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.

```
>>> 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, 3 - t_0)}
```

The most interesting case is when $\Delta > 0$ and it is not a perfect square. In this case, the equation has either no solutions or an infinite number of solutions. Consider the below cases where $\Delta = 8$.

```
>>> diophantine(x**2 - 4*x*y + 2*y**2 - 3*x + 7*y - 5)
set()
>>> from sympy import sqrt
>>> n = symbols("n", integer=True)
>>> s = diophantine(x**2 -  2*y**2 - 2*x - 4*y, n)
>>> x_1, y_1 = s.pop()
>>> x_2, y_2 = s.pop()
>>> x_n = -(-2*sqrt(2) + 3)**n/2 + sqrt(2)*(-2*sqrt(2) + 3)**n/2 -␣
↪sqrt(2)*(2*sqrt(2) + 3)**n/2 - (2*sqrt(2) + 3)**n/2 + 1
>>> x_1 == x_n or x_2 == x_n
True
>>> y_n = -sqrt(2)*(-2*sqrt(2) + 3)**n/4 + (-2*sqrt(2) + 3)**n/2 +␣
↪sqrt(2)*(2*sqrt(2) + 3)**n/4 + (2*sqrt(2) + 3)**n/2 - 1
>>> y_1 == y_n or y_2 == y_n
True
```

Here n is an integer. Although x_n and y_n may not look like integers, substituting in specific values for n (and simplifying) shows that they are. For example consider the following example where we set n equal to 9.

```
>>> from sympy import simplify
>>> simplify(x_n.subs({n: 9}))
-9369318
```

Any binary quadratic of the form $ax^2 + bxy + cy^2 + dx + ey + f = 0$ can be transformed to an equivalent form $X^2 - DY^2 = N$.

[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: https://web.archive.org/web/20160323033111/http://www.jpr2718.org/ax2p.pdf

```
>>> from sympy.solvers.diophantine.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 727)

```
>>> A, B = transformation_to_DN(x**2 - 3*x*y + y**2 - 7*x + 5*y - 3)
```

Here A is a 2 X 2 matrix and B is a 2 X 1 matrix such that the transformation

$$\begin{bmatrix} X \\ Y \end{bmatrix} = A \begin{bmatrix} x \\ y \end{bmatrix} + B$$

gives the equation $X^2 - 5Y^2 = 920$. Values of A and B are as belows.

```
>>> A
Matrix([
[1/10, 3/10],
[   0,  1/5]])
>>> B
Matrix([
[  1/5],
[-11/5]])
```

We can solve an equation of the form $X^2 - DY^2 = N$ by passing D and N to *diop_DN()* (page 724)

```
>>> diop_DN(5, 920)
[]
```

Unfortunately, our equation has no solution.

Now let's turn to homogeneous ternary quadratic equations. These equations are of the form $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$. These type of equations either have infinitely many solutions or no solutions (except the obvious solution (0, 0, 0))

```
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y + 6*y*z + 7*z*x)
{(0, 0, 0)}
>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
{(-16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q +␣
↪68*q**2)}
```

If you are only interested in a base solution rather than the parameterized general solution (to be more precise, one of the general solutions), you can use *diop_ternary_quadratic()* (page 729).

```
>>> from sympy.solvers.diophantine.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 729) first converts the given equation to an equivalent equation of the form $w^2 = AX^2 + BY^2$ and then it uses *descent()* (page 730) to solve the latter equation. You can refer to the docs of *transformation_to_normal()* (page 728) to find more

on this. The equation $w^2 = AX^2 + BY^2$ can be solved more easily by using the Aforementioned *descent()* (page 730).

```
>>> from sympy.solvers.diophantine.diophantine import descent
>>> descent(3, 1) # solves the equation w**2 = 3*Y**2 + Z**2
(1, 0, 1)
```

Here the solution tuple is in the order (w, Y, Z)

