1965), Chapter 6, Section 5, Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R194] <http://dlmf.nist.gov/8>
- [R195] <http://functions.wolfram.com/GammaBetaErf/Gamma2/>
- [R196] <http://functions.wolfram.com/GammaBetaErf/Gamma3/>
- [R197] http://en.wikipedia.org/wiki/Beta_function
- [R198] <http://mathworld.wolfram.com/BetaFunction.html>
- [R199] <http://dlmf.nist.gov/5.12>
- [R200] http://en.wikipedia.org/wiki/Error_function
- [R201] <http://dlmf.nist.gov/7>
- [R202] <http://mathworld.wolfram.com/Erf.html>
- [R203] <http://functions.wolfram.com/GammaBetaErf/Erf>
- [R204] http://en.wikipedia.org/wiki/Error_function
- [R205] <http://dlmf.nist.gov/7>
- [R206] <http://mathworld.wolfram.com/Erfc.html>
- [R207] <http://functions.wolfram.com/GammaBetaErf/Erfc>
- [R208] http://en.wikipedia.org/wiki/Error_function
- [R209] <http://mathworld.wolfram.com/Erfi.html>
- [R210] <http://functions.wolfram.com/GammaBetaErf/Erfi>
- [R211] <http://functions.wolfram.com/GammaBetaErf/Erf2/>
- [R212] http://en.wikipedia.org/wiki/Error_function#Inverse_functions
- [R213] <http://functions.wolfram.com/GammaBetaErf/InverseErf/>
- [R214] http://en.wikipedia.org/wiki/Error_function#Inverse_functions
- [R215] <http://functions.wolfram.com/GammaBetaErf/InverseErfc/>
- [R216] <http://functions.wolfram.com/GammaBetaErf/InverseErf2/>
- [R217] http://en.wikipedia.org/wiki/Fresnel_integral
- [R218] <http://dlmf.nist.gov/7>
- [R219] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R220] <http://functions.wolfram.com/GammaBetaErf/FresnelS>
- [R221] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley
- [R222] http://en.wikipedia.org/wiki/Fresnel_integral
- [R223] <http://dlmf.nist.gov/7>
- [R224] <http://mathworld.wolfram.com/FresnelIntegrals.html>
- [R225] <http://functions.wolfram.com/GammaBetaErf/FresnelC>
- [R226] The converging factors for the fresnel integrals by John W. Wrench Jr. and Vicki Alley

- [R227] <http://dlmf.nist.gov/6.6>
- [R228] http://en.wikipedia.org/wiki/Exponential_integral
- [R229] Abramowitz & Stegun, section 5: http://people.math.sfu.ca/~cbm/aands/page_228.htm
- [R230] <http://dlmf.nist.gov/8.19>
- [R231] <http://functions.wolfram.com/GammaBetaErf/ExpIntegralE/>
- [R232] http://en.wikipedia.org/wiki/Exponential_integral
- [R233] http://en.wikipedia.org/wiki/Logarithmic_integral
- [R234] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R235] <http://dlmf.nist.gov/6>
- [R236] <http://mathworld.wolfram.com/SoldnersConstant.html>
- [R237] http://en.wikipedia.org/wiki/Logarithmic_integral
- [R238] <http://mathworld.wolfram.com/LogarithmicIntegral.html>
- [R239] <http://dlmf.nist.gov/6>
- [R240] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R241] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R242] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R243] http://en.wikipedia.org/wiki/Trigonometric_integral
- [R244] Abramowitz, Milton; Stegun, Irene A., eds. (1965), "Chapter 9", Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables
- [R245] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R246] http://en.wikipedia.org/wiki/Bessel_function
- [R247] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselJ/>
- [R248] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselY/>
- [R249] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselI/>
- [R250] <http://functions.wolfram.com/Bessel-TypeFunctions/BesselK/>
- [R251] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH1/>
- [R252] <http://functions.wolfram.com/Bessel-TypeFunctions/HankelH2/>
- [R253] http://en.wikipedia.org/wiki/Airy_function
- [R254] <http://dlmf.nist.gov/9>
- [R255] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R256] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R257] http://en.wikipedia.org/wiki/Airy_function
- [R258] <http://dlmf.nist.gov/9>
- [R259] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R260] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R261] http://en.wikipedia.org/wiki/Airy_function
- [R262] <http://dlmf.nist.gov/9>
- [R263] http://www.encyclopediaofmath.org/index.php/Airy_functions

- [R264] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R265] http://en.wikipedia.org/wiki/Airy_function
- [R266] <http://dlmf.nist.gov/9>
- [R267] http://www.encyclopediaofmath.org/index.php/Airy_functions
- [R268] <http://mathworld.wolfram.com/AiryFunctions.html>
- [R269] <http://en.wikipedia.org/wiki/B-spline>
- [R270] <http://dlmf.nist.gov/25.11>
- [R271] http://en.wikipedia.org/wiki/Hurwitz_zeta_function
- [R272] http://en.wikipedia.org/wiki/Dirichlet_eta_function
- [R273] <http://mathworld.wolfram.com/DirichletEtaFunction.html>
- [R274] <https://en.wikipedia.org/wiki/Polylogarithm>
- [R275] <http://mathworld.wolfram.com/Polylogarithm.html>
- [R276] Bateman, H.; Erdelyi, A. (1953), Higher Transcendental Functions, Vol. I, New York: McGraw-Hill. Section 1.11.
- [R277] <http://dlmf.nist.gov/25.14>
- [R278] http://en.wikipedia.org/wiki/Lerch_transcendent
- [R279] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R280] http://en.wikipedia.org/wiki/Generalized_hypergeometric_function
- [R281] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1
- [R282] http://en.wikipedia.org/wiki/Meijer_G-function
- [R283] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R284] <http://functions.wolfram.com/EllipticIntegrals/EllipticK>
- [R285] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R286] <http://functions.wolfram.com/EllipticIntegrals/EllipticF>
- [R287] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R288] <http://functions.wolfram.com/EllipticIntegrals/EllipticE2>
- [R289] <http://functions.wolfram.com/EllipticIntegrals/EllipticE>
- [R290] http://en.wikipedia.org/wiki/Elliptic_integrals
- [R291] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi3>
- [R292] <http://functions.wolfram.com/EllipticIntegrals/EllipticPi>
- [R293] http://en.wikipedia.org/wiki/Jacobi_polynomials
- [R294] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R295] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R296] http://en.wikipedia.org/wiki/Jacobi_polynomials
- [R297] <http://mathworld.wolfram.com/JacobiPolynomial.html>
- [R298] <http://functions.wolfram.com/Polynomials/JacobiP/>
- [R299] http://en.wikipedia.org/wiki/Gegenbauer_polynomials
- [R300] <http://mathworld.wolfram.com/GegenbauerPolynomial.html>

- [R301] <http://functions.wolfram.com/Polynomials/GegenbauerC3/>
- [R302] http://en.wikipedia.org/wiki/Chebyshev_polynomial
- [R303] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R304] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R305] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R306] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R307] http://en.wikipedia.org/wiki/Chebyshev_polynomial
- [R308] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheFirstKind.html>
- [R309] <http://mathworld.wolfram.com/ChebyshevPolynomialoftheSecondKind.html>
- [R310] <http://functions.wolfram.com/Polynomials/ChebyshevT/>
- [R311] <http://functions.wolfram.com/Polynomials/ChebyshevU/>
- [R312] http://en.wikipedia.org/wiki/Legendre_polynomial
- [R313] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R314] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R315] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R316] http://en.wikipedia.org/wiki/Associated_Legendre_polynomials
- [R317] <http://mathworld.wolfram.com/LegendrePolynomial.html>
- [R318] <http://functions.wolfram.com/Polynomials/LegendreP/>
- [R319] <http://functions.wolfram.com/Polynomials/LegendreP2/>
- [R320] http://en.wikipedia.org/wiki/Hermite_polynomial
- [R321] <http://mathworld.wolfram.com/HermitePolynomial.html>
- [R322] <http://functions.wolfram.com/Polynomials/HermiteH/>
- [R323] http://en.wikipedia.org/wiki/Laguerre_polynomial
- [R324] <http://mathworld.wolfram.com/LaguerrePolynomial.html>
- [R325] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R326] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R327] http://en.wikipedia.org/wiki/Laguerre_polynomial#Assoc_laguerre_polynomials
- [R328] <http://mathworld.wolfram.com/AssociatedLaguerrePolynomial.html>
- [R329] <http://functions.wolfram.com/Polynomials/LaguerreL/>
- [R330] <http://functions.wolfram.com/Polynomials/LaguerreL3/>
- [R331] http://en.wikipedia.org/wiki/Spherical_harmonics
- [R332] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R333] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R334] <http://dlmf.nist.gov/14.30>
- [R335] http://en.wikipedia.org/wiki/Spherical_harmonics
- [R336] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R337] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R338] http://en.wikipedia.org/wiki/Spherical_harmonics

- [R339] <http://mathworld.wolfram.com/SphericalHarmonic.html>
- [R340] <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/>
- [R341] http://en.wikipedia.org/wiki/Kronecker_delta
- [WikiPappus] "Pappus's Hexagon Theorem" Wikipedia, the Free Encyclopedia. Web. 26 Apr. 2013. <http://en.wikipedia.org/wiki/Pappus's_hexagon_theorem>
- [R342] <http://paulbourke.net/geometry/polygonmesh/#insidepoly>
- [BlogPost] <http://nessgrh.wordpress.com/2011/07/07/tricky-branch-cuts/>
- [Bro05] M. Bronstein, Symbolic Integration I: Transcendental Functions, Second Edition, Springer-Verlag, 2005, pp. 35-70
- [R343] Manuel Bronstein's "Poor Man's Integrator", <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint/index.html>
- [R344] K. Geddes, L. Stefanus, On the Risch-Norman Integration Method and its Implementation in Maple, Proceedings of ISSAC'89, ACM Press, 212-217.
- [R345] J. H. Davenport, On the Parallel Risch Algorithm (I), Proceedings of EUROCAM'82, LNCS 144, Springer, 144-157.
- [R346] J. H. Davenport, On the Parallel Risch Algorithm (III): Use of Tangents, SIGSAM Bulletin 16 (1982), 3-6.
- [R347] J. H. Davenport, B. M. Trager, On the Parallel Risch Algorithm (II), ACM Transactions on Mathematical Software 11 (1985), 356-362.
- [R348] http://en.wikipedia.org/wiki/Gaussian_quadrature
- [R349] http://people.sc.fsu.edu/~jburkardt/cpp_src/legendre_rule/legendre_rule.html
- [R350] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature
- [R351] http://people.sc.fsu.edu/~jburkardt/cpp_src/laguerre_rule/laguerre_rule.html
- [R352] http://en.wikipedia.org/wiki/Gauss-Hermite_Quadrature
- [R353] http://people.sc.fsu.edu/~jburkardt/cpp_src/hermite_rule/hermite_rule.html
- [R354] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_hermite_rule/gen_hermite_rule.html
- [R355] http://en.wikipedia.org/wiki/Gauss%E2%80%93Laguerre_quadrature
- [R356] http://people.sc.fsu.edu/~jburkardt/cpp_src/gen_laguerre_rule/gen_laguerre_rule.html
- [R357] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature
- [R358] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev1_rule/chebyshev1_rule.html
- [R359] http://en.wikipedia.org/wiki/Chebyshev%E2%80%93Gauss_quadrature
- [R360] http://people.sc.fsu.edu/~jburkardt/cpp_src/chebyshev2_rule/chebyshev2_rule.html
- [R361] http://en.wikipedia.org/wiki/Gauss%E2%80%93Jacobi_quadrature
- [R362] http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_rule/jacobi_rule.html
- [R363] http://people.sc.fsu.edu/~jburkardt/cpp_src/gegenbauer_rule/gegenbauer_rule.html
- [R364] en.wikipedia.org/wiki/Quine-McCluskey_algorithm
- [R365] en.wikipedia.org/wiki/Quine-McCluskey_algorithm
- [R366] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse

- [R367] https://en.wikipedia.org/wiki/Moore-Penrose_pseudoinverse#Obtaining_all_solutions_of_a_linear_system
- [Wester1999] Michael J. Wester, A Critique of the Mathematical Abilities of CA Systems, 1999, <http://www.math.unm.edu/~wester/cas/book/Wester.pdf>
- [R422] [*ManWright94*] (page 1273)
- [R423] [*Koepf98*] (page 1273)
- [R424] [*Abramov71*] (page 1273)
- [R425] [*Man93*] (page 1273)
- [R426] [*ManWright94*] (page 1273)
- [R427] [*Koepf98*] (page 1273)
- [R428] [*Abramov71*] (page 1273)
- [R429] [*Man93*] (page 1273)
- [R430] [*Buchberger01*] (page 1272)
- [R431] [*Cox97*] (page 1272)
- [R432] David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230.
- [R433] [*ManWright94*] (page 1273)
- [R434] [*Koepf98*] (page 1273)
- [R435] [*Abramov71*] (page 1273)
- [R436] [*Man93*] (page 1273)
- [R437] [*ManWright94*] (page 1273)
- [R438] [*Koepf98*] (page 1273)
- [R439] [*Abramov71*] (page 1273)
- [R440] [*Man93*] (page 1273)
- [R441] Alkiviadis G. Akritas and Adam W. Strzebonski: A Comparative Study of Two Real Root Isolation Methods. Nonlinear Analysis: Modelling and Control, Vol. 10, No. 4, 297-304, 2005.
- [R442] Alkiviadis G. Akritas, Adam W. Strzebonski and Panagiotis S. Vigklas: Improving the Performance of the Continued Fractions Method Using new Bounds of Positive Roots. Nonlinear Analysis: Modelling and Control, Vol. 13, No. 3, 265-279, 2008.
- [R443] J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering.
- [R444] David A. Cox, John B. Little, Donal O'Shea. Ideals, Varieties and Algorithms, 3rd edition, p. 230.
- [R445] http://en.wikipedia.org/wiki/Horner_scheme
- [R446] http://en.wikipedia.org/wiki/Cubic_function#Trigonometric_.28and_hyperbolic.29_method
- [R447] [*Bronstein93*] (page 1272)
- [R448] [*ManWright94*] (page 1273)
- [R449] [*Koepf98*] (page 1273)
- [R450] [*Abramov71*] (page 1273)

- [R451] [*Man93*] (page 1273)
- [R452] [*ManWright94*] (page 1273)
- [R453] [*Koepf98*] (page 1273)
- [R454] [*Abramov71*] (page 1273)
- [R455] [*Man93*] (page 1273)
- [R396] [*Kozen89*] (page 1271)
- [R397] [*Monagan93*] (page 1272)
- [R398] [*Gathen99*] (page 1272)
- [R399] [*Gathen99*] (page 1272)
- [R400] [*Gathen99*] (page 1272)
- [R401] [*Gathen92*] (page 1272)
- [R402] [*Geddes92*] (page 1272)
- [R403] [*Gathen99*] (page 1272)
- [R404] ‘An introduction to the Theory of Numbers’ 5th Edition by Ivan Niven, Zuckerman and Montgomery.
- [R405] [*Liao95*] (page 1271)
- [R406] [*Gathen99*] (page 1272)
- [R407] [*Gathen99*] (page 1272)
- [R408] [*Weisstein09*] (page 1272)
- [R409] [*Gathen99*] (page 1272)
- [R410] [*Wang78*] (page 1272)
- [R411] [*Geddes92*] (page 1272)
- [R412] [*Gathen99*] (page 1272)
- [R413] J.C. Faugere, P. Gianni, D. Lazard, T. Mora (1994). Efficient Computation of Zero-dimensional Groebner Bases by Change of Ordering.
- [R414] [*Monagan00*] (page 1273)
- [R415] [*Monagan00*] (page 1273)
- [R416] [*Monagan00*] (page 1273)
- [R417] [*Brown71*] (page 1273)
- [R418] [*Hoeij04*] (page 1273)
- [R419] [*Monagan00*] (page 1273)
- [R420] [*Brown71*] (page 1273)
- [R421] A. Bostan, P. Flajolet, B. Salvy and E. Schost “Fast Computation with Two Algebraic Numbers”, (2002) Research Report 4579, Institut National de Recherche en Informatique et en Automatique.
- [Kozen89] D. Kozen, S. Landau, Polynomial decomposition algorithms, Journal of Symbolic Computation 7 (1989), pp. 445-456
- [Liao95] Hsin-Chao Liao, R. Fateman, Evaluation of the heuristic polynomial GCD, International Symposium on Symbolic and Algebraic Computation (ISSAC), ACM Press, Montreal, Quebec, Canada, 1995, pp. 240-247

- [Gathen99] J. von zur Gathen, J. Gerhard, *Modern Computer Algebra*, First Edition, Cambridge University Press, 1999
- [Weisstein09] Eric W. Weisstein, *Cyclotomic Polynomial*, From MathWorld - A Wolfram Web Resource, <http://mathworld.wolfram.com/CyclotomicPolynomial.html>
- [Wang78] P. S. Wang, *An Improved Multivariate Polynomial Factoring Algorithm*, *Math. of Computation* 32, 1978, pp. 1215-1231
- [Geddes92] K. Geddes, S. R. Czapor, G. Labahn, *Algorithms for Computer Algebra*, Springer, 1992
- [Monagan93] Michael Monagan, *In-place Arithmetic for Polynomials over Z_n* , *Proceedings of DISCO '92*, Springer-Verlag LNCS, 721, 1993, pp. 22-34
- [Kaltofen98] E. Kaltofen, V. Shoup, *Subquadratic-time Factoring of Polynomials over Finite Fields*, *Mathematics of Computation*, Volume 67, Issue 223, 1998, pp. 1179-1197
- [Shoup95] V. Shoup, *A New Polynomial Factorization Algorithm and its Implementation*, *Journal of Symbolic Computation*, Volume 20, Issue 4, 1995, pp. 363-397
- [Gathen92] J. von zur Gathen, V. Shoup, *Computing Frobenius Maps and Factoring Polynomials*, *ACM Symposium on Theory of Computing*, 1992, pp. 187-224
- [Shoup91] V. Shoup, *A Fast Deterministic Algorithm for Factoring Polynomials over Finite Fields of Small Characteristic*, In *Proceedings of International Symposium on Symbolic and Algebraic Computation*, 1991, pp. 14-21
- [Cox97] D. Cox, J. Little, D. O'Shea, *Ideals, Varieties and Algorithms*, Springer, Second Edition, 1997
- [Ajwa95] I.A. Ajwa, Z. Liu, P.S. Wang, *Groebner Bases Algorithm*, <https://citeseer.ist.psu.edu/myciteseer/login>, 1995
- [Bose03] N.K. Bose, B. Buchberger, J.P. Guiver, *Multidimensional Systems Theory and Applications*, Springer, 2003
- [Giovini91] A. Giovini, T. Mora, "One sugar cube, please" or Selection strategies in Buchberger algorithm, *ISSAC '91*, ACM
- [Bronstein93] M. Bronstein, B. Salvy, *Full partial fraction decomposition of rational functions*, *Proceedings ISSAC '93*, ACM Press, Kiev, Ukraine, 1993, pp. 157-160
- [Buchberger01] B. Buchberger, *Groebner Bases: A Short Introduction for Systems Theorists*, In: R. Moreno-Diaz, B. Buchberger, J. L. Freire, *Proceedings of EUROCAST'01*, February, 2001
- [Davenport88] J.H. Davenport, Y. Siret, E. Tournier, *Computer Algebra Systems and Algorithms for Algebraic Computation*, Academic Press, London, 1988, pp. 124-128
- [Greuel2008] G.-M. Greuel, Gerhard Pfister, *A Singular Introduction to Commutative Algebra*, Springer, 2008
- [Atiyah69] M.F. Atiyah, I.G. MacDonald, *Introduction to Commutative Algebra*, Addison-Wesley, 1969
- [Collins67] G.E. Collins, *Subresultants and Reduced Polynomial Remainder Sequences*. *J. ACM* 14 (1967) 128-142
- [BrownTraub71] W.S. Brown, J.F. Traub, *On Euclid's Algorithm and the Theory of Subresultants*. *J. ACM* 18 (1971) 505-514
- [Brown78] W.S. Brown, *The Subresultant PRS Algorithm*. *ACM Transaction of Mathematical Software* 4 (1978) 237-249

- [Monagan00] M. Monagan and A. Wittkopf, On the Design and Implementation of Brown's Algorithm over the Integers and Number Fields, Proceedings of ISSAC 2000, pp. 225-233, ACM, 2000.
- [Brown71] W.S. Brown, On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors, J. ACM 18, 4, pp. 478-504, 1971.
- [Hoeij04] M. van Hoeij and M. Monagan, Algorithms for polynomial GCD computation over algebraic function fields, Proceedings of ISSAC 2004, pp. 297-304, ACM, 2004.
- [Wang81] P.S. Wang, A p-adic algorithm for univariate partial fractions, Proceedings of SYM-SAC 1981, pp. 212-217, ACM, 1981.
- [Hoeij02] M. van Hoeij and M. Monagan, A modular GCD algorithm over number fields presented with multiple extensions, Proceedings of ISSAC 2002, pp. 109-116, ACM, 2002
- [ManWright94] Yiu-Kwong Man and Francis J. Wright, "Fast Polynomial Dispersion Computation and its Application to Indefinite Summation", Proceedings of the International Symposium on Symbolic and Algebraic Computation, 1994, Pages 175-180 <http://dl.acm.org/citation.cfm?doid=190347.190413>
- [Koepf98] Wolfram Koepf, "Hypergeometric Summation: An Algorithmic Approach to Summation and Special Function Identities", Advanced lectures in mathematics, Vieweg, 1998
- [Abramov71] S. A. Abramov, "On the Summation of Rational Functions", USSR Computational Mathematics and Mathematical Physics, Volume 11, Issue 4, 1971, Pages 324-330
- [Man93] Yiu-Kwong Man, "On Computing Closed Forms for Indefinite Summations", Journal of Symbolic Computation, Volume 16, Issue 4, 1993, Pages 355-376 <http://www.sciencedirect.com/science/article/pii/S0747717183710539>
- [R456] <http://www.w3.org/TR/MathML2/>
- [R457] http://en.wikipedia.org/wiki/Big_O_notation
- [R458] http://en.wikipedia.org/wiki/Residue_%28complex_analysis%29
- [R459] http://en.wikipedia.org/wiki/Residue_theorem
- [R460] Gruntz Thesis
- [R461] http://en.wikipedia.org/wiki/Disjoint_sets
- [R462] http://en.wikipedia.org/wiki/Power_set
- [R463] http://en.wikipedia.org/wiki/Interval_%28mathematics%29
- [R464] http://en.wikipedia.org/wiki/Finite_set
- [R465] http://en.wikipedia.org/wiki/Union_%28set_theory%29
- [R466] http://en.wikipedia.org/wiki/Intersection_%28set_theory%29
- [R467] http://en.wikipedia.org/wiki/Cartesian_product
- [R468] <http://mathworld.wolfram.com/ComplementSet.html>
- [R469] http://en.wikipedia.org/wiki/Empty_set
- [R470] http://en.wikipedia.org/wiki/Universal_set
- [Roach1996] Kelly B. Roach. Hypergeometric Function Representations. In: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, pages 301-308, New York, 1996. ACM.
- [Roach1997] Kelly B. Roach. Meijer G Function Representations. In: Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation, pages 205-211, New York, 1997. ACM.

- [Luke1969] Luke, Y. L. (1969), The Special Functions and Their Approximations, Volume 1.
- [Prudnikov1990] A. P. Prudnikov, Yu. A. Brychkov and O. I. Marichev (1990). Integrals and Series: More Special Functions, Vol. 3, Gordon and Breach Science Publisher.
- [R524] http://en.wikipedia.org/wiki/Arcsine_distribution
- [R525] http://en.wikipedia.org/wiki/Benini_distribution
- [R526] <http://reference.wolfram.com/legacy/v8/ref/BeniniDistribution.html>
- [R527] http://en.wikipedia.org/wiki/Beta_distribution
- [R528] <http://mathworld.wolfram.com/BetaDistribution.html>
- [R529] http://en.wikipedia.org/wiki/Beta_prime_distribution
- [R530] <http://mathworld.wolfram.com/BetaPrimeDistribution.html>
- [R531] http://en.wikipedia.org/wiki/Cauchy_distribution
- [R532] <http://mathworld.wolfram.com/CauchyDistribution.html>
- [R533] http://en.wikipedia.org/wiki/Chi_distribution
- [R534] <http://mathworld.wolfram.com/ChiDistribution.html>
- [R535] http://en.wikipedia.org/wiki/Noncentral_chi_distribution
- [R536] http://en.wikipedia.org/wiki/Chi_squared_distribution
- [R537] <http://mathworld.wolfram.com/Chi-SquaredDistribution.html>
- [R538] http://en.wikipedia.org/wiki/Dagum_distribution
- [R539] http://en.wikipedia.org/wiki/Erlang_distribution
- [R540] <http://mathworld.wolfram.com/ErlangDistribution.html>
- [R541] http://en.wikipedia.org/wiki/Exponential_distribution
- [R542] <http://mathworld.wolfram.com/ExponentialDistribution.html>
- [R543] <http://en.wikipedia.org/wiki/F-distribution>
- [R544] <http://mathworld.wolfram.com/F-Distribution.html>
- [R545] http://en.wikipedia.org/wiki/Fisher%27s_z-distribution
- [R546] <http://mathworld.wolfram.com/Fishersz-Distribution.html>
- [R547] http://en.wikipedia.org/wiki/Fr%C3%A9chet_distribution
- [R548] http://en.wikipedia.org/wiki/Gamma_distribution
- [R549] <http://mathworld.wolfram.com/GammaDistribution.html>
- [R550] http://en.wikipedia.org/wiki/Inverse-gamma_distribution
- [R551] http://en.wikipedia.org/wiki/Kumaraswamy_distribution
- [R552] http://en.wikipedia.org/wiki/Laplace_distribution
- [R553] <http://mathworld.wolfram.com/LaplaceDistribution.html>
- [R554] http://en.wikipedia.org/wiki/Logistic_distribution
- [R555] <http://mathworld.wolfram.com/LogisticDistribution.html>
- [R556] <http://en.wikipedia.org/wiki/Lognormal>
- [R557] <http://mathworld.wolfram.com/LogNormalDistribution.html>
- [R558] http://en.wikipedia.org/wiki/Maxwell_distribution

- [R559] <http://mathworld.wolfram.com/MaxwellDistribution.html>
- [R560] http://en.wikipedia.org/wiki/Nakagami_distribution
- [R561] http://en.wikipedia.org/wiki/Normal_distribution
- [R562] <http://mathworld.wolfram.com/NormalDistributionFunction.html>
- [R563] http://en.wikipedia.org/wiki/Pareto_distribution
- [R564] <http://mathworld.wolfram.com/ParetoDistribution.html>
- [R565] http://en.wikipedia.org/wiki/U-quadratic_distribution
- [R566] http://en.wikipedia.org/wiki/Raised_cosine_distribution
- [R567] http://en.wikipedia.org/wiki/Rayleigh_distribution
- [R568] <http://mathworld.wolfram.com/RayleighDistribution.html>
- [R569] http://en.wikipedia.org/wiki/Student_t-distribution
- [R570] <http://mathworld.wolfram.com/Studentst-Distribution.html>
- [R571] http://en.wikipedia.org/wiki/Triangular_distribution
- [R572] <http://mathworld.wolfram.com/TriangularDistribution.html>
- [R573] http://en.wikipedia.org/wiki/Uniform_distribution_%28continuous%29
- [R574] <http://mathworld.wolfram.com/UniformDistribution.html>
- [R575] http://en.wikipedia.org/wiki/Uniform_sum_distribution
- [R576] <http://mathworld.wolfram.com/UniformSumDistribution.html>
- [R577] http://en.wikipedia.org/wiki/Von_Mises_distribution
- [R578] <http://mathworld.wolfram.com/vonMisesDistribution.html>
- [R579] http://en.wikipedia.org/wiki/Weibull_distribution
- [R580] <http://mathworld.wolfram.com/WeibullDistribution.html>
- [R581] http://en.wikipedia.org/wiki/Wigner_semicircle_distribution
- [R582] <http://mathworld.wolfram.com/WignersSemicircleLaw.html>
- [R492] http://en.wikipedia.org/wiki/Exact_differential_equation
- [R493] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 73.
- [R494] http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [R495] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59.
- [R496] http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [R497] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59.
- [R498] http://en.wikipedia.org/wiki/Homogeneous_differential_equation
- [R499] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 59.
- [R500] http://en.wikipedia.org/wiki/Linear_differential_equation#First_order_equation
- [R501] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 92.
- [R502] http://en.wikipedia.org/wiki/Bernoulli_differential_equation
- [R503] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 95.
- [R504] Goldstein and Braun, "Advanced Methods for the Solution of Differential Equations", pp. 98.