The extended Pythagorean equation, $a_1x_1^2 + a_2x_2^2 + \ldots + a_nx_n^2 = a_{n+1}x_{n+1}^2$ and the general sum of squares equation, $x_1^2 + x_2^2 + \ldots + x_n^2 = k$ can also be solved using the Diophantine module.

```
>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
{(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 731) can also be called directly to solve the same equation. Either you can call *diop_general_pythagorean()* (page 731) or use the high level API. For the general sum of squares, this is also true, but one advantage of calling *diop_general_sum_of_squares()* (page 731) is that you can control how many solutions are returned.

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_of_
↪squares
>>> eq = a**2 + b**2 + c**2 + d**2 - 18
>>> diophantine(eq)
{(0, 0, 3, 3), (0, 1, 1, 4), (1, 2, 2, 3)}
>>> diop_general_sum_of_squares(eq, 2)
{(0, 0, 3, 3), (1, 2, 2, 3)}
```

The *sum_of_squares()* (page 737) routine will providean iterator that returns solutions and one may control whether the solutions contain zeros or not (and the solutions not containing zeros are returned first):

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_squares
>>> sos = sum_of_squares(18, 4, zeros=True)
>>> next(sos)
(1, 2, 2, 3)
>>> next(sos)
(0, 0, 3, 3)
```

Simple Eqyptian fractions can be found with the Diophantine module, too. For example, here are the ways that one might represent 1/2 as a sum of two unit fractions:

```
>>> from sympy import Eq, S
>>> diophantine(Eq(1/x + 1/y, S(1)/2))
{(-2, 1), (1, -2), (3, 6), (4, 4), (6, 3)}
```

To get a more thorough understanding of the Diophantine module, please refer to the following blog.

http://thilinaatsympy.wordpress.com/

References

User Functions

This functions is imported into the global namespace with `from sympy import *`:

diophantine

sympy.solvers.diophantine.diophantine.**diophantine**(*eq*, *param=t*, *syms=None*, *permute=False*)

Simplify the solution procedure of diophantine equation `eq` by converting it into a product of terms which should equal zero.

Explanation

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()`. (Although it is possible to call `diop_solve()` directly, one must be careful to pass an equation in the correct form and to interpret the output correctly; `diophantine()` is the public-facing function to use in general.)

Output of `diophantine()` is a set of tuples. The elements of the tuple are the solutions for each variable in the equation and are arranged according to the alphabetic ordering of the variables. e.g. For an equation with two variables, a and b, the first element of the tuple is the solution for a and the second for b.

Usage

`diophantine(eq, t, syms)`: Solve the diophantine equation `eq`. `t` is the optional parameter to be used by `diop_solve()`. `syms` is an optional list of symbols which determines the order of the elements in the returned tuple.

By default, only the base solution is returned. If `permute` is set to True then permutations of the base solution and/or permutations of the signs of the values will be returned when applicable.

Examples

```
>>> from sympy.abc import x, y, z
>>> diophantine(x**2 - y**2)
{(t_0, -t_0), (t_0, t_0)}
```

```
>>> diophantine(x*(2*x + 3*y - z))
{(0, n1, n2), (t_0, t_1, 2*t_0 + 3*t_1)}
>>> diophantine(x**2 + 3*x*y + 4*x)
{(0, n1), (3*t_0 - 4, -t_0)}
```

Details

`eq` should be an expression which is assumed to be zero. `t` is the parameter to be used in the solution.

See also:

diop_solve (page 721), *sympy.utilities.iterables.permute_signs* (page 2090), *sympy.utilities.iterables.signed_permutations* (page 2094)

And this function is imported with `from sympy.solvers.diophantine import *`:

classify_diop

`sympy.solvers.diophantine.diophantine.`**`classify_diop`**(*eq*, *_dict=True*)

Internal Functions

These functions are intended for internal use in the Diophantine module.

diop_solve

`sympy.solvers.diophantine.diophantine.`**`diop_solve`**(*eq*, *param=t*)

Solves the diophantine equation `eq`.

Explanation

Unlike `diophantine()`, factoring of `eq` is not attempted. Uses `classify_diop()` to determine the type of the equation and calls the appropriate solver function.

Use of `diophantine()` is recommended over other helper functions. `diop_solve()` can return either a set or a tuple depending on the nature of the equation.

Usage

`diop_solve(eq, t)`: Solve diophantine equation, `eq` using `t` as a parameter if needed.

Details

`eq` should be an expression which is assumed to be zero. `t` is a parameter to be used in the solution.

Examples

```
>>> from sympy.solvers.diophantine import diop_solve
>>> from sympy.abc import x, y, z, w
>>> diop_solve(2*x + 3*y - 5)
(3*t_0 - 5, 5 - 2*t_0)
>>> diop_solve(4*x + 3*y - 4*z + 5)
(t_0, 8*t_0 + 4*t_1 + 5, 7*t_0 + 3*t_1 + 5)
>>> diop_solve(x + 3*y - 4*z + w - 6)
(t_0, t_0 + t_1, 6*t_0 + 5*t_1 + 4*t_2 - 6, 5*t_0 + 4*t_1 + 3*t_2 - 6)
>>> diop_solve(x**2 + y**2 - 5)
{(-2, -1), (-2, 1), (-1, -2), (-1, 2), (1, -2), (1, 2), (2, -1), (2, 1)}
```

See also:

diophantine (page 720)

diop_linear

sympy.solvers.diophantine.diophantine.**diop_linear**(*eq*, *param=t*)

Solves linear diophantine equations.

A linear diophantine equation is an equation of the form $a_1x_1 + a_2x_2 + .. + a_nx_n = 0$ where $a_1, a_2, ..a_n$ are integer constants and $x_1, x_2, ..x_n$ are integer variables.

Usage

`diop_linear(eq)`: Returns a tuple containing solutions to the diophantine equation `eq`. Values in the tuple is arranged in the same order as the sorted variables.

Details

`eq` is a linear diophantine equation which is assumed to be zero. `param` is the parameter to be used in the solution.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_linear
>>> from sympy.abc import x, y, z
>>> diop_linear(2*x - 3*y - 5) # solves equation 2*x - 3*y - 5 == 0
(3*t_0 - 5, 2*t_0 - 5)
```

Here x = -3*t_0 - 5 and y = -2*t_0 - 5

```
>>> diop_linear(2*x - 3*y - 4*z -3)
(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)
```

See also:

diop_quadratic (page 723), *diop_ternary_quadratic* (page 729), *diop_general_pythagorean* (page 731), *diop_general_sum_of_squares* (page 731)

base_solution_linear

sympy.solvers.diophantine.diophantine.**base_solution_linear**(*c*, *a*, *b*, *t=None*)

Return the base solution for the linear equation, $ax + by = c$.

Explanation

Used by `diop_linear()` to find the base solution of a linear Diophantine equation. If `t` is given then the parametrized solution is returned.

Usage

`base_solution_linear(c, a, b, t)`: `a`, `b`, `c` are coefficients in $ax + by = c$ and `t` is the parameter to be used in the solution.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import base_solution_
↪linear
>>> from sympy.abc import t
>>> base_solution_linear(5, 2, 3) # equation 2*x + 3*y = 5
(-5, 5)
>>> base_solution_linear(0, 5, 7) # equation 5*x + 7*y = 0
(0, 0)
>>> base_solution_linear(5, 2, 3, t) # equation 2*x + 3*y = 5
(3*t - 5, 5 - 2*t)
>>> base_solution_linear(0, 5, 7, t) # equation 5*x + 7*y = 0
(7*t, -5*t)
```

diop_quadratic

sympy.solvers.diophantine.diophantine.**diop_quadratic**(*eq*, *param=t*)

Solves quadratic diophantine equations.

i.e. equations of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$. Returns a set containing the tuples (x, y) which contains the solutions. If there are no solutions then $(None, None)$ is returned.

Usage

`diop_quadratic(eq, param)`: `eq` is a quadratic binary diophantine equation. `param` is used to indicate the parameter to be used in the solution.

Details

`eq` should be an expression which is assumed to be zero. `param` is a parameter to be used in the solution.

Examples