- [R505] <http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville>
- [R506] http://en.wikipedia.org/wiki/Linear_differential_equation section: Nonhomogeneous equation with constant coefficients
- [R507] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 211.
- [R508] http://en.wikipedia.org/wiki/Method_of_undetermined_coefficients
- [R509] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 221.
- [R510] http://en.wikipedia.org/wiki/Variation_of_parameters
- [R511] <http://planetmath.org/VariationOfParameters>
- [R512] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 233.
- [R513] M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 52.
- [R514] Solving differential equations by Symmetry Groups, John Starrett, pp. 1-14.
- [R515] <http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx>
- [R516] George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184.
- [R517] Ernst Hairer, Syvert Paul Nørsett, Gerhard Wanner. Solving Ordinary Differential Equations I. Nonstiff Problems. Springer Series in Comput. Mathematics, Vol. 8, Springer-Verlag 1987, Second revised edition 1993, pp. 73-76.
- [R518] S. A. Abramov, M. Bronstein and M. Petkovšek, On polynomial solutions of linear operator equations, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 290-296.
- [R519] M. Petkovšek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R520] 13. Petkovšek, H. S. Wilf, D. Zeilberger, A = B, 1996.
- [R521] S. A. Abramov, Rational solutions of linear difference and q-difference equations with polynomial coefficients, in: T. Levelt, ed., Proc. ISSAC '95, ACM Press, New York, 1995, 285-289
- [R522] M. Petkovšek, Hypergeometric solutions of linear recurrences with polynomial coefficients, J. Symbolic Computation, 14 (1992), 243-264.
- [R523] 13. Petkovšek, H. S. Wilf, D. Zeilberger, A = B, 1996.
- [R471] Methods to solve $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, [online], Available: <http://www.alpertron.com.ar/METHODS.HTM>
- [R472] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, [online], Available: <http://www.jpr2718.org/ax2p.pdf>
- [R473] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Pages 16 - 17. [online], Available: <http://www.jpr2718.org/pell.pdf>
- [R474] 1. Nitaj, "L'algorithme de Cornacchia"
- [R475] Solving the diophantine equation $ax^2 + by^2 = m$ by Cornacchia's method, [online], Available: <http://www.numbertheory.org/php/cornacchia.html>
- [R476] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 15. <http://www.jpr2718.org/pell.pdf>
- [R477] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>
- [R478] Solving the equation $ax^2 + bxy + cy^2 + dx + ey + f = 0$, John P. Robertson, May 8, 2003, Page 7 - 11. <http://www.jpr2718.org/ax2p.pdf>

- [R479] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R480] Representing an Integer as a sum of three squares, [online], Available: http://www.proofwiki.org/wiki/Integer_as_Sum_of_Three_Squares
- [R481] Generating Integer Partitions, [online], Available: <http://jeromekelleher.net/partitions.php>
- [R482] Representing a number as a sum of three squares, [online], Available: <http://www.schorn.ch/howto.html>
- [R483] Representing a number as a sum of four squares, [online], Available: <http://www.schorn.ch/howto.html>
- [R484] Solving the generalized Pell equation $x^2 - Dy^2 = N$, John P. Robertson, July 31, 2004, Pages 4 - 8. <http://www.jpr2718.org/pell.pdf>
- [R485] Solving the generalized Pell equation $x^2 - D*y^2 = N$, John P. Robertson, July 31, 2004, Page 12. <http://www.jpr2718.org/pell.pdf>
- [R486] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R487] The algorithmic resolution of Diophantine equations, Nigel P. Smart, London Mathematical Society Student Texts 41, Cambridge University Press, Cambridge, 1998.
- [R488] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R489] Gaussian lattice Reduction [online]. Available: <http://home.ie.cuhk.edu.hk/~wkshum/wordpress/?p=404>
- [R490] Efficient Solution of Rational Conics, J. E. Cremona and D. Rusin, Mathematics of Computation, Volume 00, Number 0.
- [R491] Representing a number as a sum of four squares, [online], Available: <http://www.schorn.ch/howto.html>
- [R583] <http://docs.scipy.org/doc/numpy/reference/ufuncs.html>
- [AOCP] Algorithm 7.1.2.5M in Volume 4A, Combinatorial Algorithms, Part 1, of The Art of Computer Programming, by Donald Knuth.
- [Factorisatio] On a Problem of Oppenheim concerning "Factorisatio Numerorum" E. R. Canfield, Paul 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.
- [Yorgey] Generating Multiset Partitions, Brent Yorgey, The Monad.Reader, Issue 8, September 2007.
- [R584] TAOCP 4, section 7.2.1.5, problem 64
- [R585] <http://mathworld.wolfram.com/PermutationInvolution.html>
- [R586] 20. Beyer and S.M. Hedetniemi: constant time generation of rooted trees, SIAM J. Computing Vol. 9, No. 4, November 1980
- [R587] <http://stackoverflow.com/questions/1633833/oriented-forest-taocp-algorithm-in-python>
- [R588] <http://code.activestate.com/recipes/218332-generator-for-integer-partitions/>
- [R1] http://en.wikipedia.org/wiki/Euler%E2%80%93Lagrange_equation
- [R2] http://en.wikipedia.org/wiki/Mathematical_singularity

[R3] Generation of Finite Difference Formulas on Arbitrarily Spaced Grids, Bengt Fornberg; Mathematics of computation; 51; 184; (1988); 699-706; doi:10.1090/S0025-5718-1988-0935077-0

[WikiDel] <http://en.wikipedia.org/wiki/Del>

[R589] http://en.wikipedia.org/wiki/Dyadic_tensor

[R590] Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

[Dyadics] <http://en.wikipedia.org/wiki/Dyadics>

[DyadicProducts] http://en.wikipedia.org/wiki/Dyadic_product

[DelOperator] <http://en.wikipedia.org/wiki/Del>

Symbols

_TensorManager (class in diofant.tensor.tensor), 1106
 __contains__() (diofant.combinatorics.perm_groups.PermutationGroup method), 189
 __eq__() (diofant.combinatorics.perm_groups.PermutationGroup method), 189
 __getitem__() (diofant.matrices.dense.DenseMatrix method), 629
 __init__() (diofant.vector.coordsysrect.CoordSysCartesian method), 1209
 __init__() (diofant.vector.orienters.AxisOrienter method), 1220
 __init__() (diofant.vector.orienters.BodyOrienter method), 1221
 __init__() (diofant.vector.orienters.QuaternionOrienter method), 1223
 __init__() (diofant.vector.orienters.SpaceOrienter method), 1222
 __mul__() (diofant.combinatorics.perm_groups.PermutationGroup method), 190
 __mul__() (diofant.matrices.dense.DenseMatrix method), 630
 __new__() (diofant.combinatorics.perm_groups.PermutationGroup static method), 190
 af_parity() (in module diofant.combinatorics.permutations), 186
 base_ordering() (in module diofant.combinatorics.util), 237
 check_cycles_alt_sym() (in module diofant.combinatorics.util), 238
 cmp_perm_lists() (in module diofant.combinatorics.testutil), 243
 distribute_gens_by_base() (in module diofant.combinatorics.util), 238
 handle_Integral() (in module diofant.solvers.ode), 1045
 handle_precomputed_bsgs() (in module diofant.combinatorics.util), 239
 _linear_2eq_order1_type1() (in module diofant.solvers.ode), 1028
 _linear_2eq_order1_type2() (in module diofant.solvers.ode), 1029
 _linear_2eq_order1_type3() (in module diofant.solvers.ode), 1030
 _linear_2eq_order1_type4() (in module diofant.solvers.ode), 1030
 _linear_2eq_order1_type5() (in module diofant.solvers.ode), 1031
 _linear_2eq_order1_type6() (in module diofant.solvers.ode), 1031
 _linear_2eq_order1_type7() (in module diofant.solvers.ode), 1031
 _linear_2eq_order2_type1() (in module diofant.solvers.ode), 1032
 _linear_2eq_order2_type11() (in module diofant.solvers.ode), 1037
 _linear_2eq_order2_type2() (in module diofant.solvers.ode), 1033
 _linear_2eq_order2_type3() (in module diofant.solvers.ode), 1034
 _linear_2eq_order2_type5() (in module diofant.solvers.ode), 1034
 _linear_2eq_order2_type6() (in module diofant.solvers.ode), 1035
 _linear_2eq_order2_type7() (in module diofant.solvers.ode), 1035
 _linear_2eq_order2_type8() (in module diofant.solvers.ode), 1036
 _linear_2eq_order2_type9() (in module diofant.solvers.ode), 1036
 _linear_3eq_order1_type4() (in module diofant.solvers.ode), 1038
 _linear_neq_order1_type1() (in module diofant.solvers.ode), 1038
 _modgcd_multivariate_p() (in module diofant.polys.modulargcd), 858
 naive_list_centralizer() (in module diofant.combinatorics.testutil), 243
 _nonlinear_2eq_order1_type1() (in module diofant.solvers.ode), 1038

- `_nonlinear_2eq_order1_type2()` (in module `diofant.solvers.ode`), 1039
 - `_nonlinear_2eq_order1_type3()` (in module `diofant.solvers.ode`), 1039
 - `_nonlinear_2eq_order1_type4()` (in module `diofant.solvers.ode`), 1040
 - `_nonlinear_2eq_order1_type5()` (in module `diofant.solvers.ode`), 1040
 - `_nonlinear_3eq_order1_type1()` (in module `diofant.solvers.ode`), 1040
 - `_nonlinear_3eq_order1_type2()` (in module `diofant.solvers.ode`), 1041
 - `_orbits_transversals_from_bsgs()` (in module `diofant.combinatorics.util`), 240
 - `_print()` (`diofant.printing.printer.Printer` method), 866
 - `_random_pr_init()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 190
 - `_remove_gens()` (in module `diofant.combinatorics.util`), 240
 - `_strip()` (in module `diofant.combinatorics.util`), 241
 - `_strong_gens_from_distr()` (in module `diofant.combinatorics.util`), 242
 - `_undetermined_coefficients_match()` (in module `diofant.solvers.ode`), 1044
 - `_union_find_merge()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 191
 - `_union_find_rep()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 191
 - `_verify_bsgs()` (in module `diofant.combinatorics.testutil`), 243
 - `_verify_centralizer()` (in module `diofant.combinatorics.testutil`), 244
 - `_verify_normal_closure()` (in module `diofant.combinatorics.testutil`), 244
- A**
- `a2idx()` (in module `diofant.matrices.matrices`), 629
 - `AbelianGroup()` (in module `diofant.combinatorics.named_groups`), 237
 - `above()` (`diofant.printing.pretty.stringpict.stringpic` method), 882
 - `Abs` (class in `diofant.functions.elementary.complexes`), 309
 - `abs()` (`diofant.polys.domains.domain.Domain` method), 765
 - `abs()` (`diofant.polys.polyclasses.DMP` method), 775
 - `abs()` (`diofant.polys.polytools.Poly` method), 692
 - `accepted_latex_functions` (in module `diofant.printing.latex`), 875
 - `acos` (class in `diofant.functions.elementary.trigonometric`), 318
 - `acosh` (class in `diofant.functions.elementary.hyperbolic`), 327
 - `acot` (class in `diofant.functions.elementary.trigonometric`), 321
 - `acoth` (class in `diofant.functions.elementary.hyperbolic`), 328
 - `acsc` (class in `diofant.functions.elementary.trigonometric`), 322
 - `Add` (class in `diofant.core.add`), 107
 - `add()` (`diofant.matrices.matrices.MatrixBase` method), 590
 - `add()` (`diofant.matrices.sparse.SparseMatrixBase` method), 643
 - `add()` (`diofant.polys.domains.domain.Domain` method), 765
 - `add()` (`diofant.polys.polyclasses.DMF` method), 780
 - `add()` (`diofant.polys.polyclasses.DMP` method), 775
 - `add()` (`diofant.polys.polytools.Poly` method), 692
 - `add()` (`diofant.polys.rings.PolyRing` method), 827
 - `add_formulae()` (in module `diofant.simplify.hyperexpand`), 956
 - `add_ground()` (`diofant.polys.polyclasses.DMP` method), 775
 - `add_ground()` (`diofant.polys.polytools.Poly` method), 692
 - `adjoint` (class in `diofant.functions.elementary.complexes`), 310
 - `adjoint()` (`diofant.core.expr.Expr` method), 59
 - `adjoint()` (`diofant.matrices.immutable.ImmutableMatrix` method), 653
 - `adjoint()` (`diofant.matrices.matrices.MatrixBase` method), 590
 - `adjugate()` (`diofant.matrices.matrices.MatrixBase` method), 590
 - `airyai` (class in `diofant.functions.special.bessel`), 393

airyaiprime (class in diofant.functions.special.bessel), 395
 AiryBase (class in diofant.functions.special.bessel), 393
 airybi (class in diofant.functions.special.bessel), 394
 airybiprime (class in diofant.functions.special.bessel), 397
 algebraic_field() (diofant.polys.domains.AlgebraicField method), 772
 algebraic_field() (diofant.polys.domains.domain.Domain method), 765
 algebraic_field() (diofant.polys.domains.IntegerRing method), 771
 algebraic_field() (diofant.polys.domains.RationalField method), 772
 AlgebraicField (class in diofant.polys.domains), 772
 AlgebraicNumber (class in diofant.polys.numberfields), 733
 all_coeffs() (diofant.polys.polyclasses.DMP method), 776
 all_coeffs() (diofant.polys.polytools.Poly method), 692
 all_monoms() (diofant.polys.polyclasses.DMP method), 776
 all_monoms() (diofant.polys.polytools.Poly method), 693
 all_roots() (diofant.polys.polytools.Poly method), 693
 all_terms() (diofant.polys.polyclasses.DMP method), 776
 all_terms() (diofant.polys.polytools.Poly method), 693
 allhints (in module diofant.solvers.ode), 1002
 almosteq() (diofant.polys.domains.domain.Domain method), 765
 almosteq() (diofant.polys.domains.RealField method), 774
 almosteq() (diofant.polys.rings.PolyElement method), 828
 alternating() (diofant.combinatorics.generators method), 187
 AlternatingGroup() (in module diofant.combinatorics.named_groups), 236
 altitudes (diofant.geometry.polygon.Triangle attribute), 513
 ambient_dimension (diofant.geometry.entity.GeometryEntity attribute), 433
 ambient_dimension (diofant.geometry.point.Point attribute), 439
 an (diofant.functions.special.hyper.meijerg attribute), 408
 And (class in diofant.logic.boolalg), 569
 angle_between() (diofant.geometry.line.LinearEntity method), 450
 angle_between() (diofant.geometry.line3d.LinearEntity3D method), 468
 angle_between() (diofant.geometry.plane.Plane method), 520
 angles (diofant.geometry.polygon.Polygon attribute), 500
 angles (diofant.geometry.polygon.RegularPolygon attribute), 506
 annotated() (in module diofant.printing.pretty.pretty_symbology), 882
 ANP (class in diofant.polys.polyclasses), 781
 aother (diofant.functions.special.hyper.meijerg attribute), 408
 ap (diofant.functions.special.hyper.hyper attribute), 406
 ap (diofant.functions.special.hyper.meijerg attribute), 408
 apart() (diofant.core.expr.Expr method), 59
 apart() (in module diofant.polys.partfrac), 737
 apart_list() (in module diofant.polys.partfrac), 738
 apoapsis (diofant.geometry.ellipse.Ellipse attribute), 485
 apothem (diofant.geometry.polygon.RegularPolygon attribute), 506
 append() (diofant.plotting.plot.Plot method), 888
 apply() (diofant.simplify.epathtools.EPath method), 944
 apply_finite_diff() (in module diofant.calculus.finite_diff), 1173
 applyfunc() (diofant.matrices.dense.DenseMatrix method), 630
 applyfunc() (diofant.matrices.sparse.SparseMatrixBase method), 644
 approximation_interval() (diofant.core.numbers.NumberSymbol method), 92
 arbitrary_point() (diofant.geometry.curve.Curve method),

- 481
- arbitrary_point() (diofant.geometry.ellipse.Ellipse method), 486
- arbitrary_point() (diofant.geometry.line.LinearEntity method), 450
- arbitrary_point() (diofant.geometry.line3d.LinearEntity3D method), 469
- arbitrary_point() (diofant.geometry.plane.Plane method), 520
- arbitrary_point() (diofant.geometry.polygon.Polygon method), 500
- Arcsin() (in module diofant.stats), 960
- are_collinear() (diofant.geometry.point.Point3D static method), 446
- are_concurrent() (diofant.geometry.line.LinearEntity static method), 451
- are_concurrent() (diofant.geometry.line3d.LinearEntity3D static method), 469
- are_concurrent() (diofant.geometry.plane.Plane static method), 520
- are_coplanar() (diofant.geometry.point.Point3D static method), 446
- are_similar() (in module diofant.geometry.util), 437
- area (diofant.geometry.ellipse.Ellipse attribute), 486
- area (diofant.geometry.polygon.Polygon attribute), 501
- area (diofant.geometry.polygon.RegularPolygonas attribute), 506
- arg (class in diofant.functions.elementary.complexes), 310
- args (diofant.core.basic.Basic attribute), 42
- args (diofant.core.containers.Dict attribute), 149
- args (diofant.geometry.polygon.RegularPolygonas attribute), 507
- args (diofant.polys.polytools.Poly attribute), 693
- args (diofant.tensor.indexed.IndexedBase attribute), 1102
- args_cnc() (diofant.core.expr.Expr method), 59
- Argument (class in diofant.utilities.codegen), 1129
- argument (diofant.functions.special.bessel.BesselBase attribute), 388
- argument (diofant.functions.special.hyper.hyper attribute), 406
- argument (diofant.functions.special.hyper.meijerg attribute), 408
- array_form (diofant.combinatorics.permutations.Permutation attribute), 167
- array_form (diofant.combinatorics.polyhedron.Polyhedron attribute), 217
- as_base_exp() (diofant.core.expr.Expr method), 60
- as_base_exp() (diofant.core.function.Function method), 135
- as_base_exp() (diofant.core.mul.Mul method), 103
- as_base_exp() (diofant.core.power.Pow method), 101
- as_base_exp() (diofant.functions.elementary.exponential.log method), 331
- as_coeff_Add() (diofant.core.add.Add method), 107
- as_coeff_add() (diofant.core.add.Add method), 107
- as_coeff_Add() (diofant.core.expr.Expr method), 60
- as_coeff_add() (diofant.core.expr.Expr method), 60
- as_coeff_Add() (diofant.core.numbers.Number method), 86
- as_coeff_add() (diofant.core.numbers.Number method), 86
- as_coeff_exponent() (diofant.core.expr.Expr method), 60
- as_coeff_Mul() (diofant.core.expr.Expr method), 60
- as_coeff_mul() (diofant.core.expr.Expr method), 60
- as_coeff_Mul() (diofant.core.mul.Mul method), 103
- as_coeff_mul() (diofant.core.mul.Mul method), 103
- as_coeff_Mul() (diofant.core.numbers.Number method), 86
- as_coeff_mul() (diofant.core.numbers.Number method), 86

`as_coefficient()` (diofant.core.expr.Expr method), 61
`as_coefficients_dict()` (diofant.core.add.Add method), 107
`as_coefficients_dict()` (diofant.core.expr.Expr method), 62
`as_content_primitive()` (diofant.core.add.Add method), 107
`as_content_primitive()` (diofant.core.basic.Basic method), 43
`as_content_primitive()` (diofant.core.expr.Expr method), 62
`as_content_primitive()` (diofant.core.mul.Mul method), 104
`as_content_primitive()` (diofant.core.numbers.Rational method), 90
`as_content_primitive()` (diofant.core.power.Pow method), 102
`as_dict()` (diofant.combinatorics.partitions.IntegerPartition method), 160
`as_dict()` (diofant.polys.polytools.Poly method), 694
`as_dummy()` (diofant.concrete.expr_with_limits.ExprWithLimits method), 288
`as_explicit()` (diofant.matrices.expressions.MatrixExpr method), 655
`as_expr()` (diofant.core.expr.Expr method), 63
`as_expr()` (diofant.polys.numberfields.AlgebraicNumber method), 733
`as_expr()` (diofant.polys.polytools.Poly method), 694
`as_ferrers()` (diofant.combinatorics.partitions.IntegerPartition method), 160
`as_finite_diff()` (in module diofant.calculus.finite_diff), 1174
`as_immutable()` (diofant.matrices.dense.DenseMatrix method), 630
`as_immutable()` (diofant.matrices.sparse.SparseMatrixBase method), 644
`as_independent()` (diofant.core.expr.Expr method), 63
`as_int()` (in module diofant.core.compatibility), 149, 155
`as_leading_term()` (diofant.core.expr.Expr method), 66
`as_list()` (diofant.polys.polytools.Poly method), 694
`as_mutable()` (diofant.matrices.dense.DenseMatrix method), 630
`as_mutable()` (diofant.matrices.expressions.MatrixExpr method), 656
`as_mutable()` (diofant.matrices.immutable.ImmutableMatrix method), 653
`as_mutable()` (diofant.matrices.sparse.SparseMatrixBase method), 644
`as_numer_denom()` (diofant.core.add.Add method), 108
`as_numer_denom()` (diofant.core.expr.Expr method), 66
`as_numer_denom()` (diofant.core.mul.Mul method), 104
`as_numer_denom()` (diofant.core.numbers.Integer method), 91
`as_numer_denom()` (diofant.core.numbers.Rational method), 91
`as_numer_denom()` (diofant.core.power.Pow method), 102
`as_numer_denom()` (diofant.functions.elementary.exponential.exp_polar method), 330
`as_ordered_factors()` (diofant.core.expr.Expr method), 66
`as_ordered_factors()` (diofant.core.mul.Mul method), 104
`as_ordered_terms()` (diofant.core.expr.Expr method), 66
`as_poly()` (diofant.core.basic.Basic method), 43
`as_poly()` (diofant.polys.numberfields.AlgebraicNumber method), 733
`as_powers_dict()` (diofant.core.expr.Expr method), 66
`as_powers_dict()` (diofant.core.mul.Mul method), 104
`as_real_imag()` (diofant.core.add.Add method), 108
`as_real_imag()` (diofant.core.expr.Expr method), 67
`as_real_imag()` (diofant.core.mul.Mul method), 104
`as_real_imag()` (diofant.core.power.Pow method), 103
`as_real_imag()` (diofant.functions.elementary.complexes.im method), 308
`as_real_imag()` (diofant.functions.elementary.complexes.re method), 307
`as_real_imag()` (diofant.functions.elementary.exponential.log method), 331
`as_real_imag()` (diofant.functions.elementary.hyperbolic.sinh method), 331

- method), 325
 - as_relational() (diofant.sets.sets.FiniteSet method), 916
 - as_relational() (diofant.sets.sets.Intersection method), 917
 - as_relational() (diofant.sets.sets.Interval method), 914
 - as_relational() (diofant.sets.sets.Union method), 916
 - as_set() (diofant.core.relational.Relational method), 111
 - as_sum() (diofant.integrals.integrals.Integral method), 553
 - as_terms() (diofant.core.expr.Expr method), 67
 - as_two_terms() (diofant.core.add.Add method), 108
 - as_two_terms() (diofant.core.mul.Mul method), 104
 - ascents() (diofant.combinatorics.permutations.Permutation method), 168
 - asec (class in diofant.functions.elementary.trigonometric), 319, 321
 - aseries() (diofant.core.expr.Expr method), 67
 - asin (class in diofant.functions.elementary.trigonometric), 317
 - asinh (class in diofant.functions.elementary.hyperbolic), 327
 - assemble_partfrac_list() (in module diofant.polys.partfrac), 740
 - AssignmentError, 881
 - assoc_laguerre (class in diofant.functions.special.polynomials), 420
 - assoc_legendre (class in diofant.functions.special.polynomials), 418
 - assumptions0 (diofant.core.basic.Basic attribute), 43
 - atan (class in diofant.functions.elementary.trigonometric), 320
 - atan2 (class in diofant.functions.elementary.trigonometric), 323
 - atanh (class in diofant.functions.elementary.hyperbolic), 328
 - Atom (class in diofant.core.basic), 41
 - AtomicExpr (class in diofant.core.expr), 80
 - atoms() (diofant.combinatorics.permutations.Permutation method), 168
 - atoms() (diofant.core.basic.Basic method), 43
 - atoms() (diofant.matrices.matrices.MatrixBase method), 590
 - atoms_table (in module diofant.printing.pretty.pretty_symbology), 882
 - auto_number() (in module diofant.parsing.sympy_parser), 1170
 - auto_symbol() (in module diofant.parsing.sympy_parser), 1170
 - AutomaticSymbols (class in diofant.interactive.session), 886
 - autowrap() (in module diofant.utilities.autowrap), 1124
 - AxisOrienter (class in diofant.vector.orienters), 1220
- ## B
- Permutation
 - base (diofant.combinatorics.perm_groups.PermutationGroup attribute), 191
 - base (diofant.core.power.Pow attribute), 103
 - base (diofant.tensor.indexed.Indexed attribute), 1100
 - base_oneform() (diofant.diffgeom.CoordSystem method), 1179
 - base_oneforms() (diofant.diffgeom.CoordSystem method), 1179
 - base_solution_linear() (in module diofant.solvers.diophantine), 1079
 - base_vector() (diofant.diffgeom.CoordSystem method), 1180
 - base_vectors() (diofant.diffgeom.CoordSystem method), 1180
 - BaseCovarDerivativeOp (class in diofant.diffgeom), 1186
 - BasePolynomialError (class in diofant.polys.polyerrors), 853
 - BaseScalarField (class in diofant.diffgeom), 1181
 - BaseSeries (class in diofant.plotting.plot), 899
 - baseswap() (diofant.combinatorics.perm_groups.Permutation method), 192
 - BaseVectorField (class in diofant.diffgeom), 1182
 - Basic (class in diofant.core.basic), 42
 - basic_orbits (diofant.combinatorics.perm_groups.Permutation attribute), 193

basic_stabilizers (diofant.combinatorics.perm_groups.PermutationGroup attribute), 193
 basic_transversals (diofant.combinatorics.perm_groups.PermutationGroup attribute), 194
 basis() (diofant.polys.agca.modules.FreeModule method), 748
 bell (class in diofant.functions.combinatorial.numbers), 339
 below() (diofant.printing.pretty.stringpict.stringPict method), 883
 Benini() (in module diofant.stats), 961
 berkowitz() (diofant.matrices.matrices.MatrixBase method), 590
 berkowitz_charpoly() (diofant.matrices.matrices.MatrixBase method), 591
 berkowitz_det() (diofant.matrices.matrices.MatrixBase method), 592
 berkowitz_eigenvals() (diofant.matrices.matrices.MatrixBase method), 592
 berkowitz_minors() (diofant.matrices.matrices.MatrixBase method), 592
 bernoulli (class in diofant.functions.combinatorial.numbers), 340
 Bernoulli() (in module diofant.stats), 957
 BesselBase (class in diofant.functions.special.bessel), 388
 besseli (class in diofant.functions.special.bessel), 389
 besselj (class in diofant.functions.special.bessel), 388
 bess elk (class in diofant.functions.special.bessel), 390
 besselsimp() (in module diofant.simplify.simplify), 925
 bessely (class in diofant.functions.special.bessel), 389
 beta (class in diofant.functions.special.beta_functions), 366
 Beta() (in module diofant.stats), 961
 BetaPrime() (in module diofant.stats), 962
 bin_to_gray() (diofant.combinatorics.graycode method), 234
 binary_function() (in module diofant.utilities.autowrap), 1125
 binary_partitions() (in module diofant.utilities.iterables), 1144
 binomial (class in diofant.functions.combinatorial.factorials), 341
 Binomial() (in module diofant.stats), 958
 binomial_coefficients() (in module diofant.ntheory.multinomial), 268
 binomial_coefficients_list() (in module diofant.ntheory.multinomial), 268
 bisectors() (diofant.geometry.polygon.Triangle method), 513
 bitlist_from_subset() (diofant.combinatorics.subsets.Subset class method), 224
 block_collapse() (in module diofant.matrices.expressions.blockmatrix), 660
 BlockDiagMatrix (class in diofant.matrices.expressions.blockmatrix), 659
 BlockMatrix (class in diofant.matrices.expressions.blockmatrix), 659
 bm (diofant.functions.special.hyper.meijerg attribute), 409
 BodyOrienter (class in diofant.vector.orienters), 1221
 bool_map() (in module diofant.logic.boolalg), 575
 BooleanFalse (class in diofant.logic.boolalg), 568
 BooleanTrue (class in diofant.logic.boolalg), 567
 bother (diofant.functions.special.hyper.meijerg attribute), 409
 boundary (diofant.sets.sets.Set attribute), 908
 bounds (diofant.geometry.point.Point2D attribute), 443
 bq (diofant.functions.special.hyper.hyper attribute), 406
 bq (diofant.functions.special.hyper.meijerg attribute), 409
 bracelets() (in module diofant.utilities.iterables), 1144
 bsgs_direct_product() (in module diofant.combinatorics.tensor_can), 249
 bspline_basis() (in module diofant.functions.special.bsplines), 398
 bspline_basis_set() (in module diofant.functions.special.bsplines),

- 399
- C**
- C (diofant.matrices.immutable.ImmutableMatrix attribute), 653
 - C (diofant.matrices.matrices.MatrixBase attribute), 586
 - cacheit() (in module diofant.core.cache), 41
 - cancel() (diofant.core.expr.Expr method), 68
 - cancel() (diofant.polys.polyclasses.DMF method), 780
 - cancel() (diofant.polys.polyclasses.DMP method), 776
 - cancel() (diofant.polys.polytools.Poly method), 694
 - cancel() (diofant.polys.rings.PolyElement method), 828
 - cancel() (in module diofant.polys.polytools), 688
 - canon_bp() (diofant.tensor.tensor.TensAdd method), 1117
 - canon_bp() (diofant.tensor.tensor.TensMul method), 1119
 - canon_bp() (in module diofant.tensor.tensor), 1121
 - canonical (diofant.core.relational.Relational attribute), 111
 - canonical_variables (diofant.core.basic.Basic attribute), 45
 - canonicalize() (in module diofant.combinatorics.tensor_can), 244
 - cantor_product() (in module diofant.utilities.iterables), 1144
 - capture() (in module diofant.utilities.iterables), 1145
 - cardinality (diofant.combinatorics.permutations.Permutation attribute), 168
 - cardinality (diofant.combinatorics.subsets.Subset attribute), 224
 - casoratian() (in module diofant.matrices.dense), 625
 - Catalan (class in diofant.core.numbers), 99
 - catalan (class in diofant.functions.combinatorial.numbers), 343
 - Cauchy() (in module diofant.stats), 963
 - ccode() (in module diofant.printing.ccode), 868
 - CCodeGen (class in diofant.utilities.codegen), 1131
 - CCodePrinter (class in diofant.printing.ccode), 868
 - ceiling (class in diofant.functions.elementary.integers), 328
 - ceiling() (diofant.core.numbers.Float method), 89
 - center (diofant.geometry.ellipse.Ellipse attribute), 486
 - center (diofant.geometry.polygon.RegularPolygon attribute), 507
 - center() (diofant.combinatorics.perm_groups.PermutationGroup method), 194
 - centralizer() (diofant.combinatorics.perm_groups.PermutationGroup method), 195
 - centroid (diofant.geometry.polygon.Polygon attribute), 501
 - centroid (diofant.geometry.polygon.RegularPolygon attribute), 507
 - centroid() (in module diofant.geometry.util), 437
 - change_index() (diofant.concrete.expr_with_intlimits.ExprWithIntLimits method), 290
 - characteristic() (diofant.polys.domains.domain.Domain method), 765
 - characteristic() (diofant.polys.domains.FiniteField method), 770
 - charpoly() (diofant.matrices.matrices.MatrixBase method), 592
 - chebyshevt (class in diofant.functions.special.polynomials), 415
 - chebyshevt_poly() (in module diofant.polys.orthopolys), 736
 - chebyshevt_root (class in diofant.functions.special.polynomials), 416
 - chebyshevu (class in diofant.functions.special.polynomials), 415
 - chebyshevu_poly() (in module diofant.polys.orthopolys), 736
 - chebyshevu_root (class in diofant.functions.special.polynomials), 417
 - check_assumptions() (in module diofant.solvers.solvers), 1065
 - checkinfsol() (in module diofant.solvers.ode), 1001
 - checkodesol() (in module diofant.solvers.ode), 998
 - checkpdesol() (in module diofant.solvers.pde), 1049

`checksol()` (in module `diofant.solvers.solvers`), 1065
`Chi` (class in `diofant.functions.special.error_functions`), 386
`Chi()` (in module `diofant.stats`), 964
`ChiNoncentral()` (in module `diofant.stats`), 964
`ChiSquared()` (in module `diofant.stats`), 965
`cholesky()` (`diofant.matrices.matrices.MatrixBase` method), 593
`cholesky()` (`diofant.matrices.sparse.SparseMatrixBase` method), 645
`cholesky_solve()` (`diofant.matrices.matrices.MatrixBase` method), 593
`Ci` (class in `diofant.functions.special.error_functions`), 384
`Circle` (class in `diofant.geometry.ellipse`), 495
`circumcenter` (`diofant.geometry.polygon.RegularPolygon` attribute), 507
`circumcenter` (`diofant.geometry.polygon.Triangle` attribute), 514
`circumcircle` (`diofant.geometry.polygon.RegularPolygon` attribute), 508
`circumcircle` (`diofant.geometry.polygon.Triangle` attribute), 514
`circumference` (`diofant.geometry.ellipse.Circle` attribute), 496
`circumference` (`diofant.geometry.ellipse.Ellipse` attribute), 487
`circumradius` (`diofant.geometry.polygon.RegularPolygon` attribute), 508
`circumradius` (`diofant.geometry.polygon.Triangle` attribute), 515
`CL` (`diofant.matrices.sparse.SparseMatrixBase` attribute), 643
`class_key()` (`diofant.core.add.Add` class method), 108
`class_key()` (`diofant.core.basic.Atom` class method), 41
`class_key()` (`diofant.core.basic.Basic` class method), 45
`class_key()` (`diofant.core.function.Function` class method), 135
`class_key()` (`diofant.core.mul.Mul` class method), 105
`class_key()` (`diofant.core.numbers.Number` class method), 86
`class_key()` (`diofant.core.power.Pow` class method), 103
`class_key()` (`diofant.core.symbol.Dummy` class method), 83
`classify_diop()` (in module `diofant.solvers.diophantine`), 1078
`classify_ode()` (in module `diofant.solvers.ode`), 997
`classify_pde()` (in module `diofant.solvers.pde`), 1048
`classof()` (in module `diofant.matrices.matrices`), 621
`clear()` (`diofant.tensor.tensor.TensorManager` method), 1106
`clear_denoms()` (`diofant.polys.polyclasses.DMP` method), 776
`clear_denoms()` (`diofant.polys.polytools.Poly` method), 695
`CodeGen` (class in `diofant.utilities.codegen`), 1129
`codegen()` (in module `diofant.utilities.codegen`), 1133
`CodePrinter` (class in `diofant.printing.codeprinter`), 881
`CodeWrapper` (class in `diofant.utilities.autowrap`), 1123
`coeff()` (`diofant.core.expr.Expr` method), 68
`coeff()` (`diofant.polys.rings.PolyElement` method), 828
`coeff_monomial()` (`diofant.polys.polytools.Poly` method), 695
`coefficients` (`diofant.geometry.line.LinearEntity` attribute), 451
`coeffs()` (`diofant.polys.numberfields.AlgebraicNumber` method), 733
`coeffs()` (`diofant.polys.polyclasses.DMP` method), 776
`coeffs()` (`diofant.polys.polytools.Poly` method), 695
`coeffs()` (`diofant.polys.rings.PolyElement` method), 829
`CoercionFailed` (class in `diofant.polys.polyerrors`), 853
`cofactor()` (`diofant.matrices.matrices.MatrixBase` method), 593
`cofactorMatrix()` (`diofant.matrices.matrices.MatrixBase` method), 593
`cofactors()` (`diofant.core.numbers.Number` method), 86

cofactors() (diofant.polys.domains.domain.Domain method), 765
 cofactors() (diofant.polys.polyclasses.DMP method), 776
 cofactors() (diofant.polys.polytools.Poly method), 696
 cofactors() (in module diofant.polys.polytools), 681
 Coin() (in module diofant.stats), 957
 col() (diofant.matrices.dense.DenseMatrix method), 631
 col() (diofant.matrices.sparse.SparseMatrixBase method), 645
 col_del() (diofant.matrices.dense.MutableDenseMatrix method), 633
 col_del() (diofant.matrices.sparse.MutableSparseMatrix method), 639
 col_insert() (diofant.matrices.matrices.MatrixBase method), 594
 col_join() (diofant.matrices.matrices.MatrixBase method), 594
 col_join() (diofant.matrices.sparse.MutableSparseMatrix method), 639
 col_list() (diofant.matrices.sparse.SparseMatrixBase method), 645
 col_op() (diofant.matrices.dense.MutableDenseMatrix method), 633
 col_op() (diofant.matrices.sparse.MutableSparseMatrix method), 640
 col_swap() (diofant.matrices.dense.MutableDenseMatrix method), 633
 col_swap() (diofant.matrices.sparse.MutableSparseMatrix method), 640
 collect() (diofant.core.expr.Expr method), 70
 collect() (in module diofant.simplify.radsimp), 930
 collect_const() (in module diofant.simplify.radsimp), 934
 collect_sqrt() (in module diofant.simplify.radsimp), 933
 combsimp() (diofant.core.expr.Expr method), 70
 combsimp() (in module diofant.simplify.combsimp), 940
 comm_i2symbol() (diofant.tensor.tensor._TensorManager method), 1106
 comm_symbols2i() (diofant.tensor.tensor._TensorManager method), 1106
 common_prefix() (in module diofant.utilities.iterables), 1145
 common_suffix() (in module diofant.utilities.iterables), 1145
 Commutator (class in diofant.diffgeom), 1183
 commutator() (diofant.combinatorics.perm_groups.PermutationGroup method), 195
 commutator() (diofant.combinatorics.permutations.Permutation method), 168
 commutes_with() (diofant.combinatorics.permutations.Permutation method), 169
 commutes_with() (diofant.tensor.tensor.TensorHead method), 1115
 Commutator() (diofant.core.basic.Basic method), 45
 Complement (class in diofant.series.gruntz), 906
 complement() (diofant.sets.sets.Set method), 909
 ComplexInfinity (class in diofant.core.numbers), 97
 components (diofant.vector.dyadic.Dyadic attribute), 1217
 components (diofant.vector.vector.Vector attribute), 1215
 components() (in module diofant.integrals.heurisch), 551
 compose() (diofant.polys.polyclasses.DMP method), 776
 compose() (diofant.polys.polytools.Poly method), 696
 compose() (diofant.polys.rings.PolyElement method), 829
 compose() (in module diofant.polys.polytools), 683
 CompositeDomain (class in diofant.polys.domains.compositedomain), 770
 ComputationFailed (class in diofant.polys.polyerrors), 853
 compute_leading_term() (diofant.core.expr.Expr method), 70
 cond (diofant.functions.elementary.piecewise.ExprCondPair attribute), 332
 condition_number() (diofant.matrices.matrices.MatrixBase method), 594
 ConditionalDomain (class in diofant.stats.rv), 991
 conjugate (class in diofant.functions.elementary.complexes), 310

- conjugate (diofant.combinatorics.partitions.IntegerPartition attribute), 160
- conjugate() (diofant.core.expr.Expr method), 70
- conjugate() (diofant.matrices.immutable.ImmutableMatrix method), 653
- conjugate() (diofant.matrices.matrices.MatrixBase method), 595
- connect_to() (diofant.diffgeom.CoordSystem method), 1180
- conserve_mpmath_dps() (in module diofant.utilities.decorator), 1137
- const() (diofant.polys.rings.PolyElement method), 829
- constant_renumber() (in module diofant.solvers.ode), 1003
- constantsimp() (in module diofant.solvers.ode), 1003
- construct_domain() (in module diofant.polys.constructor), 731
- contains() (diofant.combinatorics.perm_groups.PermutationGroup method), 196
- contains() (diofant.geometry.line.Line method), 458
- contains() (diofant.geometry.line.LinearEntity method), 452
- contains() (diofant.geometry.line.Ray method), 461
- contains() (diofant.geometry.line.Segment method), 464
- contains() (diofant.geometry.line3d.Line3D method), 466
- contains() (diofant.geometry.line3d.LinearEntity3D method), 470
- contains() (diofant.geometry.line3d.Ray3D method), 476
- contains() (diofant.geometry.line3d.Segment3D method), 479
- contains() (diofant.polys.agca.ideals.Ideal method), 754
- contains() (diofant.polys.agca.modules.Module method), 747
- contains() (diofant.polys.polytools.GroebnerBasis method), 728
- contains() (diofant.series.order.Order method), 904
- contains() (diofant.sets.sets.Set method), 909
- content() (diofant.polys.polyclasses.DMP method), 776
- content() (diofant.polys.polytools.Poly method), 696
- content() (diofant.polys.rings.PolyElement method), 829
- continued_fraction_convergents() (in module diofant.ntheory.continued_fraction), 274
- continued_fraction_iterator() (in module diofant.ntheory.continued_fraction), 274
- continued_fraction_periodic() (in module diofant.ntheory.continued_fraction), 275
- continued_fraction_reduce() (in module diofant.ntheory.continued_fraction), 276
- ContinuousDomain (class in diofant.stats.crv), 991
- ContinuousPSpace (class in diofant.stats.crv), 991
- ContinuousRV() (in module diofant.stats), 986
- contract_metric() (diofant.tensor.tensor.TensAdd method), 1117
- contract_metric() (diofant.tensor.tensor.TensMul method), 1119
- convergence_statement (diofant.functions.special.hyper.hyper attribute), 406
- convert() (diofant.polys.agca.modules.FreeModule method), 748
- convert() (diofant.polys.agca.modules.Module method), 747
- convert() (diofant.polys.agca.modules.QuotientModule method), 756
- convert() (diofant.polys.agca.modules.SubModule method), 750
- convert() (diofant.polys.domains.domain.Domain method), 765
- convert() (diofant.polys.polyclasses.DMP method), 776
- convert_from() (diofant.polys.domains.domain.Domain method), 765
- convert_xor() (in module diofant.parsing.sympy_parser), 1170
- convex_hull() (in module diofant.geometry.util), 436
- coord_function() (diofant.diffgeom.CoordSystem method), 1180
- coord_functions() (diofant.diffgeom.CoordSystem method), 1180
- coord_tuple_transform_to() (diofant.diffgeom.CoordSystem method), 1180

[coords\(\)](#) (diofant.diffgeom.Point method), [1181](#)
[CoordSysCartesian](#) (class in diofant.vector.coordsysrect), [1209](#)
[CoordSystem](#) (class in diofant.diffgeom), [1178](#)
[copy\(\)](#) (diofant.core.basic.Basic method), [45](#)
[copy\(\)](#) (diofant.polys.rings.PolyElement method), [829](#)
[copyin_list\(\)](#) (diofant.matrices.dense.MutableDenseMatrix method), [634](#)
[copyin_matrix\(\)](#) (diofant.matrices.dense.MutableDenseMatrix method), [634](#)
[core\(\)](#) (in module diofant.ntheory.factor_), [265](#)
[cornacchia\(\)](#) (in module diofant.solvers.diophantine), [1081](#)
[corners](#) (diofant.combinatorics.polyhedron.Polyhedron attribute), [217](#)
[cos](#) (class in diofant.functions.elementary.trigonometric), [313](#)
[coset_factor\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), [197](#)
[coset_rank\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), [197](#)
[coset_unrank\(\)](#) (diofant.combinatorics.perm_groups.PermutationGroup method), [198](#)
[cosh](#) (class in diofant.functions.elementary.hyperbolic), [326](#)
[cosine_transform\(\)](#) (in module diofant.integrals.transforms), [530](#)
[CosineTransform](#) (class in diofant.integrals.transforms), [558](#)
[cot](#) (class in diofant.functions.elementary.trigonometric), [315](#)
[coth](#) (class in diofant.functions.elementary.hyperbolic), [326](#)
[could_extract_minus_sign\(\)](#) (diofant.core.expr.Expr method), [70](#)
[count\(\)](#) (diofant.core.basic.Basic method), [45](#)
[count_complex_roots\(\)](#) (diofant.polys.polyclasses.DMP method), [776](#)
[count_ops\(\)](#) (diofant.core.basic.Basic method), [45](#)
[count_ops\(\)](#) (diofant.core.expr.Expr method), [70](#)
[count_ops\(\)](#) (in module diofant.core.function), [142](#)
[count_partitions\(\)](#) (diofant.utilities.enumerative.MultisetPartitionTraverse method), [1140](#)
[count_real_roots\(\)](#) (diofant.polys.polyclasses.DMP method), [776](#)
[count_roots\(\)](#) (diofant.polys.polytools.Poly method), [696](#)
[count_roots\(\)](#) (in module diofant.polys.polytools), [687](#)
[CovarDerivativeOp](#) (class in diofant.diffgeom), [1187](#)
[cross\(\)](#) (diofant.matrices.matrices.MatrixBase method), [595](#)
[cross\(\)](#) (diofant.vector.deloperator.Del method), [1219](#)
[cross\(\)](#) (diofant.vector.dyadic.Dyadic method), [1218](#)
[cross\(\)](#) (diofant.vector.vector.Vector method), [1215](#)
[crt\(\)](#) (in module diofant.ntheory.modular), [266](#)
[crt\(\)](#) (in module diofant.ntheory.modular), [267](#)
[crt\(\)](#) (in module diofant.ntheory.modular), [267](#)
[csc](#) (class in diofant.functions.elementary.trigonometric), [317](#)
[csch](#) (class in diofant.functions.elementary.hyperbolic), [327](#)
[cse\(\)](#) (in module diofant.simplify.cse_main), [941](#)
[curl\(\)](#) (in module diofant.vector), [1225](#)
[current](#) (diofant.combinatorics.graycode.GrayCode attribute), [231](#)
[Curve](#) (class in diofant.geometry.curve), [480](#)
[Cycle](#) (class in diofant.combinatorics.permutations), [184](#)
[cycle_length\(\)](#) (in module diofant.ntheory.generate), [255](#)
[cycle_structure](#) (diofant.combinatorics.permutations.Permutation attribute), [169](#)
[cycles](#) (diofant.combinatorics.permutations.Permutation attribute), [169](#)
[cyclic\(\)](#) (diofant.combinatorics.generators method), [187](#)
[cyclic_form](#) (diofant.combinatorics.permutations.Permutation attribute), [170](#)

- cyclic_form (diofant.combinatorics.polyhedron.Polyhedron attribute), 217
- CyclicGroup() (in module diofant.combinatorics.named_groups), 235
- cyclotomic_poly() (in module diofant.polys.specialpolys), 735
- CythonCodeWrapper (class in diofant.utilities.autowrap), 1123
- ## D
- D (diofant.matrices.matrices.MatrixBase attribute), 586
- Dagum() (in module diofant.stats), 965
- DataType (class in diofant.utilities.codegen), 1129
- debug() (in module diofant.utilities.misc), 1166
- decompose() (diofant.polys.polyclasses.DMP method), 776
- decompose() (diofant.polys.polytools.Poly method), 697
- decompose() (in module diofant.polys.polytools), 683
- default_sort_key() (in module diofant.core.compatibility), 150
- deflate() (diofant.polys.polyclasses.DMP method), 776
- deflate() (diofant.polys.polytools.Poly method), 697
- degree (diofant.combinatorics.perm_groups.PermutationGroup attribute), 198
- degree() (diofant.polys.polyclasses.DMP method), 776
- degree() (diofant.polys.polytools.Poly method), 697
- degree() (diofant.polys.rings.PolyElement method), 830
- degree() (in module diofant.polys.polytools), 672
- degree_list() (diofant.polys.polyclasses.DMP method), 776
- degree_list() (diofant.polys.polytools.Poly method), 698
- degree_list() (in module diofant.polys.polytools), 673
- degrees() (diofant.polys.rings.PolyElement method), 830
- Del (class in diofant.vector.deloperator), 1219
- delta (diofant.functions.special.hyper.meijerg attribute), 409
- deltaintegrate() (in module diofant.integrals.deltafunctions), 549
- denom() (diofant.polys.domains.AlgebraicField method), 772
- denom() (diofant.polys.domains.domain.Domain method), 765
- denom() (diofant.polys.domains.ExpressionDomain method), 774
- denom() (diofant.polys.domains.FractionField method), 773
- denom() (diofant.polys.domains.ring.Ring method), 769
- denom() (diofant.polys.polyclasses.DMF method), 780
- DenseMatrix (class in diofant.matrices.dense), 629
- density() (in module diofant.stats), 987
- dependent() (in module diofant.stats.rv), 993
- depth() (diofant.polys.agca.ideals.Ideal method), 754
- Derivative (class in diofant.core.function), 129
- derived_series() (diofant.combinatorics.perm_groups.PermutationGroup method), 198
- derived_subgroup() (diofant.combinatorics.perm_groups.PermutationGroup method), 199
- descent() (in module diofant.solvers.diophantine), 1085
- descents() (diofant.combinatorics.permutations.Permutation method), 170
- det() (diofant.matrices.matrices.MatrixBase method), 595
- det_bareis() (diofant.matrices.matrices.MatrixBase method), 595
- det_LU_decomposition() (diofant.matrices.matrices.MatrixBase method), 595
- diag() (in module diofant.matrices.dense), 622
- diagonal_solve() (diofant.matrices.matrices.MatrixBase method), 595
- diagonalize() (diofant.matrices.matrices.MatrixBase method), 596
- Dict (class in diofant.core.containers), 148
- dict_merge() (in module diofant.utilities.iterables), 1145
- Die() (in module diofant.stats), 957
- DiePSpace (class in diofant.stats.frv_types), 991
- diff() (diofant.core.expr.Expr method), 70
- diff() (diofant.matrices.matrices.MatrixBase method), 596

- diff() (diofant.polys.polyclasses.DMP method), 776
- diff() (diofant.polys.polytools.Poly method), 698
- diff() (diofant.polys.rings.PolyElement method), 830
- diff() (in module diofant.core.function), 132
- Differential (class in diofant.diffgeom), 1184
- digamma() (in module diofant.functions.special.gamma_functions), 363
- digit_2txt (in module diofant.printing.pretty.pretty_symbology), 881
- dihedral() (diofant.combinatorics.generators method), 187
- DihedralGroup() (in module diofant.combinatorics.named_groups), 235
- diofant (module), 37
- diofant.calculus (module), 1170
- diofant.calculus.euler (module), 1170
- diofant.calculus.finite_diff (module), 1172
- diofant.calculus.optimization (module), 1172
- diofant.calculus.singularities (module), 1171
- diofant.combinatorics.generators (module), 186
- diofant.combinatorics.graycode (module), 230
- diofant.combinatorics.group_constructs (module), 243
- diofant.combinatorics.named_groups (module), 235
- diofant.combinatorics.partitions (module), 158
- diofant.combinatorics.perm_groups (module), 188
- diofant.combinatorics.permutations (module), 163
- diofant.combinatorics.polyhedron (module), 216
- diofant.combinatorics.prufer (module), 219
- diofant.combinatorics.subsets (module), 223
- diofant.combinatorics.tensor_can (module), 244
- diofant.combinatorics.testutil (module), 243
- diofant.combinatorics.util (module), 237
- diofant.core.add (module), 107
- diofant.core.assumptions (module), 40
- diofant.core.basic (module), 41
- diofant.core.cache (module), 41
- diofant.core.compatibility (module), 149
- diofant.core.containers (module), 148
- diofant.core.core (module), 58
- diofant.core.evalf (module), 147
- diofant.core.evaluate (module), 58
- diofant.core.expr (module), 59
- diofant.core.exprtools (module), 156
- diofant.core.function (module), 127
- diofant.core.mod (module), 110
- diofant.core.mul (module), 103
- diofant.core.multidimensional (module), 127
- diofant.core.numbers (module), 86
- diofant.core.power (module), 100
- diofant.core.relational (module), 110
- diofant.core.singleton (module), 58
- diofant.core.symbol (module), 81
- diofant.core.sympify (module), 37
- diofant.diffgeom (module), 1177
- diofant.functions (module), 305
- diofant.functions.special.bessel (module), 388
- diofant.functions.special.beta_functions (module), 366
- diofant.functions.special.elliptic_integrals (module), 409
- diofant.functions.special.error_functions (module), 367
- diofant.functions.special.gamma_functions (module), 358
- diofant.functions.special.polynomials (module), 411
- diofant.functions.special.zeta_functions (module), 400
- diofant.geometry.curve (module), 480
- diofant.geometry.ellipse (module), 484
- diofant.geometry.entity (module), 433
- diofant.geometry.line (module), 449
- diofant.geometry.line3d (module), 466
- diofant.geometry.plane (module), 519
- diofant.geometry.point (module), 438
- diofant.geometry.polygon (module), 498
- diofant.geometry.util (module), 435
- diofant.integrals (module), 525
- diofant.integrals.meijerint_doc (module), 543
- diofant.integrals.transforms (module), 526
- diofant.interactive (module), 885
- diofant.interactive.printing (module), 885
- diofant.interactive.session (module), 886
- diofant.logic (module), 566
- diofant.matrices (module), 577
- diofant.matrices.dense (module), 629
- diofant.matrices.expressions (module), 654
- diofant.matrices.expressions.blockmatrix (module), 659
- diofant.matrices.immutable (module), 653
- diofant.matrices.matrices (module), 577
- diofant.matrices.sparse (module), 639

- diofant.ntheory.continued_fraction (module), 274
- diofant.ntheory.egyptian_fraction (module), 277
- diofant.ntheory.factor_ (module), 256
- diofant.ntheory.generate (module), 250
- diofant.ntheory.modular (module), 266
- diofant.ntheory.multinomial (module), 268
- diofant.ntheory.partitions_ (module), 269
- diofant.ntheory.primetest (module), 270
- diofant.ntheory.residue_ntheory (module), 270
- diofant.plotting.plot (module), 886
- diofant.polys (module), 660
- diofant.printing.ccode (module), 868
- diofant.printing.codeprinter (module), 881
- diofant.printing.conventions (module), 880
- diofant.printing.fcode (module), 870
- diofant.printing.lambdarepr (module), 875
- diofant.printing.latex (module), 875
- diofant.printing.mathematica (module), 874
- diofant.printing.mathml (module), 877
- diofant.printing.precedence (module), 881
- diofant.printing.pretty.pretty (module), 867
- diofant.printing.pretty.pretty_symbology (module), 881
- diofant.printing.pretty.stringpict (module), 882
- diofant.printing.printer (module), 864
- diofant.printing.python (module), 878
- diofant.printing.repr (module), 878
- diofant.printing.str (module), 878
- diofant.printing.tree (module), 879
- diofant.series.gruntz (module), 905
- diofant.series.limits (module), 902
- diofant.series.order (module), 903
- diofant.series.residues (module), 904
- diofant.series.series (module), 903
- diofant.sets.fancysets (module), 919
- diofant.sets.sets (module), 908
- diofant.simplify.combsimp (module), 940
- diofant.simplify.cse_main (module), 901, 941
- diofant.simplify.epathtools (module), 944
- diofant.simplify.hyperexpand (module), 943
- diofant.simplify.hyperexpand_doc (module), 954
- diofant.simplify.powsimp (module), 937
- diofant.simplify.radsimp (module), 929
- diofant.simplify.ratsimp (module), 935
- diofant.simplify.sqrtdenest (module), 940
- diofant.simplify.traversaltools (module), 944
- diofant.simplify.trigsimp (module), 936
- diofant.solvers (module), 1054
- diofant.solvers.inequalities (module), 1093
- diofant.solvers.ode (module), 1041
- diofant.solvers.pde (module), 1053
- diofant.solvers.recurr (module), 1068
- diofant.stats (module), 956
- diofant.stats.crv (module), 991
- diofant.stats.crv_types (module), 991
- diofant.stats.Die() (in module diofant.stats.crv_types), 992
- diofant.stats.frv (module), 991
- diofant.stats.frv_types (module), 991
- diofant.stats.Normal() (in module diofant.stats.crv_types), 992
- diofant.stats.rv (module), 990
- diofant.tensor (module), 1096
- diofant.tensor.index_methods (module), 1103
- diofant.tensor.indexed (module), 1096
- diofant.tensor.tensor (module), 1106
- diofant.utilities (module), 1121
- diofant.utilities.autowrap (module), 1122
- diofant.utilities.codegen (module), 1128
- diofant.utilities.decorator (module), 1137
- diofant.utilities.enumerative (module), 1138
- diofant.utilities.iterables (module), 1144
- diofant.utilities.lambdify (module), 1162
- diofant.utilities.memoization (module), 1166
- diofant.utilities.misc (module), 1166
- diofant.utilities.randtest (module), 1166
- diofant.vector (module), 1193
- diofant.vector.orienters (module), 1220
- diop_bf_DN() (in module diofant.solvers.diophantine), 1081
- diop_DN() (in module diofant.solvers.diophantine), 1080
- diop_general_pythagorean() (in module diofant.solvers.diophantine), 1085
- diop_general_sum_of_squares() (in module diofant.solvers.diophantine), 1086
- diop_linear() (in module diofant.solvers.diophantine), 1078
- diop_quadratic() (in module diofant.solvers.diophantine), 1080
- diop_solve() (in module diofant.solvers.diophantine), 1077
- diop_ternary_quadratic() (in module diofant.solvers.diophantine), 1084
- diop_ternary_quadratic_normal() (in module diofant.solvers.diophantine), 1091
- diophantine() (in module diofant.solvers.diophantine), 1077
- DiracDelta (class in diofant.functions.special.delta_functions), 357
- direction (diofant.geometry.line.Ray attribute), 461

- direction_cosine (diofant.geometry.line3d.LinearEntity3D attribute), 470
- direction_cosine() (diofant.geometry.point.Point3D method), 447
- direction_ratio (diofant.geometry.line3d.LinearEntity3D attribute), 470
- direction_ratio() (diofant.geometry.point.Point3D method), 447
- DirectProduct() (in module diofant.combinatorics.group_constructs), 243
- dirichlet_eta (class in diofant.functions.special.zeta_functions), 401
- DiscreteUniform() (in module diofant.stats), 957
- discriminant() (diofant.polys.polyclasses.DMP method), 776
- discriminant() (diofant.polys.polytools.Poly method), 698
- discriminant() (in module diofant.polys.polytools), 677
- dispersion() (diofant.polys.polytools.Poly method), 698
- dispersion() (in module diofant.polys.dispersion), 677, 742
- dispersionset() (diofant.polys.polytools.Poly method), 699
- dispersionset() (in module diofant.polys.dispersion), 679, 741
- distance() (diofant.geometry.line.Line method), 458
- distance() (diofant.geometry.line.Ray method), 461
- distance() (diofant.geometry.line.Segment method), 464
- distance() (diofant.geometry.line3d.Line3D method), 467
- distance() (diofant.geometry.line3d.Ray3D method), 476
- distance() (diofant.geometry.line3d.Segment3D method), 479
- distance() (diofant.geometry.plane.Plane method), 521
- distance() (diofant.geometry.point.Point method), 439
- distance() (diofant.geometry.polygon.Polygon method), 501
- div() (diofant.polys.domains.domain.Domain method), 765
- div() (diofant.polys.domains.field.Field method), 768
- div() (diofant.polys.domains.ring.Ring method), 769
- div() (diofant.polys.polyclasses.DMP method), 776
- div() (diofant.polys.polytools.Poly method), 700
- div() (diofant.polys.rings.PolyElement method), 830
- div() (in module diofant.polys.polytools), 675
- divergence() (in module diofant.vector), 1225
- divisible() (in module diofant.solvers.diophantine), 1088
- divisor_count() (in module diofant.ntheory.factor_), 264
- divisors() (in module diofant.ntheory.factor_), 264
- DMF (class in diofant.polys.polyclasses), 780
- DMP (class in diofant.polys.polyclasses), 775
- dmp_abs() (in module diofant.polys.densearith), 796
- dmp_add() (in module diofant.polys.densearith), 796
- dmp_add_ground() (in module diofant.polys.densearith), 794
- dmp_add_mul() (in module diofant.polys.densearith), 796
- dmp_add_term() (in module diofant.polys.densearith), 793
- dmp_apply_pairs() (in module diofant.polys.densebasic), 793
- dmp_cancel() (in module diofant.polys.euclidtools), 846
- dmp_clear_denoms() (in module diofant.polys.densebasic), 809
- dmp_compose() (in module diofant.polys.densebasic), 808
- dmp_content() (in module diofant.polys.euclidtools), 846
- dmp_convert() (in module diofant.polys.densebasic), 786
- dmp_copy() (in module diofant.polys.densebasic), 785
- dmp_deflate() (in module diofant.polys.densebasic), 790
- dmp_degree() (in module diofant.polys.densebasic), 783
- dmp_degree_in() (in module diofant.polys.densebasic), 784
- dmp_degree_list() (in module diofant.polys.densebasic), 784