```
>>> from sympy.abc import x, y, t
>>> from sympy.solvers.diophantine.diophantine import diop_quadratic
>>> diop_quadratic(x**2 + y**2 + 2*x + 2*y + 2, t)
{(-1, -1)}
```

See also:

diop_linear (page 722), *diop_ternary_quadratic* (page 729), *diop_general_sum_of_squares* (page 731), *diop_general_pythagorean* (page 731)

References

[R765], [R766]

diop_DN

`sympy.solvers.diophantine.diophantine.`**diop_DN**(*D*, *N*, *t=t*)

Solves the equation $x^2 - Dy^2 = N$.

Explanation

Mainly concerned with the case $D > 0, D$ is not a perfect square, which is the same as the generalized Pell equation. The LMM algorithm [R767] is used to solve this equation.

Returns one solution tuple, (x, y) for each class of the solutions. Other solutions of the class can be constructed according to the values of `D` and `N`.

Usage

`diop_DN(D, N, t)`: D and N are integers as in $x^2 - Dy^2 = N$ and `t` is the parameter to be used in the solutions.

Details

`D` and `N` correspond to D and N in the equation. `t` is the parameter to be used in the solutions.

Examples

```
>>> from sympy.solvers.diophantine.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)]
```

See also:

find_DN (page 728), *diop_bf_DN* (page 726)

References

[R767]

cornacchia

sympy.solvers.diophantine.diophantine.**cornacchia**(*a*, *b*, *m*)

Solves $ax^2 + by^2 = m$ where $\gcd(a, b) = 1 = gcd(a, m)$ and $a, b > 0$.

Explanation

Uses the algorithm due to Cornacchia. The method only finds primitive solutions, i.e. ones with $\gcd(x, y) = 1$. So this method cannot be used to find the solutions of $x^2 + y^2 = 20$ since the only solution to former is $(x, y) = (4, 2)$ and it is not primitive. When $a = b$, only the solutions with $x \leq y$ are found. For more details, see the References.

Examples

```
>>> from sympy.solvers.diophantine.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)}
```

See also:

sympy.utilities.iterables.signed_permutations (page 2094)

References

[R768], [R769]

diop_bf_DN

sympy.solvers.diophantine.diophantine.**diop_bf_DN**(*D*, *N*, *t=t*)

Uses brute force to solve the equation, $x^2 - Dy^2 = N$.

Explanation

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 [R770]. 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.

Usage

`diop_bf_DN(D, N, t)`: `D` and `N` are coefficients in $x^2 - Dy^2 = N$ and `t` is the parameter to be used in the solutions.

Details

`D` and `N` correspond to D and N in the equation. `t` is the parameter to be used in the solutions.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_bf_DN
>>> diop_bf_DN(13, -4)
[(3, 1), (-3, 1), (36, 10)]
>>> diop_bf_DN(986, 1)
[(49299, 1570)]
```

See also:

diop_DN (page 724)

References

[R770]

transformation_to_DN

sympy.solvers.diophantine.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.

Explanation

This is used to solve the general quadratic equation by transforming it to the latter form. Refer to [R771] 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,

Transpose([x y]) = A * Transpose([X Y]) + B

Usage

`transformation_to_DN(eq)`: where `eq` is the quadratic to be transformed.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import transformation_to_
↪DN
>>> 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 Transpose((x y)) = A * Transpose((X Y)) + B. Substituting these values for x and y and a bit of simplifying work will give an equation of the form $x^2 - Dy^2 = N$.

```
>>> from sympy.abc import X, Y
>>> from sympy import Matrix, simplify
>>> u = (A*Matrix([X, Y]) + B)[0] # Transformation for x
>>> u
X/26 + 3*Y/26 - 6/13
>>> v = (A*Matrix([X, Y]) + B)[1] # Transformation for y
>>> v
Y/13 - 4/13
```

Next we will substitute these formulas for x and y and do `simplify()`.

```
>>> eq = simplify((x**2 - 3*x*y - y**2 - 2*y + 1).subs(zip((x, y), (u,␣
↪v))))
>>> eq
X**2/676 - Y**2/52 + 17/13
```

By multiplying the denominator appropriately, we can get a Pell equation in the standard form.

```
>>> eq * 676
X**2 - 13*Y**2 + 884
```

If only the final equation is needed, `find_DN()` can be used.