- | | | | | |
|---------------------------|---|--------------------------------|---|------|
| dmp_diff() | (in module fant.polys.denstools), 802 | dio- dmp_ground_monic() | (in module fant.polys.denstools), 804 | dio- |
| dmp_diff_eval_in() | (in module fant.polys.denstools), 803 | dio- dmp_ground_nth() | (in module fant.polys.densebasic), 786 | dio- |
| dmp_diff_in() | (in module fant.polys.denstools), 802 | dio- dmp_ground_p() | (in module fant.polys.densebasic), 787 | dio- |
| dmp_discriminant() | (in module fant.polys.euclidtools), 843 | dio- dmp_ground_primitive() | (in module fant.polys.denstools), 806 | dio- |
| dmp_div() | (in module fant.polys.densearith), 799 | dio- dmp_ground_TC() | (in module fant.polys.densebasic), 783 | dio- |
| dmp_eject() | (in module fant.polys.densebasic), 792 | dio- dmp_ground_trunc() | (in module fant.polys.denstools), 804 | dio- |
| dmp_euclidean_prs() | (in module fant.polys.euclidtools), 838 | dio- dmp_grounds() | (in module fant.polys.densebasic), 788 | dio- |
| dmp_eval() | (in module fant.polys.denstools), 802 | dio- dmp_half_gcdex() | (in module fant.polys.euclidtools), 838 | dio- |
| dmp_eval_in() | (in module fant.polys.denstools), 803 | dio- dmp_include() | (in module fant.polys.densebasic), 791 | dio- |
| dmp_eval_tail() | (in module fant.polys.denstools), 803 | dio- dmp_inflate() | (in module fant.polys.densebasic), 791 | dio- |
| dmp_exclude() | (in module fant.polys.densebasic), 791 | dio- dmp_inject() | (in module fant.polys.densebasic), 792 | dio- |
| dmp_expand() | (in module fant.polys.densearith), 801 | dio- dmp_inner_gcd() | (in module fant.polys.euclidtools), 845 | dio- |
| dmp_exquo() | (in module fant.polys.densearith), 800 | dio- dmp_inner_subresultants() | (in module fant.polys.euclidtools), 841 | dio- |
| dmp_exquo_ground() | (in module fant.polys.densearith), 795 | dio- dmp_integrate() | (in module fant.polys.denstools), 801 | dio- |
| dmp_ext_factor() | (in module fant.polys.factortools), 850 | dio- dmp_integrate_in() | (in module fant.polys.denstools), 802 | dio- |
| dmp_factor_list() | (in module fant.polys.factortools), 850 | dio- dmp_invert() | (in module fant.polys.euclidtools), 838 | dio- |
| dmp_factor_list_include() | (in module fant.polys.factortools), 850 | dio- dmp_irreducible_p() | (in module fant.polys.factortools), 850 | dio- |
| dmp_ff_div() | (in module fant.polys.densearith), 799 | dio- dmp_l1_norm() | (in module fant.polys.densearith), 801 | dio- |
| dmp_ff_prs_gcd() | (in module fant.polys.euclidtools), 844 | dio- dmp_LC() | (in module fant.polys.densebasic), 782 | dio- |
| dmp_from_dict() | (in module fant.polys.densebasic), 789 | dio- dmp_lcm() | (in module fant.polys.euclidtools), 846 | dio- |
| dmp_from_diofant() | (in module fant.polys.densebasic), 786 | dio- dmp_lift() | (in module fant.polys.denstools), 809 | dio- |
| dmp_gcd() | (in module fant.polys.euclidtools), 845 | dio- dmp_list_terms() | (in module fant.polys.densebasic), 792 | dio- |
| dmp_gcdex() | (in module fant.polys.euclidtools), 838 | dio- dmp_max_norm() | (in module fant.polys.densearith), 801 | dio- |
| dmp_ground() | (in module fant.polys.densebasic), 788 | dio- dmp_mul() | (in module fant.polys.densearith), 797 | dio- |
| dmp_ground_content() | (in module fant.polys.denstools), 805 | dio- dmp_mul_ground() | (in module fant.polys.densearith), 794 | dio- |
| dmp_ground_extract() | (in module fant.polys.denstools), 807 | dio- dmp_mul_term() | (in module fant.polys.densearith), 794 | dio- |
| dmp_ground_LC() | (in module fant.polys.densebasic), 783 | dio- dmp_multi_deflate() | (in module fant.polys.densebasic), 791 | dio- |

dmp_neg()	(in module fant.polys.densearith), 796	dio- dmp_slice()	(in module fant.polys.densebasic), 793	dio-
dmp_negative_p()	(in module fant.polys.densebasic), 788	dio- dmp_sqr()	(in module fant.polys.densearith), 797	dio-
dmp_nest()	(in module fant.polys.densebasic), 790	dio- dmp_strip()	(in module fant.polys.densebasic), 784	dio-
dmp_normal()	(in module fant.polys.densebasic), 785	dio- dmp_sub()	(in module fant.polys.densearith), 796	dio-
dmp_nth()	(in module fant.polys.densebasic), 786	dio- dmp_sub_ground()	(in module fant.polys.densearith), 794	dio-
dmp_one()	(in module fant.polys.densebasic), 787	dio- dmp_sub_mul()	(in module fant.polys.densearith), 797	dio-
dmp_one_p()	(in module fant.polys.densebasic), 787	dio- dmp_sub_term()	(in module fant.polys.densearith), 793	dio-
dmp_pdiv()	(in module fant.polys.densearith), 798	dio- dmp_subresultants()	(in module fant.polys.euclidtools), 841	dio-
dmp_permute()	(in module fant.polys.densebasic), 790	dio- dmp_swap()	(in module fant.polys.densebasic), 789	dio-
dmp_pexquo()	(in module fant.polys.densearith), 798	dio- dmp_TC()	(in module fant.polys.densebasic), 782	dio-
dmp_positive_p()	(in module fant.polys.densebasic), 789	dio- dmp_terms_gcd()	(in module fant.polys.densebasic), 792	dio-
dmp_pow()	(in module fant.polys.densearith), 797	dio- dmp_to_dict()	(in module fant.polys.densebasic), 789	dio-
dmp_pquo()	(in module fant.polys.densearith), 798	dio- dmp_to_tuple()	(in module fant.polys.densebasic), 785	dio-
dmp_prem()	(in module fant.polys.densearith), 798	dio- dmp_trial_division()	(in module fant.polys.factortools), 847	dio-
dmp_primitive()	(in module fant.polys.euclidtools), 846	dio- dmp_true_LT()	(in module fant.polys.densebasic), 783	dio-
dmp_primitive_prs()	(in module fant.polys.euclidtools), 839	dio- dmp_trunc()	(in module fant.polys.densetools), 804	dio-
dmp_prs_resultant()	(in module fant.polys.euclidtools), 841	dio- dmp_validate()	(in module fant.polys.densebasic), 784	dio-
dmp_qq_collins_resultant()	(in module fant.polys.euclidtools), 843	dio- dmp_zero()	(in module fant.polys.densebasic), 787	dio-
dmp_qq_heu_gcd()	(in module fant.polys.euclidtools), 845	dio- dmp_zero_p()	(in module fant.polys.densebasic), 787	dio-
dmp_quo()	(in module fant.polys.densearith), 800	dio- dmp_zeros()	(in module fant.polys.densebasic), 788	dio-
dmp_quo_ground()	(in module fant.polys.densearith), 795	dio- dmp_zz_collins_resultant()	(in module fant.polys.euclidtools), 842	dio-
dmp_raise()	(in module fant.polys.densebasic), 790	dio- dmp_zz_diophantine()	(in module fant.polys.factortools), 849	dio-
dmp_rem()	(in module fant.polys.densearith), 800	dio- dmp_zz_factor()	(in module fant.polys.factortools), 850	dio-
dmp_resultant()	(in module fant.polys.euclidtools), 843	dio- dmp_zz_heu_gcd()	(in module fant.polys.euclidtools), 844	dio-
dmp_revert()	(in module fant.polys.densetools), 810	dio- dmp_zz_mignotte_bound()	(in module fant.polys.factortools), 847	dio-
dmp_rr_div()	(in module fant.polys.densearith), 799	dio- dmp_zz_modular_resultant()	(in module fant.polys.euclidtools), 842	dio-
dmp_rr_prs_gcd()	(in module fant.polys.euclidtools), 843	dio- dmp_zz_wang()	(in module fant.polys.factortools), 849	dio-

- dmp_zz_wang_hensel_lifting() (in module diofant.polys.factortools), 849
- dmp_zz_wang_lead_coeffs() (in module diofant.polys.factortools), 849
- dmp_zz_wang_non_divisors() (in module diofant.polys.factortools), 849
- dmp_zz_wang_test_points() (in module diofant.polys.factortools), 849
- doctest_depends_on() (in module diofant.utilities.decorator), 1137
- doit() (diofant.core.basic.Atom method), 41
- doit() (diofant.core.basic.Basic method), 45
- doit() (diofant.core.function.Derivative method), 132
- doit() (diofant.core.function.Subs method), 136
- doit() (diofant.functions.elementary.piecewise.Piecewise method), 333
- doit() (diofant.integrals.integrals.Integral method), 554
- doit() (diofant.integrals.transforms.IntegralTransform method), 557
- doit() (diofant.series.limits.Limit method), 902
- doit_numerically() (diofant.core.function.Derivative method), 132
- Domain (class in diofant.polys.domains.domain), 765
- domain (diofant.polys.polytools.Poly attribute), 701
- DomainError (class in diofant.polys.polyerrors), 853
- doprint() (diofant.printing.mathematica.MCodePrinter method), 874
- doprint() (diofant.printing.mathml.MathMLPrinter method), 877
- doprint() (diofant.printing.printer.Printer method), 867
- dot() (diofant.geometry.point.Point method), 439
- dot() (diofant.matrices.matrices.MatrixBase method), 597
- dot() (diofant.vector.deloperator.Del method), 1219
- dot() (diofant.vector.dyadic.Dyadic method), 1218
- dot() (diofant.vector.vector.Vector method), 1216
- dotprint() (in module diofant.printing.dot), 884
- double_coset_can_rep() (in module diofant.combinatorics.tensor_can), 246
- drop() (diofant.polys.rings.PolyRing method), 828
- drop_to_ground() (diofant.polys.rings.PolyRing method), 828
- dsolve() (in module diofant.solvers.ode), 994
- dtype (diofant.polys.agca.modules.FreeModule attribute), 748
- dtype (diofant.polys.agca.modules.QuotientModule attribute), 756
- dtype (diofant.polys.domains.AlgebraicField attribute), 772
- dtype (diofant.polys.domains.ExpressionDomain attribute), 774
- dual() (diofant.matrices.matrices.MatrixBase method), 597
- Dummy (class in diofant.core.symbol), 82
- dummy_eq() (diofant.core.basic.Basic method), 46
- DummyWrapper (class in diofant.utilities.autowrap), 1123
- dump_c() (diofant.utilities.autowrap.UfuncifyCodeWrapper method), 1124
- dump_c() (diofant.utilities.codegen.CCodeGen method), 1131
- dump_code() (diofant.utilities.codegen.CodeGen method), 1130
- dump_f95() (diofant.utilities.codegen.FCodeGen method), 1132
- dump_h() (diofant.utilities.codegen.CCodeGen method), 1131
- dump_h() (diofant.utilities.codegen.FCodeGen method), 1132
- dump_m() (diofant.utilities.codegen.OctaveCodeGen method), 1133
- dump_pyx() (diofant.utilities.autowrap.CythonCodeWrapper method), 1123
- dup_content() (in module diofant.polys.densetools), 805
- dup_cyclotomic_p() (in module diofant.polys.factortools), 848
- dup_decompose() (in module diofant.polys.densetools), 808
- dup_extract() (in module diofant.polys.densetools), 806
- dup_gf_factor() (in module diofant.polys.factortools), 850
- dup_lshift() (in module diofant.polys.densearith), 795
- dup_mirror() (in module diofant.polys.densetools), 807
- dup_monic() (in module diofant.polys.densetools), 804

- dup_primitive() (in module `fant.polys.densertools`), 806
- dup_random() (in module `fant.polys.densebasic`), 793
- dup_real_imag() (in module `fant.polys.densertools`), 807
- dup_reverse() (in module `fant.polys.densebasic`), 785
- dup_rshift() (in module `fant.polys.densearith`), 795
- dup_scale() (in module `fant.polys.densertools`), 807
- dup_shift() (in module `fant.polys.densertools`), 808
- dup_sign_variations() (in module `fant.polys.densertools`), 809
- dup_transform() (in module `fant.polys.densertools`), 808
- dup_zz_cyclotomic_factor() (in module `fant.polys.factortools`), 848
- dup_zz_cyclotomic_poly() (in module `fant.polys.factortools`), 848
- dup_zz_factor() (in module `fant.polys.factortools`), 848
- dup_zz_factor_sqf() (in module `fant.polys.factortools`), 848
- dup_zz_hensel_lift() (in module `fant.polys.factortools`), 847
- dup_zz_hensel_step() (in module `fant.polys.factortools`), 847
- dup_zz_irreducible_p() (in module `fant.polys.factortools`), 848
- dup_zz_zassenhaus() (in module `fant.polys.factortools`), 848
- Dyadic (class in `diofant.vector.dyadic`), 1217
- E**
- E() (in module `diofant.stats`), 987
- E1() (in module `diofant.functions.special.error_functions`), 380
- EC() (`diofant.polys.polytools.Poly` method), 690
- eccentricity (`diofant.geometry.ellipse.Ellipse` attribute), 487
- edges (`diofant.combinatorics.polyhedron.Polyhedron` attribute), 217
- edges() (`diofant.combinatorics.prufer.Prufer` static method), 220
- egyptian_fraction() (in module `diofant.ntheory.egyptian_fraction`), 277
- Ei (class in `diofant.functions.special.error_functions`), 377
- eigenvals() (`diofant.matrices.matrices.MatrixBase` method), 597
- eigenvects() (`diofant.matrices.matrices.MatrixBase` method), 597
- Eijk() (in module `diofant.functions.special.tensor_functions`), 424
- eject() (`diofant.polys.polyclasses.DMP` method), 776
- eject() (`diofant.polys.polytools.Poly` method), 701
- elements (`diofant.combinatorics.perm_groups.PermutationGroup` attribute), 199
- Ellipse (class in `diofant.geometry.ellipse`), 484
- elliptic_e (class in `diofant.functions.special.elliptic_integrals`), 410
- elliptic_f (class in `diofant.functions.special.elliptic_integrals`), 410
- elliptic_k (class in `diofant.functions.special.elliptic_integrals`), 409
- elliptic_pi (class in `diofant.functions.special.elliptic_integrals`), 411
- EM() (`diofant.polys.polytools.Poly` method), 690
- emptyPrinter() (`diofant.printing.repr.ReprPrinter` method), 878
- EmptySet (class in `diofant.sets.sets`), 918
- encloses() (`diofant.geometry.entity.GeometryEntity` method), 433
- encloses_point() (`diofant.geometry.ellipse.Ellipse` method), 487
- encloses_point() (`diofant.geometry.polygon.Polygon` method), 502
- encloses_point() (`diofant.geometry.polygon.RegularPolygon` method), 508
- end (`diofant.sets.sets.Interval` attribute), 914
- enum_all() (`diofant.utilities.enumerative.MultisetPartitionTraverse` method), 1141
- enum_large() (`diofant.utilities.enumerative.MultisetPartitionTraverse` method), 1141
- enum_range() (`diofant.utilities.enumerative.MultisetPartitionTraverse` method), 1142
- enum_small() (`diofant.utilities.enumerative.MultisetPartitionTraverse` method), 1142

- method), 1142
- EPath (class in diofant.simplify.epathtools), 944
- epath() (in module diofant.simplify.epathtools), 945
- epsilon_eq() (diofant.core.numbers.Float method), 89
- Eq (in module diofant.core.relational), 110
- eq() (diofant.polys.agca.modules.QuotientModuleElement method), 758
- equal() (diofant.geometry.line.Line method), 459
- Equality (class in diofant.core.relational), 112
- equals() (diofant.core.expr.Expr method), 70
- equals() (diofant.core.relational.Relational method), 112
- equals() (diofant.geometry.line.Ray method), 462
- equals() (diofant.geometry.line3d.Line3D method), 467
- equals() (diofant.geometry.line3d.Ray3D method), 477
- equals() (diofant.geometry.point.Point method), 439
- equals() (diofant.matrices.dense.DenseMatrix method), 631
- equals() (diofant.matrices.expressions.MatrixExpr method), 656
- equals() (diofant.matrices.immutable.ImmutableMatrix class method), 653
- equation() (diofant.geometry.ellipse.Circle method), 496
- equation() (diofant.geometry.ellipse.Ellipse method), 488
- equation() (diofant.geometry.line.Line method), 459
- equation() (diofant.geometry.line3d.Line3D method), 467
- equation() (diofant.geometry.plane.Plane method), 521
- Equivalent (class in diofant.logic.boolalg), 573
- equivalent() (in module diofant.solvers.diophantine), 1090
- erf (class in diofant.functions.special.error_functions), 367
- erf2 (class in diofant.functions.special.error_functions), 371
- erf2inv (class in diofant.functions.special.error_functions), 373
- erfc (class in diofant.functions.special.error_functions), 368
- erfcinv (class in diofant.functions.special.error_functions), 373
- erfi (class in diofant.functions.special.error_functions), 370
- erfinv (class in diofant.functions.special.error_functions), 372
- Erlang() (in module diofant.stats), 966
- ET() (diofant.polys.polytools.Poly method), 690
- eta (diofant.functions.special.hyper.hyper attribute), 406
- euler (class in diofant.functions.combinatorial.numbers), 344
- euler_equations() (in module diofant.calculus.euler), 1170
- euler_maclaurin() (diofant.concrete.summations.Sum method), 282
- EulerGamma (class in diofant.core.numbers), 99
- eval() (diofant.functions.special.tensor_functions.KroneckerDelta class method), 426
- eval() (diofant.polys.polyclasses.DMP method), 776
- eval() (diofant.polys.polytools.Poly method), 701
- eval_expr() (in module diofant.parsing.sympy_parser), 1168
- eval_levicivita() (in module diofant.functions.special.tensor_functions), 425
- evalf() (diofant.core.evalf.EvalfMixin method), 147
- evalf() (diofant.core.function.Subs method), 136
- evalf() (diofant.geometry.point.Point method), 439
- evalf() (diofant.matrices.matrices.MatrixBase method), 597
- evalf() (diofant.polys.domains.domain.Domain method), 765
- EvalfMixin (class in diofant.core.evalf), 147
- evaluate() (in module diofant.core.evaluate), 58
- EvaluationFailed (class in diofant.polys.polyerrors), 853

evolute() (diofant.geometry.ellipse.Ellipse method), 488
 ExactQuotientFailed (class in diofant.polys.polyerrors), 853
 exclude() (diofant.polys.polyclasses.DMP method), 776
 exclude() (diofant.polys.polytools.Poly method), 702
 exp (diofant.core.power.Pow attribute), 103
 exp (diofant.functions.elementary.exponential attribute), 330
 exp() (diofant.matrices.matrices.MatrixBase method), 598
 exp() (in module diofant.functions.elementary.exponential), 330
 Exp1 (class in diofant.core.numbers), 97
 exp_polar (class in diofant.functions.elementary.exponential), 330
 expand() (diofant.core.expr.Expr method), 70
 expand() (diofant.matrices.matrices.MatrixBase method), 598
 expand() (in module diofant.core.function), 137
 expand_complex() (in module diofant.core.function), 144
 expand_func() (in module diofant.core.function), 144
 expand_log() (in module diofant.core.function), 143
 expand_mul() (in module diofant.core.function), 143
 expand_multinomial() (in module diofant.core.function), 145
 expand_power_base() (in module diofant.core.function), 145
 expand_power_exp() (in module diofant.core.function), 145
 expand_trig() (in module diofant.core.function), 144
 expint (class in diofant.functions.special.error_functions), 379
 Exponential() (in module diofant.stats), 967
 Expr (class in diofant.core.expr), 59
 expr (diofant.core.function.Derivative attribute), 132
 expr (diofant.core.function.Lambda attribute), 128
 expr (diofant.core.function.Subs attribute), 137
 expr (diofant.functions.elementary.piecewise.ExprCondPair attribute), 332
 ExprCondPair (class in diofant.functions.elementary.piecewise), 332
 express() (in module diofant.vector), 1224
 ExpressionDomain (class in diofant.polys.domains), 774
 ExpressionDomain.Expression (class in diofant.polys.domains), 774
 ExprWithIntLimits (class in diofant.concrete.expr_with_intlimits), 290
 ExprWithLimits (class in diofant.concrete.expr_with_limits), 288
 exquo() (diofant.polys.domains.domain.Domain method), 765
 exquo() (diofant.polys.domains.field.Field method), 768
 exquo() (diofant.polys.domains.ring.Ring method), 769
 exquo() (diofant.polys.polyclasses.DMF method), 781
 exquo() (diofant.polys.polyclasses.DMP method), 777
 exquo() (diofant.polys.polytools.Poly method), 702
 exquo() (in module diofant.polys.polytools), 676
 exquo_ground() (diofant.polys.polyclasses.DMP method), 777
 exquo_ground() (diofant.polys.polytools.Poly method), 702
 extend() (diofant.ntheory.generate.Sieve method), 250
 extend() (diofant.plotting.plot.Plot method), 888
 extend_to_no() (diofant.ntheory.generate.Sieve method), 250
 extended_euclid() (in module diofant.solvers.diophantine), 1089
 exterior_angle (diofant.geometry.polygon.RegularPolygon attribute), 509
 extract() (diofant.matrices.matrices.MatrixBase method), 598
 extract() (diofant.matrices.sparse.SparseMatrixBase method), 646
 extract_additively() (diofant.core.expr.Expr method), 71
 extract_branch_factor() (diofant.core.expr.Expr method), 71

- extract_leading_order() (diofant.core.add.Add method), 108
- extract_multiplicatively() (diofant.core.expr.Expr method), 72
- ExtraneousFactors (class in diofant.polys.polyerrors), 853
- eye() (diofant.matrices.dense.DenseMatrix class method), 632
- eye() (diofant.matrices.sparse.SparseMatrixBase class method), 646
- eye() (in module diofant.matrices.dense), 622
- ## F
- F2PyCodeWrapper (class in diofant.utilities.autowrap), 1124
- faces (diofant.combinatorics.polyhedron.Polyhedron attribute), 218
- factor() (diofant.core.expr.Expr method), 72
- factor() (in module diofant.polys.polytools), 685
- factor_list() (diofant.polys.polyclasses.DMP method), 777
- factor_list() (diofant.polys.polytools.Poly method), 702
- factor_list() (in module diofant.polys.polytools), 685
- factor_list_include() (diofant.polys.polyclasses.DMP method), 777
- factor_list_include() (diofant.polys.polytools.Poly method), 703
- factor_terms() (in module diofant.core.exprtools), 157
- factorial (class in diofant.functions.combinatorial.factorials), 345
- factorial() (diofant.polys.domains.FractionField method), 773
- factorial() (diofant.polys.domains.PolynomialRing method), 771
- factorial2 (class in diofant.functions.combinatorial.factorials), 347
- factoring_visitor() (in module diofant.utilities.enumerative), 1139
- factorint() (in module diofant.ntheory.factor_), 261
- factors() (diofant.core.numbers.Rational method), 91
- FallingFactorial (class in diofant.functions.combinatorial.factorials), 347
- fcode() (in module diofant.printing.fcode), 870
- FCodeGen (class in diofant.utilities.codegen), 1132
- FCodePrinter (class in diofant.printing.fcode), 872
- fdiff() (diofant.core.function.Function method), 135
- fdiff() (diofant.functions.elementary.complexes.Abs method), 309
- fdiff() (diofant.functions.elementary.exponential.LambertW method), 331
- fdiff() (diofant.functions.elementary.exponential.log method), 332
- fdiff() (diofant.functions.elementary.hyperbolic.csch method), 327
- fdiff() (diofant.functions.elementary.hyperbolic.sinh method), 325
- FDistribution() (in module diofant.stats), 968
- fglm() (diofant.polys.polytools.GroebnerBasis method), 728
- fibonacci (class in diofant.functions.combinatorial.numbers), 348
- Field (class in diofant.polys.domains.field), 768
- field_isomorphism() (in module diofant.polys.numberfields), 732
- fill() (diofant.matrices.dense.MutableDenseMatrix method), 635
- fill() (diofant.matrices.sparse.MutableSparseMatrix method), 640
- filldedent() (in module diofant.utilities.misc), 1166
- filter_symbols() (in module diofant.utilities.iterables), 1145
- find() (diofant.core.basic.Basic method), 46
- find_DN() (in module diofant.solvers.diophantine), 1083
- findrecur() (diofant.concrete.summations.Sum method), 283
- finite_diff_weights() (in module diofant.calculus.finite_diff), 1175
- FiniteDomain (class in diofant.stats.frv), 991
- FiniteField (class in diofant.polys.domains), 770
- FinitePSpace (class in diofant.stats.frv), 991
- FiniteRV() (in module diofant.stats), 958
- FiniteSet (class in diofant.sets.sets), 915
- FisherZ() (in module diofant.stats), 969
- FlagError (class in diofant.polys.polyerrors), 853
- flatten() (diofant.core.add.Add class method), 108

flatten() (diofant.core.mul.Mul class method), 105

flatten() (in module diofant.utilities.iterables), 1146

Float (class in diofant.core.numbers), 86

floor (class in diofant.functions.elementary.integers), 329

floor() (diofant.core.numbers.Float method), 89

foci (diofant.geometry.ellipse.Ellipse attribute), 488

focus_distance (diofant.geometry.ellipse.Ellipse attribute), 489

fourier_transform() (in module diofant.integrals.transforms), 528

FourierTransform (class in diofant.integrals.transforms), 558

frac (in module diofant.printing.pretty.pretty_symbology), 882

frac_field() (diofant.polys.domains.domain.Domain method), 765

frac_unify() (diofant.polys.polyclasses.DMF method), 781

fraction() (in module diofant.simplify.radsimp), 934

FractionField (class in diofant.polys.domains), 773

Frechet() (in module diofant.stats), 970

free_module() (diofant.polys.domains.ring.Ring method), 769

free_symbols (diofant.concrete.expr_with_limits.ExprWithLimits attribute), 289

free_symbols (diofant.core.basic.Basic attribute), 46

free_symbols (diofant.core.function.Derivative attribute), 132

free_symbols (diofant.core.function.Lambda attribute), 128

free_symbols (diofant.core.function.Subs attribute), 137

free_symbols (diofant.functions.elementary.piecewise.Piecewise attribute), 332

free_symbols (diofant.geometry.curve.Curve attribute), 482

free_symbols (diofant.integrals.integrals.Integral attribute), 555

free_symbols (diofant.integrals.transforms.IntegralTransform attribute), 557

free_symbols (diofant.matrices.matrices.MatrixBase attribute), 598

free_symbols (diofant.polys.polytools.Poly attribute), 703

free_symbols (diofant.polys.polytools.PurePoly attribute), 727

free_symbols_in_domain (diofant.polys.polytools.Poly attribute), 703

FreeModule (class in diofant.polys.agca.modules), 748

FreeModuleElement (class in diofant.polys.agca.modules), 753

fresnelc (class in diofant.functions.special.error_functions), 376

FresnelIntegral (class in diofant.functions.special.error_functions), 374

fresnels (class in diofant.functions.special.error_functions), 374

from_AlgebraicField() (diofant.polys.domains.domain.Domain method), 765

from_AlgebraicField() (diofant.polys.domains.FractionField method), 773

from_AlgebraicField() (diofant.polys.domains.IntegerRing method), 771

from_AlgebraicField() (diofant.polys.domains.PolynomialRing method), 771

from_AlgebraicField() (diofant.polys.domains.RationalField method), 772

from_ComplexField() (diofant.polys.domains.domain.Domain method), 766

from_dict() (diofant.polys.polyclasses.DMP class method), 777

from_dict() (diofant.polys.polytools.Poly class method), 704

from_diofant() (diofant.polys.domains.AlgebraicField method), 772

from_diofant() (diofant.polys.domains.domain.Domain method), 766

from_diofant() (diofant.polys.domains.ExpressionDomain method), 774

from_diofant() (diofant.polys.domains.FiniteField method), 770

<p>from_diofant() (diofant.polys.domains.FractionField method), 773</p> <p>from_diofant() (diofant.polys.domains.PolynomialRing method), 771</p> <p>from_diofant() (diofant.polys.domains.RealField method), 774</p> <p>from_diofant_list() (diofant.polys.polyclasses.DMP class method), 777</p> <p>from_expr() (diofant.polys.polytools.Poly class method), 704</p> <p>from_ExpressionDomain() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_ExpressionDomain() (diofant.polys.domains.ExpressionDomain method), 774</p> <p>from_FF_gmpy() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_FF_gmpy() (diofant.polys.domains.FiniteField method), 770</p> <p>from_FF_python() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_FF_python() (diofant.polys.domains.FiniteField method), 770</p> <p>from_FractionField() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_FractionField() (diofant.polys.domains.ExpressionDomain method), 774</p> <p>from_FractionField() (diofant.polys.domains.FractionField method), 773</p> <p>from_FractionField() (diofant.polys.domains.PolynomialRing method), 771</p> <p>from_GlobalPolynomialRing() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_inversion_vector() (diofant.combinatorics.permutations.Permutation class method), 170</p> <p>from_list() (diofant.polys.polyclasses.DMP class method), 777</p> <p>from_list() (diofant.polys.polytools.Poly class method), 704</p>	<p>from_poly() (diofant.polys.polytools.Poly class method), 704</p> <p>from_PolynomialRing() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_PolynomialRing() (diofant.polys.domains.ExpressionDomain method), 774</p> <p>from_PolynomialRing() (diofant.polys.domains.FractionField method), 773</p> <p>from_PolynomialRing() (diofant.polys.domains.PolynomialRing method), 771</p> <p>from_QQ_gmpy() (diofant.polys.domains.AlgebraicField method), 772</p> <p>from_QQ_gmpy() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_QQ_gmpy() (diofant.polys.domains.ExpressionDomain method), 774</p> <p>from_QQ_gmpy() (diofant.polys.domains.FiniteField method), 770</p> <p>from_QQ_gmpy() (diofant.polys.domains.FractionField method), 773</p> <p>from_QQ_gmpy() (diofant.polys.domains.PolynomialRing method), 771</p> <p>from_QQ_python() (diofant.polys.domains.AlgebraicField method), 772</p> <p>from_QQ_python() (diofant.polys.domains.domain.Domain method), 766</p> <p>from_QQ_python() (diofant.polys.domains.ExpressionDomain method), 774</p> <p>from_QQ_python() (diofant.polys.domains.FiniteField method), 770</p> <p>from_QQ_python() (diofant.polys.domains.FractionField method), 773</p> <p>from_QQ_python() (diofant.polys.domains.PolynomialRing method), 771</p> <p>from_RealField() (diofant.polys.domains.AlgebraicField method), 772</p>
--	--

[from_RealField\(\)](#) (diofant.polys.domains.domain.Domain method), 766
[from_RealField\(\)](#) (diofant.polys.domains.ExpressionDomain method), 774
[from_RealField\(\)](#) (diofant.polys.domains.FiniteField method), 770
[from_RealField\(\)](#) (diofant.polys.domains.FractionField method), 773
[from_RealField\(\)](#) (diofant.polys.domains.PolynomialRing method), 771
[from_rgs\(\)](#) (diofant.combinatorics.partitions.Partition class method), 158
[from_sequence\(\)](#) (diofant.combinatorics.permutations.Permutation class method), 171
[from_TIDS_list\(\)](#) (diofant.tensor.tensor.TensAdd static method), 1117
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.AlgebraicField method), 772
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.domain.Domain method), 766
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.ExpressionDomain method), 774
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.FiniteField method), 770
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.FractionField method), 773
[from_ZZ_gmpy\(\)](#) (diofant.polys.domains.PolynomialRing method), 771
[from_ZZ_python\(\)](#) (diofant.polys.domains.AlgebraicField method), 772
[from_ZZ_python\(\)](#) (diofant.polys.domains.domain.Domain method), 766
[from_ZZ_python\(\)](#) (diofant.polys.domains.ExpressionDomain method), 774
[from_ZZ_python\(\)](#) (diofant.polys.domains.FiniteField method), 770
[from_ZZ_python\(\)](#) (diofant.polys.domains.FractionField method), 773
[from_ZZ_python\(\)](#) (diofant.polys.domains.PolynomialRing method), 771
[full_cyclic_form](#) (diofant.combinatorics.permutations.Permutation attribute), 171
[fun_eval\(\)](#) (diofant.tensor.tensor.TensAdd method), 1118
[fun_eval\(\)](#) (diofant.tensor.tensor.TensMul method), 1120
[func](#) (diofant.core.basic.Basic attribute), 46
[func_field_modgcd\(\)](#) (in module diofant.polys.modulargcd), 857
[Function](#) (class in diofant.core.function), 134
[function](#) (diofant.concrete.expr_with_limits.ExprWithLimits attribute), 289
[function](#) (diofant.integrals.transforms.IntegralTransform attribute), 557
[function_exponentiation\(\)](#) (in module diofant.parsing.sympy_parser), 1169
[function_variable](#) (diofant.integrals.transforms.IntegralTransform attribute), 557
[FunctionClass](#) (class in diofant.core.function), 133
[FunctionMatrix](#) (class in diofant.matrices.expressions), 658
[functions](#) (diofant.geometry.curve.Curve attribute), 482
G
[G\(\)](#) (in module diofant.printing.pretty.pretty_symbology), 881
[g\(\)](#) (in module diofant.printing.pretty.pretty_symbology), 881
[gamma](#) (class in diofant.functions.special.gamma_functions), 358
[Gamma\(\)](#) (in module diofant.stats), 970
[GammaInverse\(\)](#) (in module diofant.stats), 971
[gauss_chebyshev_t\(\)](#) (in module diofant.integrals.quadrature), 563
[gauss_chebyshev_u\(\)](#) (in module diofant.integrals.quadrature), 563
[gauss_gen_laguerre\(\)](#) (in module diofant.integrals.quadrature), 562
[gauss_hermite\(\)](#) (in module diofant.integrals.quadrature), 561

- gauss_jacobi() (in module diofant.integrals.quadrature), 564
- gauss_laguerre() (in module diofant.integrals.quadrature), 560
- gauss_legendre() (in module diofant.integrals.quadrature), 559
- gaussian_reduce() (in module diofant.solvers.diophantine), 1092
- gcd() (diofant.core.numbers.Number method), 86
- gcd() (diofant.core.numbers.Rational method), 91
- gcd() (diofant.polys.domains.domain.Domain method), 766
- gcd() (diofant.polys.domains.field.Field method), 768
- gcd() (diofant.polys.domains.PolynomialRing method), 771
- gcd() (diofant.polys.domains.RealField method), 774
- gcd() (diofant.polys.polyclasses.DMP method), 777
- gcd() (diofant.polys.polytools.Poly method), 704
- gcd() (in module diofant.polys.polytools), 681
- gcd_list() (in module diofant.polys.polytools), 681
- gcd_terms() (in module diofant.core.exprtools), 156
- gcdex() (diofant.polys.domains.domain.Domain method), 766
- gcdex() (diofant.polys.domains.PolynomialRing method), 771
- gcdex() (diofant.polys.polyclasses.DMP method), 777
- gcdex() (diofant.polys.polytools.Poly method), 704
- gcdex() (in module diofant.polys.polytools), 676
- Ge (in module diofant.core.relational), 111
- gegenbauer (class in diofant.functions.special.polynomials), 414
- gegenbauer_poly() (in module diofant.polys.orthopolys), 736
- gen (diofant.polys.polytools.Poly attribute), 704
- generate() (diofant.combinatorics.perm_groups.PermGroup method), 200
- generate_bell() (in module diofant.utilities.iterables), 1146
- generate_derangements() (in module diofant.utilities.iterables), 1147
- generate_dimino() (diofant.combinatorics.perm_groups.PermutationGroup method), 200
- generate_gray() (diofant.combinatorics.graycode.GrayCode method), 231
- generate_involutions() (in module diofant.utilities.iterables), 1148
- generate_oriented_forest() (in module diofant.utilities.iterables), 1148
- generate_schreier_sims() (diofant.combinatorics.perm_groups.PermutationGroup method), 201
- generators (diofant.combinatorics.perm_groups.PermutationGroup attribute), 201
- GeneratorsError (class in diofant.polys.polyerrors), 853
- GeneratorsNeeded (class in diofant.polys.polyerrors), 853
- Geometric() (in module diofant.stats), 959
- GeometryEntity (class in diofant.geometry.entity), 433
- get() (diofant.core.containers.Dict method), 149
- get_adjacency_distance() (diofant.combinatorics.permutations.Permutation method), 171
- get_adjacency_matrix() (diofant.combinatorics.permutations.Permutation method), 172
- get_comm() (diofant.tensor.tensor._TensorManager method), 1106
- get_contraction_structure() (in module diofant.tensor.index_methods), 1103
- get_default_datatype() (in module diofant.utilities.codegen), 1129
- get_diag_blocks() (diofant.matrices.matrices.MatrixBase method), 599
- get_domain() (diofant.polys.polytools.Poly method), 705
- get_exact() (diofant.polys.domains.domain.Domain method), 766
- get_exact() (diofant.polys.domains.RealField method), 774
- get_field() (diofant.polys.domains.domain.Domain method), 766
- get_field() (diofant.polys.domains.ExpressionDomain method), 774
- get_field() (diofant.polys.domains.field.Field method), 768
- get_field() (diofant.polys.domains.FiniteField method), 770

[get_field\(\) \(diofant.polys.domains.IntegerRing method\), 771](#)
[get_field\(\) \(diofant.polys.domains.PolynomialRing method\), 771](#)
[get_indices\(\) \(diofant.tensor.tensor.TensMul method\), 1120](#)
[get_indices\(\) \(in module diofant.tensor.index_methods\), 1105](#)
[get_interface\(\) \(diofant.utilities.codegen.FCodeGen method\), 1132](#)
[get_matrix\(\) \(diofant.tensor.tensor.TensExpr method\), 1115](#)
[get_modulus\(\) \(diofant.polys.polytools.Poly method\), 705](#)
[get_period\(\) \(diofant.functions.special.hyper.meijerg method\), 409](#)
[get_positional_distance\(\) \(diofant.combinatorics.permutations.Permutation method\), 172](#)
[get_precedence_distance\(\) \(diofant.combinatorics.permutations.Permutation method\), 172](#)
[get_precedence_matrix\(\) \(diofant.combinatorics.permutations.Permutation method\), 173](#)
[get_prototype\(\) \(diofant.utilities.codegen.CCodeGen method\), 1131](#)
[get_ring\(\) \(diofant.polys.domains.AlgebraicField method\), 772](#)
[get_ring\(\) \(diofant.polys.domains.domain.Domain method\), 766](#)
[get_ring\(\) \(diofant.polys.domains.ExpressionDomain method\), 775](#)
[get_ring\(\) \(diofant.polys.domains.field.Field method\), 768](#)
[get_ring\(\) \(diofant.polys.domains.FractionField method\), 773](#)
[get_ring\(\) \(diofant.polys.domains.RealField method\), 774](#)
[get_ring\(\) \(diofant.polys.domains.ring.Ring method\), 769](#)
[get_segments\(\) \(diofant.plotting.plot.LineOver1DRangeSeries method\), 899](#)
[get_segments\(\) \(diofant.plotting.plot.Parametric2DLineSeries method\), 899](#)
[get_subset_from_bitstring\(\) \(diofant.combinatorics.graycode method\), 234](#)
[get_symmetric_group_sgs\(\) \(in module diofant.combinatorics.tensor_can\), 249](#)
[getn\(\) \(diofant.core.expr.Expr method\), 72](#)
[getO\(\) \(diofant.core.add.Add method\), 109](#)
[getO\(\) \(diofant.core.expr.Expr method\), 72](#)
[gf_add\(\) \(in module diofant.polys.galoistools\), 814](#)
[gf_add_ground\(\) \(in module diofant.polys.galoistools\), 813](#)
[gf_add_mul\(\) \(in module diofant.polys.galoistools\), 815](#)
[gf_berlekamp\(\) \(in module diofant.polys.galoistools\), 823](#)
[gf_cofactors\(\) \(in module diofant.polys.galoistools\), 818](#)
[gf_compose\(\) \(in module diofant.polys.galoistools\), 820](#)
[gf_compose_mod\(\) \(in module diofant.polys.galoistools\), 820](#)
[gfCRT\(\) \(in module diofant.polys.galoistools\), 810](#)
[gfCRT1\(\) \(in module diofant.polys.galoistools\), 811](#)
[gfCRT2\(\) \(in module diofant.polys.galoistools\), 811](#)
[gf_csolve\(\) \(in module diofant.polys.galoistools\), 825](#)
[gf_degree\(\) \(in module diofant.polys.galoistools\), 811](#)
[gf_diff\(\) \(in module diofant.polys.galoistools\), 819](#)
[gf_div\(\) \(in module diofant.polys.galoistools\), 816](#)
[gf_eval\(\) \(in module diofant.polys.galoistools\), 820](#)
[gf_expand\(\) \(in module diofant.polys.galoistools\), 816](#)
[gf_exquo\(\) \(in module diofant.polys.galoistools\), 817](#)
[gf_factor\(\) \(in module diofant.polys.galoistools\), 824](#)
[gf_factor_sqf\(\) \(in module diofant.polys.galoistools\), 824](#)
[gf_from_dict\(\) \(in module diofant.polys.galoistools\), 812](#)
[gf_from_int_poly\(\) \(in module diofant.polys.galoistools\), 813](#)
[gf_gcd\(\) \(in module diofant.polys.galoistools\), 818](#)
[gf_gcdex\(\) \(in module diofant.polys.galoistools\), 819](#)
[gf_int\(\) \(in module diofant.polys.galoistools\), 811](#)
[gf_irreducible\(\) \(in module diofant.polys.galoistools\), 821](#)

`gf_irreducible_p()` (in module `diofant.polys.galoistools`), 821
`gf_LC()` (in module `diofant.polys.galoistools`), 811
`gf_lcm()` (in module `diofant.polys.galoistools`), 818
`gf_lshift()` (in module `diofant.polys.galoistools`), 817
`gf_monic()` (in module `diofant.polys.galoistools`), 819
`gf_mul()` (in module `diofant.polys.galoistools`), 815
`gf_mul_ground()` (in module `diofant.polys.galoistools`), 814
`gf_multi_eval()` (in module `diofant.polys.galoistools`), 820
`gf_neg()` (in module `diofant.polys.galoistools`), 813
`gf_normal()` (in module `diofant.polys.galoistools`), 812
`gf_pow()` (in module `diofant.polys.galoistools`), 817
`gf_pow_mod()` (in module `diofant.polys.galoistools`), 818
`gf_Qbasis()` (in module `diofant.polys.galoistools`), 823
`gf_Qmatrix()` (in module `diofant.polys.galoistools`), 823
`gf_quo()` (in module `diofant.polys.galoistools`), 816
`gf_quo_ground()` (in module `diofant.polys.galoistools`), 814
`gf_random()` (in module `diofant.polys.galoistools`), 821
`gf_rem()` (in module `diofant.polys.galoistools`), 816
`gf_rshift()` (in module `diofant.polys.galoistools`), 817
`gf_shoup()` (in module `diofant.polys.galoistools`), 824
`gf_sqf_list()` (in module `diofant.polys.galoistools`), 822
`gf_sqf_p()` (in module `diofant.polys.galoistools`), 822
`gf_sqf_part()` (in module `diofant.polys.galoistools`), 822
`gf_sqr()` (in module `diofant.polys.galoistools`), 815
`gf_strip()` (in module `diofant.polys.galoistools`), 812
`gf_sub()` (in module `diofant.polys.galoistools`), 815
`gf_sub_ground()` (in module `diofant.polys.galoistools`), 814
`gf_sub_mul()` (in module `diofant.polys.galoistools`), 815
`gf_TC()` (in module `diofant.polys.galoistools`), 812
`gf_to_dict()` (in module `diofant.polys.galoistools`), 813
`gf_to_int_poly()` (in module `diofant.polys.galoistools`), 813
`gf_trace_map()` (in module `diofant.polys.galoistools`), 820
`gf_trunc()` (in module `diofant.polys.galoistools`), 812
`gf_value()` (in module `diofant.polys.galoistools`), 825
`gf_zassenhaus()` (in module `diofant.polys.galoistools`), 824
`gff()` (in module `diofant.polys.polytools`), 684
`gff_list()` (`diofant.polys.polyclasses.DMP` method), 777
`gff_list()` (`diofant.polys.polytools.Poly` method), 705
`gff_list()` (in module `diofant.polys.polytools`), 684
`given()` (in module `diofant.stats`), 988
`GMPYFiniteField` (class in `diofant.polys.domains`), 775
`GMPYIntegerRing` (class in `diofant.polys.domains`), 775
`GMPYRationalField` (class in `diofant.polys.domains`), 775
`GoldenRatio` (class in `diofant.core.numbers`), 100
`gospers_normal()` (in module `diofant.concrete.gospers`), 294
`gospers_sum()` (in module `diofant.concrete.gospers`), 295
`gospers_term()` (in module `diofant.concrete.gospers`), 295
`GradedLexOrder` (class in `diofant.polys.orderings`), 734
`gradient()` (`diofant.vector.deloperator.Del` method), 1220
`gradient()` (in module `diofant.vector`), 1226
`GramSchmidt()` (in module `diofant.matrices.dense`), 625
`gray_to_bin()` (`diofant.combinatorics.graycode` method), 233
`GrayCode` (class in `diofant.combinatorics.graycode`), 230
`graycode_subsets()` (`diofant.combinatorics.graycode` method), 234

GreaterThan (class in diofant.core.relational), 113
 greek_letters (in module diofant.printing.pretty.pretty_symbology), 881
 groebner() (in module diofant.polys.groebnertools), 850
 groebner() (in module diofant.polys.polytools), 689
 GroebnerBasis (class in diofant.polys.polytools), 727
 ground_roots() (diofant.polys.polytools.Poly method), 705
 ground_roots() (in module diofant.polys.polytools), 688
 group() (in module diofant.utilities.iterables), 1148
 Gt (in module diofant.core.relational), 111

H

H (diofant.matrices.matrices.MatrixBase attribute), 587
 Half (class in diofant.core.numbers), 95
 half_gcdex() (diofant.polys.domains.domain.Domain method), 766
 half_gcdex() (diofant.polys.polyclasses.DMP method), 777
 half_gcdex() (diofant.polys.polytools.Poly method), 705
 half_gcdex() (in module diofant.polys.polytools), 676
 half_per() (diofant.polys.polyclasses.DMF method), 781
 hankel1 (class in diofant.functions.special.bessel), 390
 hankel2 (class in diofant.functions.special.bessel), 391
 hankel_transform() (in module diofant.integrals.transforms), 531
 HankelTransform (class in diofant.integrals.transforms), 558
 harmonic (class in diofant.functions.combinatorial.numbers), 349
 has() (diofant.core.basic.Basic method), 47
 has() (diofant.matrices.matrices.MatrixBase method), 599
 has() (diofant.matrices.sparse.SparseMatrixBase method), 646
 has_dups() (in module diofant.utilities.iterables), 1149
 has_only_gens() (diofant.polys.polytools.Poly method), 706
 has_variety() (in module diofant.utilities.iterables), 1149
 Heaviside (class in diofant.functions.special.delta_functions), 358
 height() (diofant.polys.agca.ideals.Ideal method), 754
 height() (diofant.printing.pretty.stringpict.stringPict method), 883
 hermite (class in diofant.functions.special.polynomials), 419
 hermite_poly() (in module diofant.polys.orthopolys), 736
 hessian() (in module diofant.matrices.dense), 624
 heurisch() (in module diofant.integrals.heurisch), 551
 heurisch_wrapper() (in module diofant.integrals.heurisch), 552
 HeuristicGCDFailed (class in diofant.polys.polyerrors), 853
 hobj() (in module diofant.printing.pretty.pretty_symbology), 882
 holzer() (in module diofant.solvers.diophantine), 1092
 homogeneous_order() (diofant.polys.polyclasses.DMP method), 777
 homogeneous_order() (diofant.polys.polytools.Poly method), 706
 homogeneous_order() (in module diofant.solvers.ode), 999
 homogenize() (diofant.polys.polyclasses.DMP method), 777
 homogenize() (diofant.polys.polytools.Poly method), 706
 homomorphism() (in module diofant.polys.agca.homomorphisms), 759
 HomomorphismFailed (class in diofant.polys.polyerrors), 853
 horner() (in module diofant.polys.polyfuncs), 730
 hradius (diofant.geometry.ellipse.Ellipse attribute), 489
 hstack() (diofant.matrices.matrices.MatrixBase class method), 599
 hyper (class in diofant.functions.special.hyper), 404
 HyperbolicFunction (class in diofant.functions.elementary.hyperbolic),

- 325
- hyperexpand() (in module diofant.simplify.hyperexpand), 943
- Hypergeometric() (in module diofant.stats), 958
- hypersimilar() (in module diofant.simplify.simplify), 926
- hypersimp() (in module diofant.simplify.simplify), 926
- I
- ibin() (in module diofant.utilities.iterables), 1149
- Ideal (class in diofant.polys.agca.ideals), 753
- ideal() (diofant.polys.domains.ring.Ring method), 769
- Identity (class in diofant.matrices.expressions), 658
- identity_hom() (diofant.polys.agca.modules.FreeModule method), 748
- identity_hom() (diofant.polys.agca.modules.Module method), 747
- identity_hom() (diofant.polys.agca.modules.QuotientModule method), 756
- identity_hom() (diofant.polys.agca.modules.SubModule method), 750
- IdentityFunction (class in diofant.functions.elementary.miscellaneous), 333
- idiff() (in module diofant.geometry.util), 435
- Idx (class in diofant.tensor.indexed), 1098
- igcd() (in module diofant.core.numbers), 93
- ilcm() (in module diofant.core.numbers), 93
- im (class in diofant.functions.elementary.complexes), 307
- image() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 761
- ImageSet (class in diofant.sets.fancysets), 921
- imageset() (in module diofant.sets.sets), 912
- ImaginaryUnit (class in diofant.core.numbers), 98
- ImmutableMatrix (class in diofant.matrices.immutable), 653
- ImmutableSparseMatrix (class in diofant.matrices.immutable), 651
- implemented_function() (in module diofant.utilities.lambdify), 1162
- implicit_application() (in module diofant.parsing.sympy_parser), 1169
- implicit_multiplication() (in module diofant.parsing.sympy_parser), 1169
- implicit_multiplication_application() (in module diofant.parsing.sympy_parser), 1170
- ImplicitSeries (class in diofant.plotting.plot_implicit), 900
- Implies (class in diofant.logic.boolalg), 572
- imul_num() (diofant.polys.rings.PolyElement method), 830
- in_terms_of_generators() (diofant.polys.agca.modules.SubModule method), 750
- incenter (diofant.geometry.polygon.Triangle attribute), 515
- incircle (diofant.geometry.polygon.RegularPolygon attribute), 509
- incircle (diofant.geometry.polygon.Triangle attribute), 515
- inclusion_hom() (diofant.polys.agca.modules.SubModule method), 750
- indent_code() (diofant.printing.ccode.CCodePrinter method), 868
- indent_code() (diofant.printing.fcode.FCodePrinter method), 872
- independent() (in module diofant.stats.rv), 993
- index() (diofant.combinatorics.permutations.Permutation method), 173
- index() (diofant.concrete.expr_with_intlimits.ExprWithIntLimits method), 291
- index() (diofant.core.containers.Tuple method), 148
- index() (diofant.polys.rings.PolyRing method), 828
- Indexed (class in diofant.tensor.indexed), 1100
- IndexedBase (class in diofant.tensor.indexed), 1101
- indices (diofant.tensor.indexed.Indexed attribute), 1100
- indices_contain_equal_information (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 426
- inf (diofant.sets.sets.Set attribute), 909
- infinitesimals() (in module diofant.solvers.ode), 1000
- Infinity (class in diofant.core.numbers), 96

[init_printing\(\)](#) (in module `diofant.interactive.printing`), 885
[inject\(\)](#) (`diofant.polys.domains.compositedomain.CompositeDomain` method), 770
[inject\(\)](#) (`diofant.polys.domains.domain.Domain` method), 766
[inject\(\)](#) (`diofant.polys.domains.simpledomain.SimpleDomain` method), 770
[inject\(\)](#) (`diofant.polys.polyclasses.DMP` method), 777
[inject\(\)](#) (`diofant.polys.polytools.Poly` method), 707
[inradius](#) (`diofant.geometry.polygon.RegularPolygon` attribute), 509
[inradius](#) (`diofant.geometry.polygon.Triangle` attribute), 516
[intcurve_diffequ\(\)](#) (in module `diofant.diffgeom`), 1188
[intcurve_series\(\)](#) (in module `diofant.diffgeom`), 1187
[Integer](#) (class in `diofant.core.numbers`), 91
[integer_nthroot\(\)](#) (in module `diofant.core.power`), 103
[IntegerPartition](#) (class in `diofant.combinatorics.partitions`), 159
[IntegerRing](#) (class in `diofant.polys.domains`), 771
[Integers](#) (class in `diofant.sets.fancysets`), 920
[IntegerWrapper](#) (class in `diofant.interactive.session`), 886
[Integral](#) (class in `diofant.integrals.integrals`), 553
[Integral.is_commutative](#) (in module `diofant.integrals.transforms`), 553
[IntegralTransform](#) (class in `diofant.integrals.transforms`), 557
[integrand\(\)](#) (`diofant.functions.special.hyper.meijer` method), 409
[integrate\(\)](#) (`diofant.core.expr.Expr` method), 72
[integrate\(\)](#) (`diofant.matrices.matrices.MatrixBase` method), 600
[integrate\(\)](#) (`diofant.polys.polyclasses.DMP` method), 777
[integrate\(\)](#) (`diofant.polys.polytools.Poly` method), 707
[integrate\(\)](#) (in module `diofant.integrals.integrals`), 546
[interactive_traversal\(\)](#) (in module `diofant.utilities.iterables`), 1150
[interior_angle](#) (`diofant.geometry.polygon.RegularPolygon` attribute), 510
[interpolate\(\)](#) (in module `diofant.polys.polyfuncs`), 730
[inCompleteOnly\(\)](#) (in module `diofant.polys.specialpolys`), 735
[intersect\(\)](#) (`diofant.polys.agca.ideals.Ideal` method), 754
[intersect\(\)](#) (`diofant.polys.agca.modules.SubModule` method), 750
[intersect\(\)](#) (`diofant.sets.sets.Set` method), 909
[Intersection](#) (class in `diofant.sets.sets`), 916
[intersection\(\)](#) (`diofant.geometry.ellipse.Circle` method), 497
[intersection\(\)](#) (`diofant.geometry.ellipse.Ellipse` method), 489
[intersection\(\)](#) (`diofant.geometry.entity.GeometryEntity` method), 433
[intersection\(\)](#) (`diofant.geometry.line.LinearEntity` method), 452
[intersection\(\)](#) (`diofant.geometry.line3d.LinearEntity3D` method), 471
[intersection\(\)](#) (`diofant.geometry.plane.Plane` method), 521
[intersection\(\)](#) (`diofant.geometry.point.Point` method), 440
[intersection\(\)](#) (`diofant.geometry.point.Point3D` method), 447
[intersection\(\)](#) (`diofant.geometry.polygon.Polygon` method), 502
[intersection\(\)](#) (`diofant.sets.sets.Set` method), 910
[intersection\(\)](#) (in module `diofant.geometry.util`), 435
[Interval](#) (class in `diofant.sets.sets`), 913
[intervals\(\)](#) (`diofant.polys.polyclasses.DMP` method), 777
[intervals\(\)](#) (`diofant.polys.polytools.Poly` method), 707
[intervals\(\)](#) (in module `diofant.polys.polytools`), 686
[inv\(\)](#) (`diofant.matrices.matrices.MatrixBase` method), 600
[inv_mod\(\)](#) (`diofant.matrices.matrices.MatrixBase` method), 600
[Inverse](#) (class in `diofant.matrices.expressions`), 657

inverse() (diofant.functions.elementary.exponential.inverse.CopulaTransform (class in diofant.integrals.transforms), 558
 method), 330

inverse() (diofant.functions.elementary.exponential.inverse.FourierTransform (class in diofant.integrals.transforms), 558
 method), 332

inverse() (diofant.functions.elementary.hyperbolic.inverse.HankelTransform (class in diofant.integrals.transforms), 559
 method), 328

inverse() (diofant.functions.elementary.hyperbolic.inverse.LaplaceTransform (class in diofant.integrals.transforms), 558
 method), 328

inverse() (diofant.functions.elementary.hyperbolic.inverse.MellinTransform (class in diofant.integrals.transforms), 557
 method), 327

inverse() (diofant.functions.elementary.hyperbolic.inverse.SineTransform (class in diofant.integrals.transforms), 558
 method), 328

inverse() (diofant.functions.elementary.hyperbolic.inverse.inversion_vector() (diofant.combinatorics.permutations.Permutation
 method), 326

inverse() (diofant.functions.elementary.hyperbolic.sinh method), 174
 method), 325

inverse() (diofant.functions.elementary.hyperbolic.tanh method), 174
 method), 326

inverse() (diofant.functions.elementary.trigonometric.acos method), 766
 method), 319

inverse() (diofant.functions.elementary.trigonometric.acos method), 769
 method), 321

inverse() (diofant.functions.elementary.trigonometric.acos method), 781
 method), 323

inverse() (diofant.functions.elementary.trigonometric.asin method), 777
 method), 320, 322

inverse() (diofant.functions.elementary.trigonometric.asin method), 778
 method), 318

inverse() (diofant.functions.elementary.trigonometric.atan method), 776
 method), 321

inverse() (diofant.functions.elementary.trigonometric.cot attribute), 201
 method), 316

inverse() (diofant.functions.elementary.trigonometric.tan attribute), 201
 method), 315

inverse_ADJ() (diofant.matrices.matrices.MatrixBase attribute), 600
 method), 600

inverse_cosine_transform() (in module diofant.integrals.transforms), 531

inverse_fourier_transform() (in module diofant.integrals.transforms), 529

inverse_GE() (diofant.matrices.matrices.MatrixBase method), 601

inverse_hankel_transform() (in module diofant.integrals.transforms), 532

inverse_laplace_transform() (in module diofant.integrals.transforms), 528

inverse_LU() (diofant.matrices.matrices.MatrixBase method), 601

inverse_mellin_transform() (in module diofant.integrals.transforms), 527

inverse_sine_transform() (in module diofant.integrals.transforms), 530

inverse_CopulaTransform (class in diofant.integrals.transforms), 558

inverse_FourierTransform (class in diofant.integrals.transforms), 558

inverse_HankelTransform (class in diofant.integrals.transforms), 559

inverse_LaplaceTransform (class in diofant.integrals.transforms), 558

inverse_MellinTransform (class in diofant.integrals.transforms), 557

inverse_SineTransform (class in diofant.integrals.transforms), 558

inversions() (diofant.combinatorics.permutations.Permutation method), 174

invert() (diofant.polys.domains.domain.Domain method), 766

invert() (diofant.polys.domains.ring.Ring method), 769

invert() (diofant.polys.polyclasses.DMF method), 781

invert() (diofant.polys.polyclasses.DMP method), 777

invert() (diofant.polys.polytools.Poly method), 518

invert() (in module diofant.polys.polytools), 516

is_abelian (diofant.combinatorics.perm_groups.Permutation attribute), 201

is_above_fermi (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 427

is_algebraic (diofant.core.basic.Basic attribute), 47

is_algebraic_expr() (diofant.core.expr.Expr method), 72

is_aliased (diofant.polys.numberfields.AlgebraicNumber attribute), 733

is_alt_sym() (diofant.combinatorics.perm_groups.Permutation method), 202

is_anti_symmetric() (diofant.matrices.matrices.MatrixBase method), 601

is_antihhermitian (diofant.core.basic.Basic attribute), 47

is_below_fermi (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 427

is_cnf() (in module diofant.logic.boolalg), 574

is_collinear() (diofant.geometry.point.Point method), 440

- is_commutative (diofant.core.basic.Basic attribute), 47
- is_comparable (diofant.core.basic.Basic attribute), 47
- is_complex (diofant.core.basic.Basic attribute), 48
- is_composite (diofant.core.basic.Basic attribute), 48
- is_composite (diofant.core.numbers.Integer attribute), 91
- is_concyclic() (diofant.geometry.point.Point2D method), 443
- is_conservative() (in module diofant.vector), 1226
- is_constant() (diofant.core.expr.Expr method), 73
- is_constant() (diofant.core.numbers.Number method), 86
- is_constant() (diofant.core.power.Pow method), 103
- is_convex() (diofant.geometry.polygon.Polygon method), 503
- is_coplanar() (diofant.geometry.plane.Plane method), 522
- is_cyclotomic (diofant.polys.polyclasses.DMP attribute), 777
- is_cyclotomic (diofant.polys.polytools.Poly attribute), 708
- is_diagonal() (diofant.matrices.matrices.MatrixBase method), 602
- is_diagonalizable() (diofant.matrices.matrices.MatrixBase method), 603
- is_disjoint() (diofant.sets.sets.Set method), 910
- is_dnf() (in module diofant.logic.boolalg), 574
- is_Empty (diofant.combinatorics.permutations.Permutation attribute), 175
- is_equilateral() (diofant.geometry.polygon.Triangle method), 516
- is_even (diofant.combinatorics.permutations.Permutation attribute), 176
- is_even (diofant.core.basic.Basic attribute), 48
- is_even (diofant.core.numbers.Integer attribute), 91
- is_extended_real (diofant.core.basic.Basic attribute), 48
- is_finite (diofant.core.basic.Basic attribute), 48
- is_full_module() (diofant.polys.agca.modules.SubModule method), 751
- is_full_module() (diofant.polys.agca.modules.SubQuotientModule method), 758
- is_groebner() (in module diofant.polys.groebnertools), 851
- is_ground (diofant.polys.polyclasses.ANP attribute), 781
- is_ground (diofant.polys.polyclasses.DMP attribute), 778
- is_ground (diofant.polys.polytools.Poly attribute), 708
- is_hermitian (diofant.core.basic.Basic attribute), 48
- is_hermitian (diofant.matrices.matrices.MatrixBase attribute), 603
- is_hermitian (diofant.matrices.sparse.SparseMatrixBase attribute), 647
- is_homogeneous (diofant.polys.polyclasses.DMP attribute), 778
- is_homogeneous (diofant.polys.polytools.Poly attribute), 709
- is_hypergeometric() (diofant.core.basic.Basic method), 48
- is_Identity (diofant.combinatorics.permutations.Permutation attribute), 175
- is_identity (diofant.core.function.Lambda attribute), 128
- is_imaginary (diofant.core.basic.Basic attribute), 49
- is_imaginary (diofant.core.numbers.Integer attribute), 92
- is_infinite (diofant.core.basic.Basic attribute), 49
- is_injective() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 761
- is_integer (diofant.core.basic.Basic attribute), 49
- is_irrational (diofant.core.basic.Basic attribute), 49
- is_irreducible (diofant.polys.polyclasses.DMP attribute), 778
- is_irreducible (diofant.polys.polytools.Poly attribute), 709
- is_isomorphism() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 761
- is_isosceles() (diofant.geometry.polygon.Triangle method), 516
- is_left_unbounded (diofant.sets.sets.Interval attribute), 914
- is_linear (diofant.polys.polyclasses.DMP attribute), 778

`is_linear` (diofant.polys.polytools.Poly attribute), 709
`is_lower` (diofant.matrices.matrices.MatrixBase attribute), 604
`is_lower_hessenberg` (diofant.matrices.matrices.MatrixBase attribute), 604
`is_maximal()` (diofant.polys.agca.ideals.Ideal method), 755
`is_minimal()` (in module diofant.polys.groebnertools), 851
`is_monic` (diofant.polys.polyclasses.DMP attribute), 778
`is_monic` (diofant.polys.polytools.Poly attribute), 709
`is_monomial` (diofant.polys.polyclasses.DMP attribute), 778
`is_monomial` (diofant.polys.polytools.Poly attribute), 710
`is_multivariate` (diofant.polys.polytools.Poly attribute), 710
`is_negative` (diofant.core.basic.Basic attribute), 49
`is_negative()` (diofant.polys.domains.AlgebraicField method), 772
`is_negative()` (diofant.polys.domains.domain.Domain method), 766
`is_negative()` (diofant.polys.domains.ExpressionDomain method), 775
`is_negative()` (diofant.polys.domains.FractionField method), 773
`is_negative()` (diofant.polys.domains.PolynomialRing method), 771
`is_nilpotent` (diofant.combinatorics.perm_groups.PermutationGroup attribute), 202
`is_nilpotent()` (diofant.matrices.matrices.MatrixBase method), 605
`is_noninteger` (diofant.core.basic.Basic attribute), 49
`is_nonnegative` (diofant.core.basic.Basic attribute), 49
`is_nonnegative()` (diofant.polys.domains.AlgebraicField method), 772
`is_nonnegative()` (diofant.polys.domains.domain.Domain method), 767
`is_nonnegative()` (diofant.polys.domains.ExpressionDomain method), 775
`is_nonnegative()` (diofant.polys.domains.FractionField method), 773
`is_nonnegative()` (diofant.polys.domains.PolynomialRing method), 772
`is_nonnegative()` (diofant.polys.domains.domain.Domain method), 767
`is_nonnegative()` (diofant.polys.domains.ExpressionDomain method), 775
`is_nonnegative()` (diofant.polys.domains.FractionField method), 773
`is_nonnegative()` (diofant.polys.domains.PolynomialRing method), 772
`is_nonzero` (diofant.core.basic.Basic attribute), 50
`is_nonzero` (diofant.core.numbers.Integer attribute), 92
`is_normal()` (diofant.combinatorics.perm_groups.PermutationGroup method), 203
`is_nthpow_residue()` (in module diofant.ntheory.residue_ntheory), 272
`is_number` (diofant.concrete.expr_with_limits.ExprWithLimits attribute), 289
`is_number` (diofant.core.expr.Expr attribute), 74
`is_odd` (diofant.combinatorics.permutations.Permutation attribute), 76
`is_odd` (diofant.core.basic.Basic attribute), 50
`is_odd` (diofant.core.numbers.Integer attribute), 92
`is_one` (diofant.polys.polyclasses.ANP attribute), 781
`is_one` (diofant.polys.polyclasses.DMF attribute), 781
`is_one` (diofant.polys.polyclasses.DMP attribute), 778
`is_one` (diofant.polys.polytools.Poly attribute), 710
`is_one()` (diofant.polys.domains.domain.Domain method), 767
`is_only_above_fermi` (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 427
`is_only_below_fermi` (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 428

is_parallel() (diofant.geometry.line.LinearEntity method), 452
 is_parallel() (diofant.geometry.line3d.LinearEntity3D method), 471
 is_parallel() (diofant.geometry.plane.Plane method), 522
 is_perpendicular() (diofant.geometry.line.LinearEntity method), 453
 is_perpendicular() (diofant.geometry.line3d.LinearEntity3D method), 472
 is_perpendicular() (diofant.geometry.plane.Plane method), 522
 is_polar (diofant.core.basic.Basic attribute), 50
 is_polynomial() (diofant.core.expr.Expr method), 75
 is_positive (diofant.core.basic.Basic attribute), 50
 is_positive() (diofant.polys.domains.AlgebraicField method), 772
 is_positive() (diofant.polys.domains.domain.Domain method), 767
 is_positive() (diofant.polys.domains.ExpressionDomain method), 775
 is_positive() (diofant.polys.domains.FractionField method), 773
 is_positive() (diofant.polys.domains.PolynomialRing method), 772
 is_primary() (diofant.polys.agca.ideals.Ideal method), 755
 is_prime (diofant.core.basic.Basic attribute), 50
 is_prime (diofant.core.numbers.Integer attribute), 92
 is_prime() (diofant.polys.agca.ideals.Ideal method), 755
 is_primitive (diofant.polys.polyclasses.DMP attribute), 778
 is_primitive (diofant.polys.polytools.Poly attribute), 711
 is_primitive() (diofant.combinatorics.perm_groups.PermutationGroup method), 203
 is_primitive_root() (in module diofant.ntheory.residue_ntheory), 271
 is_principal() (diofant.polys.agca.ideals.Ideal method), 755
 is_proper_subset() (diofant.sets.sets.Set method), 910
 is_proper_superset() (diofant.sets.sets.Set method), 910
 is_quad_residue() (in module diofant.ntheory.residue_ntheory), 273
 is_quadratic (diofant.polys.polyclasses.DMP attribute), 778
 is_quadratic (diofant.polys.polytools.Poly attribute), 711
 is_radical() (diofant.polys.agca.ideals.Ideal method), 755
 is_rational (diofant.core.basic.Basic attribute), 50
 is_rational_function() (diofant.core.expr.Expr method), 75
 is_real (diofant.core.basic.Basic attribute), 50
 is_reduced() (in module diofant.polys.groebnertools), 851
 is_right() (diofant.geometry.polygon.Triangle method), 517
 is_right_unbounded (diofant.sets.sets.Interval attribute), 914
 is_scalar_multiple() (diofant.geometry.point.Point method), 441
 is_scalene() (diofant.geometry.polygon.Triangle method), 517
 is_sequence() (in module diofant.core.compatibility), 151, 155
 is_similar() (diofant.geometry.entity.GeometryEntity method), 433
 is_similar() (diofant.geometry.line.LinearEntity method), 453
 is_similar() (diofant.geometry.line3d.LinearEntity3D method), 472
 is_similar() (diofant.geometry.polygon.Triangle method), 517
 is_simple() (diofant.functions.special.delta_functions.DiracDelta method), 357
 is_Singleton (diofant.combinatorics.permutations.Permutation attribute), 175
 is_solenoidal() (in module diofant.vector), 1226
 is_solvable (diofant.combinatorics.perm_groups.PermutationGroup attribute), 204
 is_sqf (diofant.polys.polyclasses.DMP attribute), 778
 is_sqf (diofant.polys.polytools.Poly attribute), 711
 is_square (diofant.matrices.matrices.MatrixBase attribute), 605
 is_subgroup() (diofant.combinatorics.perm_groups.PermutationGroup method), 204
 is_submodule() (diofant.polys.agca.modules.FreeModule

method), 748

is_submodule() (diofant.polys.agca.modules.Module method), 747

is_submodule() (diofant.polys.agca.modules.QuotientModule method), 757

is_submodule() (diofant.polys.agca.modules.SubModule method), 751

is_subset() (diofant.sets.sets.Set method), 910

is_superset() (diofant.sets.sets.Set method), 911

is_surjective() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 762

is_symbolic() (diofant.matrices.matrices.MatrixBase method), 605

is_symmetric() (diofant.matrices.matrices.MatrixBase method), 606

is_symmetric() (diofant.matrices.sparse.SparseMatrixBase method), 647

is_tangent() (diofant.geometry.ellipse.Ellipse method), 490

is_transcendental (diofant.core.basic.Basic attribute), 50

is_transitive() (diofant.combinatorics.perm_groups.PermutationGroup method), 205

is_trivial (diofant.combinatorics.perm_groups.PermutationGroup attribute), 205

is_univariate (diofant.polys.polytools.Poly attribute), 711

is_upper (diofant.matrices.matrices.MatrixBase attribute), 607

is_upper_hessenberg (diofant.matrices.matrices.MatrixBase attribute), 607

is_whole_ring() (diofant.polys.agca.ideals.Ideal method), 755

is_zero (diofant.core.basic.Basic attribute), 51

is_zero (diofant.core.numbers.Integer attribute), 92

is_zero (diofant.geometry.point.Point attribute), 441

is_zero (diofant.matrices.immutable.ImmutableMatrix attribute), 654

is_zero (diofant.matrices.immutable.ImmutableSparseMatrix attribute), 651

is_zero (diofant.matrices.matrices.MatrixBase attribute), 608

is_zero (diofant.polys.polyclasses.ANP attribute), 781

is_zero (diofant.polys.polyclasses.DMF attribute), 781

is_zero (diofant.polys.polyclasses.DMP attribute), 778

is_zero (diofant.polys.polytools.Poly attribute), 712

is_zero() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 762

is_zero() (diofant.polys.agca.ideals.Ideal method), 755

is_zero() (diofant.polys.agca.modules.FreeModule method), 749

is_zero() (diofant.polys.agca.modules.Module method), 747

is_zero() (diofant.polys.agca.modules.QuotientModule method), 757

is_zero() (diofant.polys.agca.modules.SubModule method), 751

is_zero() (diofant.polys.domains.domain.Domain method), 767

is_zero_dimensional (diofant.polys.polytools.GroebnerBasis attribute), 728

is_zero_dimensional() (in module diofant.polys.polytools), 690

isdisjoint() (diofant.sets.sets.Set method), 910

isolate() (in module diofant.polys.numberfields), 732

IsomorphismFailed (class in diofant.polys.polyerrors), 853

isprime() (in module diofant.ntheory.primetest), 270

issubset() (diofant.sets.sets.Set method), 911

issuperset() (diofant.sets.sets.Set method), 911

ITE (class in diofant.logic.boolalg), 573

items() (diofant.core.containers.Dict method), 149

iterable() (in module diofant.core.compatibility), 152, 154

iterate_binary() (diofant.combinatorics.subsets.Subset method), 224

iterate_graycode() (diofant.combinatorics.subsets.Subset method), 224

itercoeffs() (diofant.polys.rings.PolyElement method), 831

- itermonomials() (in module diofant.polys.monomials), 733
- itermonoms() (diofant.polys.rings.PolyElement method), 831
- iterterms() (diofant.polys.rings.PolyElement method), 831
- ## J
- jacobi (class in diofant.functions.special.polynomials), 412
- jacobi_normalized() (in module diofant.functions.special.polynomials), 413
- jacobi_poly() (in module diofant.polys.orthopolys), 736
- jacobi_symbol() (in module diofant.ntheory.residue_ntheory), 273
- jacobian() (diofant.diffgeom.CoordSystem method), 1180
- jacobian() (diofant.matrices.matrices.MatrixBase method), 608
- jn (class in diofant.functions.special.bessel), 391
- jn_zeros() (in module diofant.functions.special.bessel), 392
- jordan_cell() (in module diofant.matrices.dense), 624
- jordan_cells() (diofant.matrices.matrices.MatrixBase method), 609
- jordan_form() (diofant.matrices.matrices.MatrixBase method), 609
- josephus() (diofant.combinatorics.permutations.Permutation class method), 176
- ## K
- kbins() (in module diofant.utilities.iterables), 1150
- kernel() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 762
- key2bounds() (diofant.matrices.matrices.MatrixBase method), 610
- key2ij() (diofant.matrices.matrices.MatrixBase method), 610
- keys() (diofant.core.containers.Dict method), 149
- killable_index (diofant.functions.special.tensor_functions attribute), 428
- known_functions (in module diofant.printing.ccode), 868
- known_functions (in module diofant.printing.mathematica), 874
- KroneckerDelta (class in diofant.functions.special.tensor_functions), 425
- ksubsets() (diofant.combinatorics.subsets method), 230
- Kumaraswamy() (in module diofant.stats), 972
- ## L
- l1_norm() (diofant.polys.polyclasses.DMP method), 778
- l1_norm() (diofant.polys.polytools.Poly method), 712
- label (diofant.tensor.indexed.Idx attribute), 1099
- label (diofant.tensor.indexed.IndexedBase attribute), 1102
- laguerre (class in diofant.functions.special.polynomials), 420
- laguerre_poly() (in module diofant.polys.orthopolys), 736
- Lambda (class in diofant.core.function), 127
- LambdaPrinter (class in diofant.printing.lambdarepr), 875
- lambdarepr() (in module diofant.printing.lambdarepr), 875
- lambdastr() (in module diofant.utilities.lambdify), 1163
- lambdify() (in module diofant.utilities.lambdify), 1163
- LambertW (class in diofant.functions.elementary.exponential), 331
- Laplace() (in module diofant.stats), 973
- laplace_transform() (in module diofant.integrals.transforms), 528
- LaplaceTransform (class in diofant.integrals.transforms), 558
- latex() (in module diofant.printing.latex), 875
- LatexPrinter (class in diofant.printing.latex), 875
- LC() (diofant.polys.polyclasses.ANP method), 781
- LC() (diofant.polys.polyclasses.DMP method), 775
- LC() (diofant.polys.polytools.Poly method), 691
- KroneckerDelta
- LC() (in module diofant.polys.polytools), 673

- lcm() (diofant.core.numbers.Number method), 86
- lcm() (diofant.core.numbers.Rational method), 91
- lcm() (diofant.polys.domains.domain.Domain method), 767
- lcm() (diofant.polys.domains.field.Field method), 768
- lcm() (diofant.polys.domains.PolynomialRing method), 772
- lcm() (diofant.polys.domains.RealField method), 774
- lcm() (diofant.polys.polyclasses.DMP method), 778
- lcm() (diofant.polys.polytools.Poly method), 712
- lcm() (in module diofant.polys.polytools), 681
- lcm_list() (in module diofant.polys.polytools), 682
- ldescent() (in module diofant.solvers.diophantine), 1091
- LDLdecomposition() (diofant.matrices.matrices.MatrixBase method), 587
- LDLdecomposition() (diofant.matrices.sparse.SparseMatrixBase method), 643
- LDLsolve() (diofant.matrices.matrices.MatrixBase method), 588
- Le (in module diofant.core.relational), 110
- leading_expv() (diofant.polys.rings.PolyElement method), 831
- leading_monom() (diofant.polys.rings.PolyElement method), 831
- leading_term() (diofant.polys.rings.PolyElement method), 832
- left (diofant.sets.sets.Interval attribute), 914
- left() (diofant.printing.pretty.stringpict.StringPict method), 883
- left_open (diofant.sets.sets.Interval attribute), 914
- legendre (class in diofant.functions.special.polynomials), 417
- legendre_poly() (in module diofant.polys.orthopolys), 736
- legendre_symbol() (in module diofant.ntheory.residue_ntheory), 273
- length (diofant.geometry.line.LinearEntity attribute), 453
- length (diofant.geometry.line.Segment attribute), 464
- length (diofant.geometry.line3d.LinearEntity3D attribute), 472
- length (diofant.geometry.line3d.Segment3D attribute), 479
- length (diofant.geometry.point.Point attribute), 441
- length (diofant.geometry.polygon.RegularPolygon attribute), 510
- length() (diofant.combinatorics.permutations.Permutation method), 177
- length() (diofant.polys.polytools.Poly method), 712
- lerchphi (class in diofant.functions.special.zeta_functions), 403
- LessThan (class in diofant.core.relational), 116
- LeviCivita (class in diofant.functions.special.tensor_functions), 425
- LexOrder (class in diofant.polys.orderings), 734
- lhs (diofant.core.relational.Relational attribute), 112
- Li (class in diofant.functions.special.error_functions), 382
- li (class in diofant.functions.special.error_functions), 381
- lie_heuristic_abaco1_product() (in module diofant.solvers.ode), 1025
- lie_heuristic_abaco1_simple() (in module diofant.solvers.ode), 1024
- lie_heuristic_abaco2_similar() (in module diofant.solvers.ode), 1026
- lie_heuristic_abaco2_unique_unknown() (in module diofant.solvers.ode), 1027
- lie_heuristic_bivariate() (in module diofant.solvers.ode), 1025
- lie_heuristic_chi() (in module diofant.solvers.ode), 1026
- lie_heuristic_function_sum() (in module diofant.solvers.ode), 1027
- lie_heuristic_linear() (in module diofant.solvers.ode), 1028
- LieDerivative (class in diofant.diffgeom), 1186
- lift() (diofant.polys.polyclasses.DMP method), 778
- lift() (diofant.polys.polytools.Poly method), 713
- Limit (class in diofant.series.limits), 902
- limit() (diofant.core.expr.Expr method), 76

- limit() (diofant.matrices.matrices.MatrixBase method), 610
 - limit() (in module diofant.series.limits), 902
 - limit_denominator() (diofant.core.numbers.Rational method), 91
 - limitinf() (in module diofant.series.gruntz), 906
 - limits (diofant.concrete.expr_with_limits.ExprWithLimits attribute), 289
 - limits (diofant.geometry.curve.Curve attribute), 482
 - Line (class in diofant.geometry.line), 457
 - Line2DBaseSeries (class in diofant.plotting.plot), 899
 - Line3D (class in diofant.geometry.line3d), 466
 - Line3DBaseSeries (class in diofant.plotting.plot), 899
 - line_integrate() (in module diofant.integrals.integrals), 549
 - LinearEntity (class in diofant.geometry.line), 449
 - LinearEntity3D (class in diofant.geometry.line3d), 468
 - LineOver1DRangeSeries (class in diofant.plotting.plot), 899
 - list() (diofant.combinatorics.permutations.Cycle method), 186
 - list() (diofant.combinatorics.permutations.Permutation method), 177
 - list2numpy() (in module diofant.matrices.dense), 626
 - list_visitor() (in module diofant.utilities.enumerative), 1139
 - listcoeffs() (diofant.polys.rings.PolyElement method), 832
 - listmonoms() (diofant.polys.rings.PolyElement method), 832
 - listterms() (diofant.polys.rings.PolyElement method), 832
 - liupc() (diofant.matrices.sparse.SparseMatrixBase method), 647
 - LM() (diofant.polys.polytools.Poly method), 691
 - LM() (in module diofant.polys.polytools), 673
 - locate_new() (diofant.vector.coordsysrect.CoordSysCartesian method), 1209
 - log (class in diofant.functions.elementary.exponential), 331
 - log() (diofant.polys.domains.domain.Domain method), 767
 - log() (diofant.polys.domains.IntegerRing method), 771
 - logcombine() (in module diofant.simplify.simplify), 928
 - loggamma (class in diofant.functions.special.gamma_functions), 360
 - Logistic() (in module diofant.stats), 973
 - LogNormal() (in module diofant.stats), 974
 - Lopen() (diofant.sets.sets.Interval class method), 914
 - lower (diofant.tensor.indexed.Idx attribute), 1099
 - lower_central_series() (diofant.combinatorics.perm_groups.PermutationGroup method), 205
 - lower_triangular_solve() (diofant.matrices.matrices.MatrixBase method), 610
 - lowergamma (class in diofant.functions.special.gamma_functions), 365
 - lseries() (diofant.core.expr.Expr method), 76
 - Lt (in module diofant.core.relational), 110
 - LT() (diofant.polys.polytools.Poly method), 691
 - LT() (in module diofant.polys.polytools), 673
 - ltrim() (diofant.polys.polytools.Poly method), 713
 - lutacion (class in diofant.functions.combinatorial.numbers), 351
 - LUdecomposition() (diofant.matrices.matrices.MatrixBase method), 588
 - LUdecomposition_Simple() (diofant.matrices.matrices.MatrixBase method), 589
 - LUdecompositionFF() (diofant.matrices.matrices.MatrixBase method), 588
 - lsolve() (diofant.matrices.matrices.MatrixBase method), 589
- ## M
- magnitude() (diofant.vector.vector.Vector method), 1216
 - major (diofant.geometry.ellipse.Ellipse attribute), 491
 - make_perm() (diofant.combinatorics.perm_groups.PermutationGroup method), 206
 - make_prime() (in module diofant.solvers.diophantine), 1093

- make_routine() (in module diofant.utilities.codegen), 1135
- Manifold (class in diofant.diffgeom), 1177
- map() (diofant.polys.domains.domain.Domain method), 767
- MatAdd (class in diofant.matrices.expressions), 656
- match() (diofant.core.basic.Basic method), 51
- matches() (diofant.core.add.Add method), 109
- matches() (diofant.core.basic.Atom method), 41
- matches() (diofant.core.basic.Basic method), 51
- matches() (diofant.core.function.WildFunction method), 129
- matches() (diofant.core.mul.Mul method), 106
- matches() (diofant.core.power.Pow method), 103
- matches() (diofant.core.symbol.Wild method), 82
- mathematica() (in module diofant.parsing.mathematica), 1168
- mathematica_code() (in module diofant.printing.mathematica), 874
- mathml() (in module diofant.printing.mathml), 878
- mathml_tag() (diofant.printing.mathml.MathMLPrinter method), 877
- MathMLPrinter (class in diofant.printing.mathml), 877
- MatMul (class in diofant.matrices.expressions), 656
- MatPow (class in diofant.matrices.expressions), 657
- Matrix (class in diofant.matrices), 577
- matrix2numpy() (in module diofant.matrices.dense), 626
- matrix_fgln() (in module diofant.polys.fglnertools), 851
- matrix_multiply_elementwise() (in module diofant.matrices.dense), 622
- matrix_to_vector() (in module diofant.vector), 1223
- MatrixBase (class in diofant.matrices.matrices), 586
- MatrixError (class in diofant.matrices.matrices), 621
- MatrixExpr (class in diofant.matrices.expressions), 655
- MatrixSymbol (class in diofant.matrices.expressions), 656
- Max (class in diofant.functions.elementary.miscellaneous), 334
- max() (diofant.combinatorics.permutations.Permutation method), 178
- max_div (diofant.combinatorics.perm_groups.PermutationGroup attribute), 206
- max_norm() (diofant.polys.polyclasses.DMP method), 778
- max_norm() (diofant.polys.polytools.Poly method), 713
- maximize() (in module diofant.calculus.optimization), 1172
- Maxwell() (in module diofant.stats), 975
- MCodePrinter (class in diofant.printing.mathematica), 874
- measure (diofant.sets.sets.Set attribute), 911
- medial (diofant.geometry.polygon.Triangle attribute), 518
- medians (diofant.geometry.polygon.Triangle attribute), 518
- meijerg (class in diofant.functions.special.hyper), 406
- mellin_transform() (in module diofant.integrals.transforms), 526
- MellinTransform (class in diofant.integrals.transforms), 557
- merge_solution() (in module diofant.solvers.diophantine), 1088
- metric_to_Christoffel_1st() (in module diofant.diffgeom), 1190
- metric_to_Christoffel_2nd() (in module diofant.diffgeom), 1190
- metric_to_Ricci_components() (in module diofant.diffgeom), 1191
- metric_to_Riemann_components() (in module diofant.diffgeom), 1191
- midpoint (diofant.geometry.line.Segment attribute), 465
- midpoint (diofant.geometry.line3d.Segment3D attribute), 480
- midpoint() (diofant.geometry.point.Point method), 441
- Min (class in diofant.functions.elementary.miscellaneous), 333
- min() (diofant.combinatorics.permutations.Permutation method), 178
- minimal_block() (diofant.combinatorics.perm_groups.PermutationGroup method), 207
- minimal_polynomial() (in module diofant.polys.numberfields), 731

- minimize() (in module diofant.calculus.optimization), 1172
- minlex() (in module diofant.utilities.iterables), 1151
- minor (diofant.geometry.ellipse.Ellipse attribute), 491
- minorEntry() (diofant.matrices.matrices.MatrixBase method), 611
- minorMatrix() (diofant.matrices.matrices.MatrixBase method), 611
- minpoly() (in module diofant.polys.numberfields), 732
- mobius (class in diofant.ntheory), 277
- Mod (class in diofant.core.mod), 110
- modgcd_bivariate() (in module diofant.polys.modulargcd), 854
- modgcd_multivariate() (in module diofant.polys.modulargcd), 856
- modgcd_univariate() (in module diofant.polys.modulargcd), 853
- Module (class in diofant.polys.agca.modules), 747
- module_quotient() (diofant.polys.agca.modules.SubModule method), 751
- ModuleHomomorphism (class in diofant.polys.agca.homomorphisms), 760
- monic() (diofant.polys.polyclasses.DMP method), 778
- monic() (diofant.polys.polytools.Poly method), 713
- monic() (diofant.polys.rings.PolyElement method), 832
- monic() (in module diofant.polys.polytools), 682
- Monomial (class in diofant.polys.monomials), 733
- monomial_basis() (diofant.polys.rings.PolyRing method), 828
- monomial_count() (in module diofant.polys.monomials), 733
- monoms() (diofant.polys.polyclasses.DMP method), 778
- monoms() (diofant.polys.polytools.Poly method), 714
- monoms() (diofant.polys.rings.PolyElement method), 832
- mr() (in module diofant.ntheory.primetest), 270
- mrsv() (in module diofant.series.gruntz), 907
- mrsv_leadterm() (in module diofant.series.gruntz), 907
- mrsv_max() (in module diofant.series.gruntz), 907
- Mul (class in diofant.core.mul), 103
- mul() (diofant.polys.domains.domain.Domain method), 767
- mul() (diofant.polys.polyclasses.DMF method), 781
- mul() (diofant.polys.polyclasses.DMP method), 778
- mul() (diofant.polys.polytools.Poly method), 714
- mul() (diofant.polys.rings.PolyRing method), 828
- mul_ground() (diofant.polys.polyclasses.DMP method), 778
- mul_ground() (diofant.polys.polytools.Poly method), 714
- mul_inv() (diofant.combinatorics.permutations.Permutation method), 178
- MultiFactorial (class in diofant.functions.combinatorial.factorials), 351
- multinomial_coefficients() (in module diofant.ntheory.multinomial), 268
- multinomial_coefficients_iterator() (in module diofant.ntheory.multinomial), 269
- multiplicity() (in module diofant.ntheory.factor_), 257
- multiply() (diofant.matrices.matrices.MatrixBase method), 611
- multiply() (diofant.matrices.sparse.SparseMatrixBase method), 648
- multiply_elementwise() (diofant.matrices.matrices.MatrixBase method), 611
- multiply_ideal() (diofant.polys.agca.modules.FreeModule method), 749
- multiply_ideal() (diofant.polys.agca.modules.Module method), 747
- multiply_ideal() (diofant.polys.agca.modules.SubModule method), 752
- multiset() (in module diofant.utilities.iterables), 1152
- multiset_combinations() (in module diofant.utilities.iterables), 1152
- multiset_partitions() (in module diofant.utilities.iterables), 1153
- multiset_partitions_taocp() (in module diofant.utilities.enumerative), 1138

- multiset_permutations() (in module diofant.utilities.iterables), 1154
 - MultisetPartitionTraverser (class in diofant.utilities.enumerative), 1139
 - MultivariatePolynomialError (class in diofant.polys.polyerrors), 853
 - MutableDenseMatrix (class in diofant.matrices.dense), 633
 - MutableMatrix (in module diofant.matrices.dense), 629
 - MutableSparseMatrix (class in diofant.matrices.sparse), 639
- N**
- n (diofant.combinatorics.graycode.GrayCode attribute), 231
 - n() (diofant.core.evalf.EvalfMixin method), 147
 - n() (diofant.core.function.Subs method), 137
 - n() (diofant.geometry.point.Point method), 442
 - n() (diofant.matrices.matrices.MatrixBase method), 611
 - n() (diofant.polys.domains.domain.Domain method), 767
 - N() (in module diofant.core.evalf), 148
 - n_order() (in module diofant.ntheory.residue_ntheory), 270
 - Nakagami() (in module diofant.stats), 976
 - NaN (class in diofant.core.numbers), 95
 - Nand (class in diofant.logic.boolalg), 571
 - nargs (diofant.core.function.FunctionClass attribute), 133
 - native_coeffs() (diofant.polys.numberfields.AlgebraicNumber method), 733
 - Naturals (class in diofant.sets.fancysets), 919
 - Naturals0 (class in diofant.sets.fancysets), 920
 - nC() (in module diofant.functions.combinatorial.numbers), 354
 - Ne (in module diofant.core.relational), 110
 - necklaces() (in module diofant.utilities.iterables), 1154
 - neg() (diofant.polys.domains.domain.Domain method), 767
 - neg() (diofant.polys.polyclasses.DMF method), 781
 - neg() (diofant.polys.polyclasses.DMP method), 778
 - neg() (diofant.polys.polytools.Poly method), 714
 - NegativeInfinity (class in diofant.core.numbers), 96
 - NegativeOne (class in diofant.core.numbers), 94
 - new() (diofant.polys.polytools.Poly class method), 715
 - next() (diofant.combinatorics.graycode.GrayCode method), 231
 - next() (diofant.combinatorics.prufer.Prufer method), 220
 - next() (diofant.printing.pretty.stringpict.stringPict static method), 883
 - next_binary() (diofant.combinatorics.subsets.Subset method), 225
 - next_gray() (diofant.combinatorics.subsets.Subset method), 225
 - next_lex() (diofant.combinatorics.partitions.IntegerPartition method), 160
 - next_lex() (diofant.combinatorics.permutations.Permutation method), 178
 - next_lexicographic() (diofant.combinatorics.subsets.Subset method), 225
 - next_nonlex() (diofant.combinatorics.permutations.Permutation method), 178
 - next_trotterjohnson() (diofant.combinatorics.permutations.Permutation method), 179
 - nextprime() (in module diofant.ntheory.generate), 252
 - nfloat() (in module diofant.core.function), 146
 - nnz() (diofant.matrices.sparse.SparseMatrixBase method), 648
 - no_attrs_in_subclass (class in diofant.utilities.decorator), 1137
 - nodes (diofant.combinatorics.prufer.Prufer attribute), 220
 - NonSquareMatrixError (class in diofant.matrices.matrices), 621
 - Nor (class in diofant.logic.boolalg), 572
 - norm() (diofant.matrices.matrices.MatrixBase method), 611
 - normal() (diofant.core.expr.Expr method), 77
 - Normal() (in module diofant.stats), 976
 - normal_closure() (diofant.combinatorics.perm_groups.PermutationGroup method), 207
 - normal_lines() (diofant.geometry.ellipse.Ellipse method), 492
 - normal_vector (diofant.geometry.plane.Plane attribute), 523

- normalize() (diofant.vector.vector.Vector method), 1216
- normalized() (diofant.matrices.matrices.MatrixBase method), 612
- NormalPSpace (class in diofant.stats.crv_types), 991
- Not (class in diofant.logic.boolalg), 570
- NotAlgebraic (class in diofant.polys.polyerrors), 853
- NotInvertible (class in diofant.polys.polyerrors), 853
- NotIterable (class in diofant.core.compatibility), 149
- NotReversible (class in diofant.polys.polyerrors), 853
- nP() (in module diofant.functions.combinatorial.numbers), 355
- npartitions() (in module diofant.ntheory.partitions_), 269
- nroots() (diofant.polys.polytools.Poly method), 715
- nroots() (in module diofant.polys.polytools), 687
- nseries() (diofant.core.expr.Expr method), 77
- nsimplify() (diofant.core.expr.Expr method), 78
- nsimplify() (in module diofant.simplify.simplify), 926
- nsolve() (in module diofant.solvers.solvers), 1064
- nT() (in module diofant.functions.combinatorial.numbers), 356
- nth() (diofant.polys.polyclasses.DMP method), 778
- nth() (diofant.polys.polytools.Poly method), 715
- nth_power_roots_poly() (diofant.polys.polytools.Poly method), 716
- nth_power_roots_poly() (in module diofant.polys.polytools), 688
- nthroot() (in module diofant.simplify.simplify), 925
- nthroot_mod() (in module diofant.ntheory.residue_ntheory), 272
- nu (diofant.functions.special.hyper.meijerg attribute), 409
- nullspace() (diofant.matrices.matrices.MatrixBase method), 612
- num (diofant.core.numbers.Float attribute), 89
- Number (class in diofant.core.numbers), 86
- numbered_symbols() (in module diofant.utilities.iterables), 1155
- NumberSymbol (class in diofant.core.numbers), 92
- numer() (diofant.polys.domains.AlgebraicField method), 773
- numer() (diofant.polys.domains.domain.Domain method), 767
- numer() (diofant.polys.domains.ExpressionDomain method), 775
- numer() (diofant.polys.domains.FractionField method), 773
- numer() (diofant.polys.domains.ring.Ring method), 769
- numer() (diofant.polys.polyclasses.DMF method), 781
- O**
- O (in module diofant.series.order), 903
- OctaveCodeGen (class in diofant.utilities.codegen), 1133
- ode_1st_exact() (in module diofant.solvers.ode), 1006
- ode_1st_homogeneous_coeff_best() (in module diofant.solvers.ode), 1006
- ode_1st_homogeneous_coeff_subs_dep_div_indep() (in module diofant.solvers.ode), 1007
- ode_1st_homogeneous_coeff_subs_indep_div_dep() (in module diofant.solvers.ode), 1008
- ode_1st_linear() (in module diofant.solvers.ode), 1010
- ode_1st_power_series() (in module diofant.solvers.ode), 1022
- ode_2nd_power_series_ordinary() (in module diofant.solvers.ode), 1022
- ode_2nd_power_series_regular() (in module diofant.solvers.ode), 1023
- ode_almost_linear() (in module diofant.solvers.ode), 1018
- ode_Bernoulli() (in module diofant.solvers.ode), 1011
- ode_lie_group() (in module diofant.solvers.ode), 1021
- ode_linear_coefficients() (in module diofant.solvers.ode), 1019
- ode_Liouville() (in module diofant.solvers.ode), 1012
- ode_nth_linear_constant_coeff_homogeneous() (in module diofant.solvers.ode), 1014
- ode_nth_linear_constant_coeff_undetermined_coefficients() (in module diofant.solvers.ode), 1015
- ode_nth_linear_constant_coeff_variation_of_parameters() (in module diofant.solvers.ode), 1016

ode_order() (in module diofant.solvers.deutils), 1067

ode_Riccati_special_minus2() (in module diofant.solvers.ode), 1013

ode_separable() (in module diofant.solvers.ode), 1017

ode_separable_reduced() (in module diofant.solvers.ode), 1020

ode_sol_simplicity() (in module diofant.solvers.ode), 1004

odesimp() (in module diofant.solvers.ode), 1002

of_type() (diofant.polys.domains.domain.Domain method), 767

old_frac_field() (diofant.polys.domains.domain.Domain method), 767

old_poly_ring() (diofant.polys.domains.domain.Domain method), 767

One (class in diofant.core.numbers), 94

one (diofant.polys.polytools.Poly attribute), 716

ones() (in module diofant.matrices.dense), 622

open() (diofant.sets.sets.Interval class method), 914

OperationNotSupported (class in diofant.polys.polyerrors), 853

opt_cse() (in module diofant.simplify.cse_main), 942

OptionError (class in diofant.polys.polyerrors), 853

Options (class in diofant.polys.polyoptions), 863

Or (class in diofant.logic.boolalg), 569

orbit() (diofant.combinatorics.perm_groups.PermutationGroup method), 208

orbit_rep() (diofant.combinatorics.perm_groups.PermutationGroup method), 209

orbit_transversal() (diofant.combinatorics.perm_groups.PermutationGroup method), 209

orbits() (diofant.combinatorics.perm_groups.PermutationGroup method), 210

Order (class in diofant.polys.polyoptions), 863

Order (class in diofant.series.order), 903

order (diofant.functions.special.bessel.BesselBase attribute), 388

order() (diofant.combinatorics.perm_groups.PermutationGroup method), 210

order() (diofant.combinatorics.permutations.Permutation method), 179

ordered() (in module diofant.core.compatibility), 153

orient_new() (diofant.vector.coordsysrect.CoordSysCartesian method), 1210

orient_new_axis() (diofant.vector.coordsysrect.CoordSysCartesian method), 1211

orient_new_body() (diofant.vector.coordsysrect.CoordSysCartesian method), 1211

orient_new_quaternion() (diofant.vector.coordsysrect.CoordSysCartesian method), 1212

orient_new_space() (diofant.vector.coordsysrect.CoordSysCartesian method), 1213

Orienter (class in diofant.vector.orienters), 1220

origin (diofant.geometry.point.Point attribute), 442

orthocenter (diofant.geometry.polygon.Triangle attribute), 518

outer() (diofant.vector.vector.Vector method), 1216

P

P() (in module diofant.stats), 986

p1 (diofant.geometry.line.LinearEntity attribute), 454

p1 (diofant.geometry.line3d.LinearEntity3D attribute), 473

p1 (diofant.geometry.plane.Plane attribute), 523

p2 (diofant.geometry.line.LinearEntity attribute), 454

p2 (diofant.geometry.line3d.LinearEntity3D attribute), 473

pager_print() (in module diofant.sphinx_helpers.pretty.pretty), 867

pairwise_prime() (in module diofant.solvers.diophantine), 1092

parallel_line() (diofant.geometry.line.LinearEntity method), 454

parallel_line() (diofant.geometry.line3d.LinearEntity3D method), 473

parallel_plane() (diofant.geometry.plane.Plane method), 523

parallelism_group_from_expr() (in module diofant.polys.polytools), 672

partitioner (diofant.geometry.curve.Curve attribute), 483

- Parametric2DLineSeries (class in diofant.plotting.plot), 899
- Parametric3DLineSeries (class in diofant.plotting.plot), 899
- ParametricSurfaceSeries (class in diofant.plotting.plot), 900
- parametrize_ternary_quadratic() (in module diofant.solvers.diophantine), 1090
- parens() (diofant.printing.pretty.stringpict.stringPict method), 883
- Pareto() (in module diofant.stats), 977
- parity() (diofant.combinatorics.permutations.Permutation method), 180
- parse_expr() (in module diofant.parsing.sympy_parser), 1167
- parse_maxima() (in module diofant.parsing.maxima), 1168
- Partition (class in diofant.combinatorics.partitions), 158
- partition (diofant.combinatorics.partitions.Partition attribute), 159
- partition() (in module diofant.solvers.diophantine), 1086
- partitions() (in module diofant.utilities.iterables), 1155
- Patch (class in diofant.diffgeom), 1177
- pde_1st_linear_constant_coeff() (in module diofant.solvers.pde), 1051
- pde_1st_linear_constant_coeff_homogeneous() (in module diofant.solvers.pde), 1050
- pde_1st_linear_variable_coeff() (in module diofant.solvers.pde), 1052
- pde_separate() (in module diofant.solvers.pde), 1045
- pde_separate_add() (in module diofant.solvers.pde), 1046
- pde_separate_mul() (in module diofant.solvers.pde), 1046
- pdiv() (diofant.polys.polyclasses.DMP method), 778
- pdiv() (diofant.polys.polytools.Poly method), 716
- pdiv() (in module diofant.polys.polytools), 674
- pdsolve() (in module diofant.solvers.pde), 1047
- per() (diofant.polys.polyclasses.DMF method), 781
- per() (diofant.polys.polyclasses.DMP method), 779
- per() (diofant.polys.polytools.Poly method), 716
- perfect_power() (in module diofant.ntheory.factor_), 257
- periapsis (diofant.geometry.ellipse.Ellipse attribute), 492
- perimeter (diofant.geometry.polygon.Polygon attribute), 503
- periodic_argument (class in diofant.functions.elementary.complexes), 311
- perm2tensor() (diofant.tensor.tensor.TensMul method), 1120
- Permutation (class in diofant.combinatorics.permutations), 163
- PermutationGroup (class in diofant.combinatorics.perm_groups), 188
- permute() (diofant.polys.polyclasses.DMP method), 779
- permuteBkwd() (diofant.matrices.matrices.MatrixBase method), 612
- permuteFwd() (diofant.matrices.matrices.MatrixBase method), 613
- perpendicular_bisector() (diofant.geometry.line.Segment method), 465
- perpendicular_line() (diofant.geometry.line.LinearEntity method), 455
- perpendicular_line() (diofant.geometry.line3d.LinearEntity3D method), 473
- perpendicular_line() (diofant.geometry.plane.Plane method), 523
- perpendicular_plane() (diofant.geometry.plane.Plane method), 524
- perpendicular_segment() (diofant.geometry.line.LinearEntity method), 455
- perpendicular_segment() (diofant.geometry.line3d.LinearEntity3D method), 474
- pexquo() (diofant.polys.polyclasses.DMP method), 779
- pexquo() (diofant.polys.polytools.Poly method), 716
- pexquo() (in module diofant.polys.polytools), 674
- pgroup (diofant.combinatorics.polyhedron.Polyhedron attribute), 218
- Pi (class in diofant.core.numbers), 98

Piecewise (class in diofant.functions.elementary.piecewise), 332

piecewise_fold() (in module diofant.functions.elementary.piecewise), 333

pinv() (diofant.matrices.matrices.MatrixBase method), 613

pinv_solve() (diofant.matrices.matrices.MatrixBase method), 613

Plane (class in diofant.geometry.plane), 519

Plot (class in diofant.plotting.plot), 886

plot() (in module diofant.plotting.plot), 889

plot3d() (in module diofant.plotting.plot), 893

plot3d_parametric_line() (in module diofant.plotting.plot), 894

plot3d_parametric_surface() (in module diofant.plotting.plot), 895

plot_implicit() (in module diofant.plotting.plot_implicit), 897

plot_interval() (diofant.geometry.curve.Curve method), 483

plot_interval() (diofant.geometry.ellipse.Ellipse method), 493

plot_interval() (diofant.geometry.line.Line method), 459

plot_interval() (diofant.geometry.line.Ray method), 462

plot_interval() (diofant.geometry.line.Segment method), 465

plot_interval() (diofant.geometry.line3d.Line3D method), 467

plot_interval() (diofant.geometry.line3d.Ray3D method), 477

plot_interval() (diofant.geometry.line3d.Segment3D method), 480

plot_interval() (diofant.geometry.polygon.Polygon method), 503

plot_parametric() (in module diofant.plotting.plot), 891

Point (class in diofant.diffgeom), 1180

Point (class in diofant.geometry.point), 438

point (diofant.core.function.Subs attribute), 137

point() (diofant.diffgeom.CoordSystem method), 1180

Point2D (class in diofant.geometry.point), 442

Point3D (class in diofant.geometry.point), 445

point_to_coords() (diofant.diffgeom.CoordSystem method), 1180

points (diofant.geometry.line.LinearEntity attribute), 456

points (diofant.geometry.line3d.LinearEntity3D attribute), 474

pointwise_stabilizer() (diofant.combinatorics.perm_groups.PermutationGroup method), 210

Poisson() (in module diofant.stats), 959

polar_lift (class in diofant.functions.elementary.complexes), 311

PoleError (class in diofant.core.function), 142

PolificationFailed (class in diofant.polys.polyerrors), 853

pollard_pm1() (in module diofant.ntheory.factor_), 259

pollard_rho() (in module diofant.ntheory.factor_), 258

Poly (class in diofant.polys.polytools), 690

poly() (in module diofant.polys.polytools), 672

poly_from_expr() (in module diofant.polys.polytools), 672

poly_ring() (diofant.polys.domains.domain.Domain method), 767

poly_unify() (diofant.polys.polyclasses.DMF method), 781

PolyElement (class in diofant.polys.rings), 828

polygamma (class in diofant.functions.special.gamma_functions), 361

Polygon (class in diofant.geometry.polygon), 498

Polyhedron (class in diofant.combinatorics.polyhedron), 216

polylog (class in diofant.functions.special.zeta_functions), 402

PolynomialError (class in diofant.polys.polyerrors), 853

PolynomialRing (class in diofant.polys.domains), 771

PolyRing (class in diofant.polys.rings), 827

pos() (diofant.polys.domains.domain.Domain method), 767

POSform() (in module diofant.logic.boolalg), 567

posify() (in module diofant.simplify.simplify), 927

`position_wrt()` (diofant.vector.coordsysrect.CoordSysCartesian method), 1214
`postfixes()` (in module diofant.utilities.iterables), 1156
`postorder_traversal()` (in module diofant.utilities.iterables), 1157
`Pow` (class in diofant.core.power), 100
`pow()` (diofant.polys.domains.domain.Domain method), 767
`pow()` (diofant.polys.polyclasses.ANP method), 781
`pow()` (diofant.polys.polyclasses.DMF method), 781
`pow()` (diofant.polys.polyclasses.DMP method), 779
`pow()` (diofant.polys.polytools.Poly method), 717
`powdenest()` (in module diofant.simplify.powsimp), 938
`powerset()` (diofant.sets.sets.Set method), 911
`powsimp()` (diofant.core.expr.Expr method), 78
`powsimp()` (in module diofant.simplify.powsimp), 937
`pprint()` (in module diofant.printing.pretty.pretty), 867
`pprint_nodes()` (in module diofant.printing.tree), 879
`PQa()` (in module diofant.solvers.diophantine), 1089
`pquo()` (diofant.polys.polyclasses.DMP method), 779
`pquo()` (diofant.polys.polytools.Poly method), 717
`pquo()` (in module diofant.polys.polytools), 674
`PRECEDENCE` (in module diofant.printing.precedence), 881
`precedence()` (in module diofant.printing.precedence), 881
`PRECEDENCE_FUNCTIONS` (in module diofant.printing.precedence), 881
`PRECEDENCE_VALUES` (in module diofant.printing.precedence), 881
`PrecisionExhausted` (class in diofant.core.evalf), 147
`preferred_index` (diofant.functions.special.tensor_functions.KroneckerDelta attribute), 429
`prefixes()` (in module diofant.utilities.iterables), 1157
`prem()` (diofant.polys.polyclasses.DMP method), 779
`prem()` (diofant.polys.polytools.Poly method), 717
`prem()` (in module diofant.polys.polytools), 674
`preorder_traversal` (class in diofant.core.basic), 57
`pretty()` (in module diofant.printing.pretty.pretty), 867
`pretty_atom()` (in module diofant.printing.pretty.pretty_symbology), 882
`pretty_print()` (in module diofant.printing.pretty.pretty), 867
`pretty_symbol()` (in module diofant.printing.pretty.pretty_symbology), 882
`pretty_try_use_unicode()` (in module diofant.printing.pretty.pretty_symbology), 881
`pretty_use_unicode()` (in module diofant.printing.pretty.pretty_symbology), 881
`prettyForm` (class in diofant.printing.pretty.stringpict), 884
`PrettyPrinter` (class in diofant.printing.pretty.pretty), 867
`prev()` (diofant.combinatorics.prufer.Prufer method), 221
`prev_binary()` (diofant.combinatorics.subsets.Subset method), 226
`prev_gray()` (diofant.combinatorics.subsets.Subset method), 226
`prev_lex()` (diofant.combinatorics.partitions.IntegerPartition method), 161
`prev_lexicographic()` (diofant.combinatorics.subsets.Subset method), 226
`prevprime()` (in module diofant.ntheory.generate), 252
`prime()` (in module diofant.ntheory.generate), 251
`prime_as_sum_of_two_squares()` (in module diofant.solvers.diophantine), 1092
`primefactors()` (in module diofant.ntheory.factor_), 263
`primepi()` (in module diofant.ntheory.generate), 252
`primrange()` (diofant.ntheory.generate.Sieve method), 251
`primrange()` (in module diofant.ntheory.generate), 253

- primitive() (diofant.core.add.Add method), 109
- primitive() (diofant.core.expr.Expr method), 78
- primitive() (diofant.polys.polyclasses.DMP method), 779
- primitive() (diofant.polys.polytools.Poly method), 718
- primitive() (diofant.polys.rings.PolyElement method), 832
- primitive() (in module diofant.polys.polytools), 683
- primitive_element() (in module diofant.polys.numberfields), 732
- primitive_root() (in module diofant.ntheory.residue_ntheory), 271
- primorial() (in module diofant.ntheory.generate), 254
- principal_branch (class in diofant.functions.elementary.complexes), 312
- print_ccode() (in module diofant.printing.ccode), 870
- print_fcode() (in module diofant.printing.fcode), 872
- print_latex() (in module diofant.printing.latex), 877
- print_mathml() (in module diofant.printing.mathml), 878
- print_node() (in module diofant.printing.tree), 879
- print_nonzero() (diofant.matrices.matrices.MatrixBase method), 614
- print_tree() (in module diofant.printing.tree), 879
- Printer (class in diofant.printing.printer), 865
- printmethod (diofant.printing.ccode.CCodePrinter attribute), 868
- printmethod (diofant.printing.codeprinter.CodePrinter attribute), 881
- printmethod (diofant.printing.fcode.FCodePrinter attribute), 872
- printmethod (diofant.printing.lambdarepr.LambdaPrinter attribute), 875
- printmethod (diofant.printing.latex.LatexPrinter attribute), 875
- printmethod (diofant.printing.mathematica.MCCodePrinter attribute), 874
- printmethod (diofant.printing.mathml.MathMLPrinter attribute), 877
- printmethod (diofant.printing.printer.Printer attribute), 866
- printmethod (diofant.printing.repr.ReprPrinter attribute), 878
- printmethod (diofant.printing.str.StrPrinter attribute), 879
- prod() (in module diofant.core.mul), 106
- Product (class in diofant.concrete.products), 284
- product() (diofant.polys.agca.ideals.Ideal method), 755
- product() (in module diofant.concrete.products), 294
- ProductDomain (class in diofant.stats.rv), 991
- ProductPSpace (class in diofant.stats.rv), 991
- ProductSet (class in diofant.sets.sets), 917
- project() (diofant.matrices.matrices.MatrixBase method), 615
- projection() (diofant.geometry.line.LinearEntity method), 456
- projection() (diofant.geometry.line3d.LinearEntity3D method), 475
- projection() (diofant.geometry.plane.Plane method), 524
- projection_line() (diofant.geometry.plane.Plane method), 525
- Prufer (class in diofant.combinatorics.prufer), 219
- prufer_rank() (diofant.combinatorics.prufer.Prufer method), 221
- prufer_repr (diofant.combinatorics.prufer.Prufer attribute), 221
- PSpace (class in diofant.stats.rv), 991
- pspace() (in module diofant.stats.rv), 992
- public() (in module diofant.utilities.decorator), 1137
- PurePoly (class in diofant.polys.polytools), 727
- Python Enhancement Proposals PEP 335, 1258-1260
- PythonFiniteField (class in diofant.polys.domains), 775
- PythonIntegerRing (class in diofant.polys.domains), 775
- PythonRationalField (class in diofant.polys.domains), 775
- QRdecomposition() (diofant.matrices.matrices.MatrixBase method), 589
- QRsolve() (diofant.matrices.matrices.MatrixBase method), 590

quadratic_residues() (in module diofant.ntheory.residue_ntheory), 272
 QuadraticU() (in module diofant.stats), 978
 QuaternionOrienter (class in diofant.vector.orienters), 1223
 quo() (diofant.polys.domains.domain.Domain method), 767
 quo() (diofant.polys.domains.field.Field method), 769
 quo() (diofant.polys.domains.ring.Ring method), 769
 quo() (diofant.polys.polyclasses.DMF method), 781
 quo() (diofant.polys.polyclasses.DMP method), 779
 quo() (diofant.polys.polytools.Poly method), 718
 quo() (in module diofant.polys.polytools), 675
 quo_ground() (diofant.polys.polyclasses.DMP method), 779
 quo_ground() (diofant.polys.polytools.Poly method), 718
 quotient() (diofant.polys.agca.ideals.Ideal method), 755
 quotient_codomain() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 762
 quotient_domain() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 763
 quotient_hom() (diofant.polys.agca.modules.QuotientModule method), 757
 quotient_hom() (diofant.polys.agca.modules.SubQuotientModule method), 758
 quotient_module() (diofant.polys.agca.modules.FreeModule method), 749
 quotient_module() (diofant.polys.agca.modules.Module method), 747
 quotient_module() (diofant.polys.agca.modules.SubModule method), 752
 quotient_ring() (diofant.polys.domains.ring.Ring method), 769
 QuotientModule (class in diofant.polys.agca.modules), 756
 QuotientModuleElement (class in diofant.polys.agca.modules), 758

R

rad_rationalize() (in module diofant.simplify.radsimp), 930
 radical() (diofant.polys.agca.ideals.Ideal method), 755
 radius (diofant.geometry.ellipse.Circle attribute), 497
 radius (diofant.geometry.polygon.RegularPolygon attribute), 510
 radius_of_convergence (diofant.functions.special.hyper.hyper attribute), 406
 radsimp() (diofant.core.expr.Expr method), 78
 radsimp() (in module diofant.simplify.radsimp), 929
 RaisedCosine() (in module diofant.stats), 979
 randMatrix() (in module diofant.matrices.dense), 625
 random() (diofant.combinatorics.perm_groups.Permutation method), 211
 random() (diofant.combinatorics.permutations.Permutation class method), 180
 random_bitstring() (diofant.combinatorics.graycode method), 233
 random_complex_number() (in module diofant.utilities.randtest), 1166
 random_integer_partition() (in module diofant.combinatorics.partitions), 161
 random_point() (diofant.geometry.ellipse.Ellipse method), 493
 random_point() (diofant.geometry.line.LinearEntity method), 457
 random_point() (diofant.geometry.plane.Plane method), 525
 random_poly() (in module diofant.polys.specialpolys), 735
 random_pr() (diofant.combinatorics.perm_groups.Permutation method), 211
 random_stab() (diofant.combinatorics.perm_groups.PermutationGroup method), 211
 random_symbols() (in module diofant.stats.rv), 992
 RandomDomain (class in diofant.stats.rv), 990
 RandomSymbol (class in diofant.stats.rv), 991
 randprime() (in module diofant.ntheory.generate), 254

Range (class in diofant.sets.fancysets), 921
 ranges (diofant.tensor.indexed.Indexed attribute), 1100
 rank (diofant.combinatorics.graycode.GrayCode attribute), 232
 rank (diofant.combinatorics.partitions.Partition attribute), 159
 rank (diofant.combinatorics.prufer.Prufer attribute), 222
 rank (diofant.tensor.indexed.Indexed attribute), 1101
 rank() (diofant.combinatorics.permutations.Permutation method), 180
 rank() (diofant.matrices.matrices.MatrixBase method), 615
 rank_binary (diofant.combinatorics.subsets.Subset attribute), 227
 rank_gray (diofant.combinatorics.subsets.Subset attribute), 227
 rank_lexicographic (diofant.combinatorics.subsets.Subset attribute), 227
 rank_nonlex() (diofant.combinatorics.permutations.Permutation method), 181
 rank_trotterjohnson() (diofant.combinatorics.permutations.Permutation method), 181
 rat_clear_denoms() (diofant.polys.polytools.Poly method), 718
 ratint() (in module diofant.integrals.rationaltools), 550
 ratint_logpart() (in module diofant.integrals.rationaltools), 550
 ratint_ratpart() (in module diofant.integrals.rationaltools), 551
 Rational (class in diofant.core.numbers), 89
 RationalField (class in diofant.polys.domains), 772
 rationalize() (in module diofant.parsing.sympy_parser), 1170
 Rationals (class in diofant.sets.fancysets), 921
 ratsimp() (diofant.core.expr.Expr method), 78
 ratsimp() (in module diofant.simplify.ratsimp), 935
 Ray (class in diofant.geometry.line), 460
 Ray3D (class in diofant.geometry.line3d), 475
 Rayleigh() (in module diofant.stats), 980
 rcall() (diofant.core.basic.Basic method), 51
 rcollect() (in module diofant.simplify.radsimp), 933
 re (class in diofant.functions.elementary.complexes), 307
 real_root() (in module diofant.functions.elementary.miscellaneous), 337
 real_roots() (diofant.polys.polytools.Poly method), 719
 real_roots() (in module diofant.polys.polytools), 687
 RealField (class in diofant.polys.domains), 773
 RealNumber (in module diofant.core.numbers), 93
 reconstruct() (in module diofant.solvers.diophantine), 1093
 recurrence_memo() (in module diofant.utilities.memoization), 1166
 reduced_groebner() (in module diofant.polys.groebnertools), 851
 reduce() (diofant.polys.polytools.GroebnerBasis method), 729
 reduce() (diofant.sets.sets.Complement static method), 918
 reduce() (diofant.sets.sets.Intersection static method), 917
 reduce() (diofant.sets.sets.Union static method), 916
 reduce_element() (diofant.polys.agca.ideals.Ideal method), 755
 reduce_element() (diofant.polys.agca.modules.SubModule method), 752
 reduce_inequalities() (in module diofant.solvers.inequalities), 1096
 reduce_piecewise_inequalities() (in module diofant.solvers.inequalities), 1095
 reduce_piecewise_inequality() (in module diofant.solvers.inequalities), 1095
 reduce_rational_inequalities() (in module diofant.solvers.inequalities), 1094
 reduced() (in module diofant.polys.polytools), 689
 refine_root() (diofant.polys.polyclasses.DMP method), 779
 refine_root() (diofant.polys.polytools.Poly method), 719
 refine_root() (in module diofant.polys.polytools), 687
 RefinementFailed (class in diofant.polys.polyerrors), 853
 reflect() (diofant.geometry.ellipse.Circle method), 497

reflect() (diofant.geometry.ellipse.Ellipse method), 494
 reflect() (diofant.geometry.polygon.RegularPolygon method), 511
 RegularPolygon (class in diofant.geometry.polygon), 505
 Rel (in module diofant.core.relational), 110
 Relational (class in diofant.core.relational), 111
 rem() (diofant.polys.domains.domain.Domain method), 767
 rem() (diofant.polys.domains.field.Field method), 769
 rem() (diofant.polys.domains.ring.Ring method), 770
 rem() (diofant.polys.polyclasses.DMP method), 779
 rem() (diofant.polys.polytools.Poly method), 719
 rem() (in module diofant.polys.polytools), 675
 removeO() (diofant.core.add.Add method), 109
 removeO() (diofant.core.expr.Expr method), 78
 render() (diofant.printing.pretty.stringpict.stringPict method), 883
 reorder() (diofant.concrete.expr_with_intlimits.ExprWithIntLimits method), 292
 reorder() (diofant.polys.polytools.Poly method), 720
 reorder_limit() (diofant.concrete.expr_with_intlimits.ExprWithIntLimits method), 293
 replace() (diofant.core.basic.Basic method), 52
 replace() (diofant.matrices.matrices.MatrixBase method), 615
 replace() (diofant.polys.polytools.Poly method), 720
 reprify() (diofant.printing.repr.ReprPrinter method), 878
 ReprPrinter (class in diofant.printing.repr), 878
 reset() (diofant.combinatorics.polyhedron.Polyhedron method), 218
 reshape() (diofant.matrices.dense.DenseMatrix method), 632
 reshape() (diofant.matrices.sparse.SparseMatrix method), 648
 reshape() (in module diofant.utilities.iterables), 1157
 residue() (in module diofant.series.residues), 904
 restrict_codomain() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 763
 restrict_domain() (diofant.polys.agca.homomorphisms.ModuleHomomorphism method), 763
 Result (class in diofant.utilities.codegen), 1129
 result_variables (diofant.utilities.codegen.Routine attribute), 1129
 resultant() (diofant.polys.polyclasses.DMP method), 779
 resultant() (diofant.polys.polytools.Poly method), 720
 resultant() (in module diofant.polys.polytools), 677
 retract() (diofant.polys.polytools.Poly method), 720
 reverse_order() (diofant.concrete.products.Product method), 287
 reverse_order() (diofant.concrete.summations.Sum method), 283
 reversed (diofant.core.relational.Relational attribute), 112
 ReversedGradedLexOrder (class in diofant.polys.orderings), 734
 revert() (diofant.polys.domains.domain.Domain method), 767
 revert() (diofant.polys.domains.field.Field method), 769
 revert() (diofant.polys.domains.ring.Ring method), 770
 revert() (diofant.polys.polyclasses.DMP method), 779
 revert() (diofant.polys.polytools.Poly method), 721
 rewrite() (diofant.core.basic.Basic method), 54
 rewrite() (in module diofant.series.gruntz), 907
 RGS (in module diofant.combinatorics.partitions.Partition attribute), 158
 RGS_enum() (in module diofant.combinatorics.partitions), 162
 RGS_generalized() (in module diofant.combinatorics.partitions), 161
 RGS_rank() (in module diofant.combinatorics.partitions), 162
 RGS_unrank() (in module diofant.combinatorics.partitions), 162

rhs (diofant.core.relational.Relational attribute), 112
 riemann_cyclic() (in module diofant.tensor.tensor), 1121
 riemann_cyclic_replace() (in module diofant.tensor.tensor), 1121
 right (diofant.sets.sets.Interval attribute), 914
 right() (diofant.printing.pretty.stringpict.stringPict method), 883
 right_open (diofant.sets.sets.Interval attribute), 915
 Ring (class in diofant.polys.domains.ring), 769
 ring() (in module diofant.polys.rings), 826
 RisingFactorial (class in diofant.functions.combinatorial.factorials), 352
 RL (diofant.matrices.sparse.SparseMatrixBase attribute), 643
 rmul() (diofant.combinatorics.permutations.Permutation static method), 181
 rmul_with_af() (diofant.combinatorics.permutations.Permutation static method), 182
 root (in module diofant.printing.pretty.pretty_symbology), 882
 root() (diofant.polys.polytools.Poly method), 721
 root() (in module diofant.functions.elementary.miscellaneous), 335
 RootOf (class in diofant.polys.rootoftools), 734
 roots() (in module diofant.polys.polyroots), 734
 RootSum (class in diofant.polys.rootoftools), 734
 Ropen() (diofant.sets.sets.Interval class method), 914
 rot_axis1() (in module diofant.matrices.dense), 627
 rot_axis2() (in module diofant.matrices.dense), 628
 rot_axis3() (in module diofant.matrices.dense), 629
 rotate() (diofant.combinatorics.polyhedron.Polyhedron method), 218
 rotate() (diofant.geometry.curve.Curve method), 483
 rotate() (diofant.geometry.ellipse.Ellipse method), 494
 rotate() (diofant.geometry.entity.GeometryEntity method), 434
 rotate() (diofant.geometry.point.Point2D method), 443
 rotate() (diofant.geometry.polygon.RegularPolygon method), 511
 rotate_left() (in module diofant.utilities.iterables), 1158
 rotate_right() (in module diofant.utilities.iterables), 1158
 rotation (diofant.geometry.polygon.RegularPolygon attribute), 511
 rotation_matrix() (diofant.vector.coordsysrect.CoordSysCartesian method), 1214
 rotation_matrix() (diofant.vector.orienters.AxisOrienter method), 1221
 rotation_matrix() (diofant.vector.orienters.Orienter method), 1220
 round() (diofant.core.expr.Expr method), 78
 RoundFunction (class in diofant.functions.elementary.integers), 329
 Routine (class in diofant.utilities.codegen), 1129
 routine() (diofant.utilities.codegen.CodeGen method), 1130
 routine() (diofant.utilities.codegen.OctaveCodeGen method), 1133
 row() (diofant.matrices.dense.DenseMatrix method), 632
 row() (diofant.matrices.sparse.SparseMatrixBase method), 648
 row_del() (diofant.matrices.dense.MutableDenseMatrix method), 635
 row_del() (diofant.matrices.sparse.MutableSparseMatrix method), 641
 row_insert() (diofant.matrices.matrices.MatrixBase method), 616
 row_join() (diofant.matrices.matrices.MatrixBase method), 616
 row_join() (diofant.matrices.sparse.MutableSparseMatrix method), 641
 row_list() (diofant.matrices.sparse.SparseMatrixBase method), 648
 row_op() (diofant.matrices.dense.MutableDenseMatrix method), 635
 row_op() (diofant.matrices.sparse.MutableSparseMatrix method), 642
 row_structure_symbolic_cholesky() (diofant.matrices.sparse.SparseMatrixBase method), 649

- row_swap() (diofant.matrices.dense.MutableDenseMatrixE() (in module diofant.stats.rv), 992 method), 636
 - row_swap() (diofant.matrices.sparse.MutableSparseMatrixE() (in module diofant.logic.inference), 576 method), 642
 - rref() (diofant.matrices.matrices.MatrixBase method), 616
 - rs_compose_add() (in module diofant.polys.ring_series), 863
 - rs_exp() (in module diofant.polys.ring_series), 862
 - rs_hadamard_exp() (in module diofant.polys.ring_series), 862
 - rs_integrate() (in module diofant.polys.ring_series), 861
 - rs_log() (in module diofant.polys.ring_series), 861
 - rs_mul() (in module diofant.polys.ring_series), 860
 - rs_newton() (in module diofant.polys.ring_series), 862
 - rs_pow() (in module diofant.polys.ring_series), 860
 - rs_series_from_list() (in module diofant.polys.ring_series), 861
 - rs_series_inversion() (in module diofant.polys.ring_series), 860
 - rs_square() (in module diofant.polys.ring_series), 860
 - rs_swap() (in module diofant.stats.rv), 992
 - rs_trunc() (in module diofant.polys.ring_series), 859
 - rsolve() (in module diofant.solvers.recurr), 1068
 - rsolve_hyper() (in module diofant.solvers.recurr), 1070
 - rsolve_poly() (in module diofant.solvers.recurr), 1069
 - rsolve_ratio() (in module diofant.solvers.recurr), 1069
 - runs() (diofant.combinatorics.permutations.PermutationGroup method), 182
 - runs() (in module diofant.utilities.iterables), 1159
- S**
- S (in module diofant.core.singleton), 58
 - sample() (in module diofant.stats), 990
 - sample_iter() (in module diofant.stats), 990
 - sample_iter_lambdify() (in module diofant.stats.rv), 992
 - sample_iter_subs() (in module diofant.stats.rv), 992
 - sampling_density() (in module diofant.stats.rv), 992
 - sample_MLE() (in module diofant.stats.rv), 992
 - sampling_P() (in module diofant.stats.rv), 992
 - satisfiable() (in module diofant.logic.inference), 576
 - saturate() (diofant.polys.agca.ideals.Ideal method), 755
 - scalar_map() (diofant.vector.coordsysrect.CoordSysCartesian method), 1215
 - scalar_multiply() (diofant.matrices.sparse.SparseMatrixBase method), 649
 - scalar_potential() (in module diofant.vector), 1227
 - scalar_potential_difference() (in module diofant.vector), 1227
 - scale() (diofant.geometry.curve.Curve method), 484
 - scale() (diofant.geometry.ellipse.Circle method), 498
 - scale() (diofant.geometry.ellipse.Ellipse method), 494
 - scale() (diofant.geometry.entity.GeometryEntity method), 434
 - scale() (diofant.geometry.point.Point2D method), 444
 - scale() (diofant.geometry.point.Point3D method), 448
 - scale() (diofant.geometry.polygon.RegularPolygon method), 511
 - schreier_sims() (diofant.combinatorics.perm_groups.PermutationGroup method), 211
 - schreier_sims_incremental() (diofant.combinatorics.perm_groups.PermutationGroup method), 212
 - schreier_sims_random() (diofant.combinatorics.perm_groups.PermutationGroup method), 212
 - schreier_vector() (diofant.combinatorics.perm_groups.PermutationGroup method), 213
 - sdm_add() (in module diofant.polys.distributedmodules), 835
 - sdm_deg() (in module diofant.polys.distributedmodules), 836
 - sdm_ecart() (in module diofant.polys.distributedmodules), 852
 - sdm_from_dict() (in module diofant.polys.distributedmodules), 834
 - sdm_from_vector() (in module diofant.polys.distributedmodules), 837
 - sdm_groebner() (in module diofant.polys.distributedmodules), 852

sdm_LC() (in module diofant.polys.distributedmodules), 834
 sdm_LM() (in module diofant.polys.distributedmodules), 835
 sdm_LT() (in module diofant.polys.distributedmodules), 835
 sdm_monomial_deg() (in module diofant.polys.distributedmodules), 833
 sdm_monomial_divides() (in module diofant.polys.distributedmodules), 834
 sdm_monomial_mul() (in module diofant.polys.distributedmodules), 833
 sdm_mul_term() (in module diofant.polys.distributedmodules), 836
 sdm_nf_mora() (in module diofant.polys.distributedmodules), 852
 sdm_spoly() (in module diofant.polys.distributedmodules), 851
 sdm_to_dict() (in module diofant.polys.distributedmodules), 834
 sdm_to_vector() (in module diofant.polys.distributedmodules), 837
 sdm_zero() (in module diofant.polys.distributedmodules), 836
 search() (diofant.ntheory.generate.Sieve method), 251
 sec (class in diofant.functions.elementary.trigonometric), 316
 sech (class in diofant.functions.elementary.hyperbolic), 326
 Segment (class in diofant.geometry.line), 463
 Segment3D (class in diofant.geometry.line3d), 478
 select() (diofant.simplify.epathtools.EPath method), 945
 selections (diofant.combinatorics.graycode.GrayCode attribute), 232
 separate() (diofant.vector.vector.Vector method), 1217
 separatevars() (in module diofant.simplify.simplify), 924
 series() (diofant.core.expr.Expr method), 79
 series() (in module diofant.series.series), 903
 Set (class in diofant.sets.sets), 908
 set_comm() (diofant.tensor.tensor._TensorManager method), 1106
 set_comms() (diofant.tensor.tensor._TensorManager method), 1107
 set_domain() (diofant.polys.polytools.Poly method), 721
 set_global_settings() (diofant.printing.printer.Printer class method), 867
 set_modulus() (diofant.polys.polytools.Poly method), 721
 seterr() (in module diofant.core.numbers), 93
 setup() (in module diofant.polys.polyconfig), 864
 shape (diofant.matrices.matrices.MatrixBase attribute), 617
 shape (diofant.tensor.indexed.Indexed attribute), 1101
 shape (diofant.tensor.indexed.IndexedBase attribute), 1103
 ShapeError (class in diofant.matrices.matrices), 621
 Shi (class in diofant.functions.special.error_functions), 385
 shift() (diofant.polys.polyclasses.DMP method), 779
 shift() (diofant.polys.polytools.Poly method), 722
 Si (class in diofant.functions.special.error_functions), 383
 sides (diofant.geometry.polygon.Polygon attribute), 504
 Sieve (class in diofant.ntheory.generate), 250
 sift() (in module diofant.utilities.iterables), 1159
 sign (class in diofant.functions.elementary.complexes), 308
 sign() (in module diofant.series.gruntz), 908
 signature() (diofant.combinatorics.permutations.Permutation method), 182
 SimpleDomain (class in diofant.polys.domains.simpledomain), 770
 simplified() (in module diofant.solvers.diophantine), 1090
 simplify() (diofant.core.expr.Expr method), 80
 simplify() (diofant.functions.special.delta_functions.DiracDelta method), 358
 simplify() (diofant.matrices.dense.MutableDenseMatrix method), 636
 simplify() (diofant.matrices.matrices.MatrixBase method), 617
 simplify() (in module diofant.simplify.simplify), 922
 simplify_logic() (in module diofant.logic.boolalg), 575
 sin (class in diofant.functions.elementary.trigonometric), 312

- sine_transform() (in module diofant.integrals.transforms), 529
- SineTransform (class in diofant.integrals.transforms), 558
- SingleDomain (class in diofant.stats.rv), 991
- SinglePSpace (class in diofant.stats.rv), 991
- singular_values() (diofant.matrices.matrices.MatrixBase method), 617
- singularities() (in module diofant.calculus.singularities), 1171
- sinh (class in diofant.functions.elementary.hyperbolic), 325
- size (diofant.combinatorics.permutations.Permutation attribute), 183
- size (diofant.combinatorics.polyhedron.Polyhedron attribute), 219
- size (diofant.combinatorics.prufer.Prufer attribute), 222
- size (diofant.combinatorics.subsets.Subset attribute), 227
- skip() (diofant.combinatorics.graycode.GrayCode method), 232
- skip() (diofant.core.basic.preorder_traversal method), 58
- slice() (diofant.polys.polyclasses.DMP method), 779
- slice() (diofant.polys.polytools.Poly method), 722
- slope (diofant.geometry.line.LinearEntity attribute), 457
- smoothness() (in module diofant.ntheory.factor_), 256
- smoothness_p() (in module diofant.ntheory.factor_), 256
- solve() (diofant.matrices.matrices.MatrixBase method), 617
- solve() (diofant.matrices.sparse.SparseMatrixBase method), 649
- solve() (in module diofant.solvers.solvers), 1054
- solve_congruence() (in module diofant.ntheory.modular), 267
- solve_least_squares() (diofant.matrices.matrices.MatrixBase method), 618
- solve_least_squares() (diofant.matrices.sparse.SparseMatrixBase method), 649
- solve_linear() (in module diofant.solvers.solvers), 1061
- solve_linear_system() (in module diofant.solvers.solvers), 1063
- solve_poly_inequalities() (in module diofant.solvers.inequalities), 1094
- solve_poly_inequality() (in module diofant.solvers.inequalities), 1094
- solve_poly_system() (in module diofant.solvers.polysys), 1071
- solve_rational_inequalities() (in module diofant.solvers.inequalities), 1093
- solve_triangulated() (in module diofant.solvers.polysys), 1071
- solve_undetermined_coeffs() (in module diofant.solvers.solvers), 1063
- solve_univariate_inequality() (in module diofant.solvers.inequalities), 1096
- sort_key() (in module diofant.logic.boolalg), 566
- sort_key() (diofant.combinatorics.partitions.Partition method), 159
- sort_key() (diofant.core.basic.Atom method), 41
- sort_key() (diofant.core.basic.Basic method), 54
- sort_key() (diofant.core.expr.Expr method), 80
- sort_key() (diofant.core.numbers.Number method), 86
- sort_key() (diofant.core.symbol.Dummy method), 83
- sorted_components() (diofant.tensor.tensor.TensMul method), 1120
- source (diofant.geometry.line.Ray attribute), 462
- source (diofant.geometry.line3d.Ray3D attribute), 477
- SpaceOrienter (class in diofant.vector.orienters), 1222
- SparseMatrix (in module diofant.matrices.sparse), 643
- SparseMatrixBase (class in diofant.matrices.sparse), 643
- spherical_bessel_fn() (in module diofant.polys.orthopolys), 736
- spin() (diofant.geometry.polygon.RegularPolygon method), 512
- split() (diofant.tensor.tensor.TensMul method), 1120
- split_super_sub() (in module diofant.printing.conventions), 880
- split_symbols() (in module diofant.parsing.sympy_parser), 1168
- split_symbols_custom() (in module diofant.parsing.sympy_parser), 1169

- `spoly()` (in module `diofant.polys.groebnertools`), 851
`sqf()` (in module `diofant.polys.polytools`), 685
`sqf_list()` (`diofant.polys.polyclasses.DMP` method), 779
`sqf_list()` (`diofant.polys.polytools.Poly` method), 722
`sqf_list()` (in module `diofant.polys.polytools`), 685
`sqf_list_include()` (`diofant.polys.polyclasses.DMP` method), 780
`sqf_list_include()` (`diofant.polys.polytools.Poly` method), 722
`sqf_norm()` (`diofant.polys.polyclasses.DMP` method), 780
`sqf_norm()` (`diofant.polys.polytools.Poly` method), 723
`sqf_norm()` (in module `diofant.polys.polytools`), 684
`sqf_part()` (`diofant.polys.polyclasses.DMP` method), 780
`sqf_part()` (`diofant.polys.polytools.Poly` method), 723
`sqf_part()` (in module `diofant.polys.polytools`), 684
`sqr()` (`diofant.polys.polyclasses.DMP` method), 780
`sqr()` (`diofant.polys.polytools.Poly` method), 723
`sqrt()` (`diofant.polys.domains.domain.Domain` method), 767
`sqrt()` (in module `diofant.functions.elementary.miscellaneous`), 338
`sqrt_mod()` (in module `diofant.ntheory.residue_ntheory`), 271
`sqrtdenest()` (in module `diofant.simplify.sqrtdenest`), 940
`square()` (`diofant.polys.rings.PolyElement` method), 832
`square_factor()` (in module `diofant.solvers.diophantine`), 1084
`srepr()` (in module `diofant.printing.repr`), 878
`sring()` (in module `diofant.polys.rings`), 827
`sstr()` (in module `diofant.printing.str`), 879
`sstrrepr()` (in module `diofant.printing.str`), 879
`stabilizer()` (`diofant.combinatorics.perm_groups.PermutationGroup` class method), 214
`stack()` (`diofant.printing.pretty.stringpict.stringPict` static method), 883
`standard_transformations` (in module `diofant.parsing.sympy_parser`), 1168
`start` (`diofant.sets.sets.Interval` attribute), 915
`std()` (in module `diofant.stats`), 989
`stirling()` (in module `diofant.functions.combinatorial.numbers`), 352
`StrictGreaterThan` (class in `diofant.core.relational`), 120
`StrictLessThan` (class in `diofant.core.relational`), 123
`stringify_expr()` (in module `diofant.parsing.sympy_parser`), 1168
`stringPict` (class in `diofant.printing.pretty.stringpict`), 882
`strip_zero()` (`diofant.polys.rings.PolyElement` method), 833
`strong_gens` (`diofant.combinatorics.perm_groups.PermutationGroup` attribute), 214
`StrPrinter` (class in `diofant.printing.str`), 878
`StudentT()` (in module `diofant.stats`), 980
`sturm()` (`diofant.polys.polyclasses.DMP` method), 780
`sturm()` (`diofant.polys.polytools.Poly` method), 724
`sturm()` (in module `diofant.polys.polytools`), 684
`sub` (in module `diofant.printing.pretty.pretty_symbology`), 882
`sub()` (`diofant.polys.domains.domain.Domain` method), 768
`sub()` (`diofant.polys.polyclasses.DMF` method), 781
`sub()` (`diofant.polys.polyclasses.DMP` method), 780
`sub()` (`diofant.polys.polytools.Poly` method), 724
`sub_ground()` (`diofant.polys.polyclasses.DMP` method), 780
`sub_ground()` (`diofant.polys.polytools.Poly` method), 724
`subfactorial` (class in `diofant.functions.combinatorial.factorials`), 346
`subgroup_search()` (`diofant.combinatorics.perm_groups.PermutationGroup` method), 215
`SubModule` (class in `diofant.polys.agca.modules`), 749
`Submodule()` (`diofant.polys.agca.modules.Module` method), 748

- submodule() (diofant.polys.agca.modules.QuotientModule attribute), 229
 - submodule() (diofant.polys.agca.modules.SubModule method), 757
 - SubQuotientModule (class in diofant.polys.agca.modules), 757
 - subresultants() (diofant.polys.polyclasses.DMP method), 780
 - subresultants() (diofant.polys.polytools.Poly method), 724
 - subresultants() (in module diofant.polys.polytools), 677
 - Subs (class in diofant.core.function), 136
 - subs() (diofant.core.basic.Basic method), 54
 - subs() (diofant.matrices.immutable.ImmutableSparseMatrix method), 652
 - subs() (diofant.matrices.matrices.MatrixBase method), 618
 - Subset (class in diofant.combinatorics.subsets), 223
 - subset (diofant.combinatorics.subsets.Subset attribute), 228
 - subset() (diofant.polys.agca.ideals.Ideal method), 755
 - subset() (diofant.polys.agca.modules.Module method), 748
 - subset_from_bitlist() (diofant.combinatorics.subsets.Subset class method), 228
 - subset_indices() (diofant.combinatorics.subsets.Subset class method), 228
 - subsets() (in module diofant.utilities.iterables), 1159
 - substitute_indices() (diofant.tensor.tensor.TensAdd method), 1118
 - Sum (class in diofant.concrete.summations), 280
 - sum_of_four_squares() (in module diofant.solvers.diophantine), 1088
 - sum_of_three_squares() (in module diofant.solvers.diophantine), 1087
 - summation() (in module diofant.concrete.summations), 293
 - sup (diofant.sets.sets.Set attribute), 912
 - sup (in module diofant.printing.pretty.pretty_symbology), 882
 - superset (diofant.combinatorics.subsets.Subset attribute), 229
 - superset_size (diofant.combinatorics.subsets.Subset attribute), 229
 - support() (diofant.combinatorics.permutations.Permutation method), 183
 - SurfaceBaseSeries (class in diofant.plotting.plot), 900
 - SurfaceOver2DRangeSeries (class in diofant.plotting.plot), 900
 - swinnerton_dyer_poly() (in module diofant.polys.specialpolys), 735
 - symarray() (in module diofant.matrices.dense), 626
 - symb_2txt (in module diofant.printing.pretty.pretty_symbology), 881
 - Symbol (class in diofant.core.symbol), 81
 - SymbolSet (in module diofant.core.symbol), 83
 - symmetric() (diofant.combinatorics.generators method), 186
 - symmetric_poly() (in module diofant.polys.specialpolys), 735
 - symmetric_residue() (in module diofant.ntheory.modular), 266
 - SymmetricGroup() (in module diofant.combinatorics.named_groups), 235
 - symmetrize() (in module diofant.polys.polyfuncs), 729
 - sympify() (in module diofant.core.sympify), 37
 - syzygy_module() (diofant.polys.agca.modules.SubModule method), 753
- ## T
- T (diofant.matrices.expressions.MatrixExpr attribute), 655
 - T (diofant.matrices.matrices.MatrixBase attribute), 590
 - table() (diofant.matrices.matrices.MatrixBase method), 619
 - tail_degree() (diofant.polys.rings.PolyElement method), 833
 - tail_degrees() (diofant.polys.rings.PolyElement method), 833
 - take() (in module diofant.utilities.iterables), 1160
 - tan (class in diofant.functions.elementary.trigonometric), 314
 - tangent_lines() (diofant.geometry.ellipse.Ellipse method), 495

tanh (class in diofant.functions.elementary.hyperbolic), 326
 taylor_term() (diofant.core.expr.Expr method), 80
 taylor_term() (diofant.functions.elementary.hyperbolic.csch static method), 327
 taylor_term() (diofant.functions.elementary.hyperbolic.sinh static method), 326
 TC() (diofant.polys.polyclasses.ANP method), 781
 TC() (diofant.polys.polyclasses.DMP method), 775
 TC() (diofant.polys.polytools.Poly method), 691
 TensAdd (class in diofant.tensor.tensor), 1116
 TensExpr (class in diofant.tensor.tensor), 1115
 TensMul (class in diofant.tensor.tensor), 1118
 tensor_indices() (in module diofant.tensor.tensor), 1110
 tensor_mul() (in module diofant.tensor.tensor), 1121
 TensorHead (class in diofant.tensor.tensor), 1112
 TensorIndex (class in diofant.tensor.tensor), 1109
 TensorIndexType (class in diofant.tensor.tensor), 1107
 TensorProduct (class in diofant.diffgeom), 1184
 TensorSymmetry (class in diofant.tensor.tensor), 1110
 tensorsymmetry() (in module diofant.tensor.tensor), 1111
 TensorType (class in diofant.tensor.tensor), 1111
 terminal_width() (diofant.printing.pretty.stringpict.stringPict method), 884
 terms() (diofant.polys.polyclasses.DMP method), 780
 terms() (diofant.polys.polytools.Poly method), 725
 terms() (diofant.polys.rings.PolyElement method), 833
 terms_gcd() (diofant.polys.polyclasses.DMP method), 780
 terms_gcd() (diofant.polys.polytools.Poly method), 725
 terms_gcd() (in module diofant.polys.polytools), 680
 termwise() (diofant.polys.polytools.Poly method), 725
 test_derivative_numerically() (in module diofant.utilities.randtest), 1166
 to_algebraic_integer() (diofant.polys.numberfields.AlgebraicNumber method), 733
 to_cnf() (in module diofant.logic.boolalg), 574
 to_dict() (diofant.polys.polyclasses.ANP method), 781
 to_dict() (diofant.polys.polyclasses.DMP method), 780
 to_diofant() (diofant.polys.domains.AlgebraicField method), 773
 to_diofant() (diofant.polys.domains.domain.Domain method), 768
 to_diofant() (diofant.polys.domains.ExpressionDomain method), 775
 to_diofant() (diofant.polys.domains.FiniteField method), 771
 to_diofant() (diofant.polys.domains.FractionField method), 773
 to_diofant() (diofant.polys.domains.PolynomialRing method), 772
 to_diofant() (diofant.polys.domains.RealField method), 774
 to_diofant_dict() (diofant.polys.polyclasses.ANP method), 782
 to_diofant_dict() (diofant.polys.polyclasses.DMP method), 780
 to_diofant_list() (diofant.polys.polyclasses.ANP method), 782
 to_dnf() (in module diofant.logic.boolalg), 574
 to_exact() (diofant.polys.polyclasses.DMP method), 780
 to_exact() (diofant.polys.polytools.Poly method), 725
 to_field() (diofant.polys.polyclasses.DMP method), 780
 to_field() (diofant.polys.polytools.Poly method), 726
 to_list() (diofant.polys.polyclasses.ANP method), 782
 to_matrix() (diofant.vector.dyadic.Dyadic method), 1218
 to_matrix() (diofant.vector.vector.Vector method), 1217
 to_number_field() (in module diofant.polys.numberfields), 732
 to_prufer() (diofant.combinatorics.prufer.Prufer static method), 222

- to_rational() (diofant.polys.domains.RealField method), 774
- to_ring() (diofant.polys.polyclasses.DMP method), 780
- to_ring() (diofant.polys.polytools.Poly method), 726
- to_tree() (diofant.combinatorics.prufer.Prufer static method), 222
- to_tuple() (diofant.polys.polyclasses.ANP method), 782
- to_tuple() (diofant.polys.polyclasses.DMP method), 780
- together() (diofant.core.expr.Expr method), 80
- together() (in module diofant.polys.rationaltools), 736
- tolist() (diofant.matrices.dense.DenseMatrix method), 632
- tolist() (diofant.matrices.sparse.SparseMatrixBase method), 650
- topological_sort() (in module diofant.utilities.iterables), 1160
- total_degree() (diofant.polys.polyclasses.DMP method), 780
- total_degree() (diofant.polys.polytools.Poly method), 726
- totient() (in module diofant.ntheory.factor_), 265
- Trace (class in diofant.matrices.expressions), 657
- trailing() (in module diofant.ntheory.factor_), 257
- transform() (diofant.geometry.point.Point2D method), 444
- transform() (diofant.geometry.point.Point3D method), 448
- transform() (diofant.integrals.integrals.Integral method), 555
- transform_variable (diofant.integrals.transforms.IntegralTransform attribute), 557
- transformation_to_DN() (in module diofant.solvers.diophantine), 1082
- transformation_to_normal() (in module diofant.solvers.diophantine), 1093
- transitivity_degree (diofant.combinatorics.perm_groups.PermutationGroup attribute), 216
- translate() (diofant.geometry.curve.Curve method), 484
- translate() (diofant.geometry.entity.GeometryEntity method), 434
- translate() (diofant.geometry.point.Point2D method), 444
- translate() (diofant.geometry.point.Point3D method), 448
- transpose (class in diofant.functions.elementary.complexes), 312
- Transpose (class in diofant.matrices.expressions), 657
- transpose() (diofant.core.expr.Expr method), 80
- transpose() (diofant.matrices.expressions.blockmatrix.BlockMatrix method), 659
- transpose() (diofant.matrices.matrices.MatrixBase method), 619
- transpositions() (diofant.combinatorics.permutations.Permutation method), 183
- tree() (in module diofant.printing.tree), 879
- tree_cse() (in module diofant.simplify.cse_main), 943
- tree_repr (diofant.combinatorics.prufer.Prufer attribute), 223
- Triangle (class in diofant.geometry.polygon), 512
- Triangular() (in module diofant.stats), 981
- trigamma() (in module diofant.functions.special.gamma_functions), 364
- trigintegrate() (in module diofant.integrals.trigonometry), 553
- trigsimp() (diofant.core.expr.Expr method), 80
- trigsimp() (in module diofant.simplify.trigsimp), 936
- trunc() (diofant.polys.polyclasses.DMP method), 780
- trunc() (diofant.polys.polytools.Poly method), 726
- trunc() (in module diofant.polys.polytools), 682
- Tuple (class in diofant.core.containers), 148
- tuple_count() (diofant.core.containers.Tuple method), 148
- twoform_to_matrix() (in module diofant.diffgeom), 1190
- U**
- U() (in module diofant.printing.pretty.pretty_symbology), 881
- unify() (in module diofant.utilities.autowrap), 1125

- UfuncifyCodeWrapper (class in diofant.utilities.autowrap), 1124
- Unequality (class in diofant.core.relational), 120
- unflatten() (in module diofant.utilities.iterables), 1161
- UnificationFailed (class in diofant.polys.polyerrors), 853
- Uniform() (in module diofant.stats), 982
- UniformSum() (in module diofant.stats), 983
- unify() (diofant.polys.domains.domain.Domain method), 768
- unify() (diofant.polys.polyclasses.ANP method), 782
- unify() (diofant.polys.polyclasses.DMP method), 780
- unify() (diofant.polys.polytools.Poly method), 727
- Union (class in diofant.sets.sets), 916
- union() (diofant.polys.agca.ideals.Ideal method), 756
- union() (diofant.polys.agca.modules.SubModule method), 753
- union() (diofant.sets.sets.Set method), 912
- uniq() (in module diofant.utilities.iterables), 1161
- unit (diofant.polys.polytools.Poly attribute), 727
- UnivariatePolynomialError (class in diofant.polys.polyerrors), 853
- UniversalSet (class in diofant.sets.sets), 919
- unrad() (in module diofant.solvers.solvers), 1066
- unrank() (diofant.combinatorics.graycode.GrayCode class method), 233
- unrank() (diofant.combinatorics.prufer.Prufer class method), 223
- unrank_binary() (diofant.combinatorics.subsets.Subset class method), 229
- unrank_gray() (diofant.combinatorics.subsets.Subset class method), 229
- unrank_lex() (diofant.combinatorics.permutations.Permutation class method), 184
- unrank_nonlex() (diofant.combinatorics.permutations.Permutation class method), 184
- unrank_trotterjohnson() (diofant.combinatorics.permutations.Permutation class method), 184
- upper (diofant.tensor.indexed.Idx attribute), 1099
- upper_triangular_solve() (diofant.matrices.matrices.MatrixBase method), 619
- uppergamma (class in diofant.functions.special.gamma_functions), 364
- use() (in module diofant.simplify.traversaltools), 944
- ## V
- values() (diofant.core.containers.Dict method), 149
- values() (diofant.matrices.matrices.MatrixBase method), 620
- var() (in module diofant.core.symbol), 85
- variables (diofant.concrete.expr_with_limits.ExprWithLimits attribute), 289
- variables (diofant.core.function.Derivative attribute), 132
- variables (diofant.core.function.Lambda attribute), 128
- variables (diofant.core.function.Subs attribute), 137
- variables (diofant.utilities.codegen.Routine attribute), 1129
- variance() (in module diofant.stats), 989
- variations() (in module diofant.utilities.iterables), 1162
- vec() (diofant.matrices.matrices.MatrixBase method), 620
- vech() (diofant.matrices.matrices.MatrixBase method), 620
- Vector (class in diofant.vector.vector), 1215
- vectorize (class in diofant.core.multidimensional), 127
- vectors_in_basis() (in module diofant.diffgeom), 1189
- verify_numerically() (in module diofant.utilities.randtest), 1166
- vertices (diofant.combinatorics.polyhedron.Polyhedron attribute), 219
- vertices (diofant.geometry.polygon.Polygon attribute), 504
- vertices (diofant.geometry.polygon.RegularPolygon attribute), 512
- vertices (diofant.geometry.polygon.Triangle attribute), 519
- VF() (in module diofant.printing.pretty.pretty_symbology), 882
- viete() (in module diofant.polys.polyfuncs), 731
- vobj() (in module diofant.printing.pretty.pretty_symbology),

882
 VonMises() (in module diofant.stats), 984
 vradius (diofant.geometry.ellipse.Circle attribute), 498
 vradius (diofant.geometry.ellipse.Ellipse attribute), 495
 vring() (in module diofant.polys.rings), 826
 vstack() (diofant.matrices.matrices.MatrixBase class method), 621

W

WedgeProduct (class in diofant.diffgeom), 1185
 Weibull() (in module diofant.stats), 984
 where() (in module diofant.stats), 989
 width() (diofant.printing.pretty.stringpict.stringPict method), 884
 WignerSemicircle() (in module diofant.stats), 985
 Wild (class in diofant.core.symbol), 81
 WildFunction (class in diofant.core.function), 128
 write() (diofant.utilities.codegen.CodeGen method), 1130
 wronskian() (in module diofant.matrices.dense), 625

X

x (diofant.geometry.point.Point2D attribute), 445
 x (diofant.geometry.point.Point3D attribute), 448
 xdirection (diofant.geometry.line.Ray attribute), 462
 xdirection (diofant.geometry.line3d.Ray3D attribute), 477
 xobj() (in module diofant.printing.pretty.pretty_symbology), 882
 XOR (class in diofant.logic.boolalg), 571
 xreplace() (diofant.core.basic.Atom method), 42
 xreplace() (diofant.core.basic.Basic method), 56
 xreplace() (diofant.matrices.immutable.ImmutableSparseMatrix method), 652
 xreplace() (diofant.matrices.matrices.MatrixBase method), 621
 xring() (in module diofant.polys.rings), 826
 xsym() (in module diofant.printing.pretty.pretty_symbology), 882

Y

y (diofant.geometry.point.Point2D attribute), 445
 y (diofant.geometry.point.Point3D attribute), 449
 ydirection (diofant.geometry.line.Ray attribute), 463
 ydirection (diofant.geometry.line3d.Ray3D attribute), 478
 yn (class in diofant.functions.special.bessel), 392
 Ynm (class in diofant.functions.special.spherical_harmonics), 422
 Ynm_c() (in module diofant.functions.special.spherical_harmonics), 424

Z

z (diofant.geometry.point.Point3D attribute), 449
 zdirection (diofant.geometry.line3d.Ray3D attribute), 478
 Zero (class in diofant.core.numbers), 93
 zero (diofant.polys.polytools.Poly attribute), 727
 ZeroMatrix (class in diofant.matrices.expressions), 658
 zeros() (diofant.matrices.dense.DenseMatrix class method), 633
 zeros() (diofant.matrices.sparse.SparseMatrixBase class method), 651
 zeros() (in module diofant.matrices.dense), 622
 zeta (class in diofant.functions.special.zeta_functions), 400
 zip_row_op() (diofant.matrices.dense.MutableDenseMatrix method), 636
 zip_row_op() (diofant.matrices.sparse.MutableSparseMatrix method), 642
 Znm (class in diofant.functions.special.spherical_harmonics), 424