See also:

find_DN (page 728)

References

[R771]

transformation_to_normal

`sympy.solvers.diophantine.diophantine.`**`transformation_to_normal`**`(`*eq*`)`

Returns the transformation Matrix that converts a general ternary quadratic equation `eq` ($ax^2+by^2+cz^2+dxy+eyz+fxz$) to a form without cross terms: $ax^2+by^2+cz^2=0$. This is not used in solving ternary quadratics; it is only implemented for the sake of completeness.

find_DN

`sympy.solvers.diophantine.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()`.

Usage

`find_DN(eq)`: where `eq` is the quadratic to be transformed.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.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()`.

See also:

transformation_to_DN (page 727)

References

[R772]

diop_ternary_quadratic

sympy.solvers.diophantine.diophantine.**diop_ternary_quadratic**(*eq*, *parameterize=False*)

Solves the general quadratic ternary form, $ax^2 + by^2 + cz^2 + fxy + gyz + hxz = 0$.

Returns a tuple (x, y, z) which is a base solution for the above equation. If there are no solutions, $(None, None, None)$ is returned.

Usage

`diop_ternary_quadratic(eq)`: Return a tuple containing a basic solution to `eq`.

Details

`eq` should be an homogeneous expression of degree two in three variables and it is assumed to be zero.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.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

sympy.solvers.diophantine.diophantine.**square_factor**(a)

Returns an integer c s.t. $a = c^2k,\ c,k \in Z$. Here k is square free. a can be given as an integer or a dictionary of factors.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import square_factor
>>> square_factor(24)
2
>>> square_factor(-36*3)
6
>>> square_factor(1)
1
>>> square_factor({3: 2, 2: 1, -1: 1})  # -18
3
```

See also:

sympy.ntheory.factor_.core (page 1504)

descent

sympy.solvers.diophantine.diophantine.**descent**(A, B)

Returns a non-trivial solution, (x, y, z), to $x^2 = Ay^2 + Bz^2$ using Lagrange's descent method with lattice-reduction. A and B are assumed to be valid for such a solution to exist.

This is faster than the normal Lagrange's descent algorithm because the Gaussian reduction is used.

Examples

```
>>> from sympy.solvers.diophantine.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)
```

References

[R773]

diop_general_pythagorean

sympy.solvers.diophantine.diophantine.**diop_general_pythagorean**(*eq*, *param=m*)

Solves the general pythagorean equation, $a_1^2x_1^2 + a_2^2x_2^2 + ... + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$.

Returns a tuple which contains a parametrized solution to the equation, sorted in the same order as the input variables.

Usage

`diop_general_pythagorean(eq, param)`: where `eq` is a general pythagorean equation which is assumed to be zero and `param` is the base parameter used to construct other parameters by subscripting.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_
↪pythagorean
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_pythagorean(a**2 + b**2 + c**2 - d**2)
(m1**2 + m2**2 - m3**2, 2*m1*m3, 2*m2*m3, m1**2 + m2**2 + m3**2)
>>> diop_general_pythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2)
(10*m1**2  + 10*m2**2  + 10*m3**2 - 10*m4**2, 15*m1**2  + 15*m2**2  +␣
↪15*m3**2  + 15*m4**2, 15*m1*m4, 12*m2*m4, 60*m3*m4)
```

diop_general_sum_of_squares

sympy.solvers.diophantine.diophantine.**diop_general_sum_of_squares**(*eq*, *limit=1*)

Solves the equation $x_1^2 + x_2^2 + ... + x_n^2 - k = 0$.

Returns at most `limit` number of solutions.

Usage

`general_sum_of_squares(eq, limit)` : Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^2 + x_2^2 + ... + x_n^2 - k = 0$.

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in Z$ then there will be no solutions. Refer to [R774] for more details.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_
↪of_squares
>>> from sympy.abc import a, b, c, d, e
>>> diop_general_sum_of_squares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345)
{(15, 22, 22, 24, 24)}
```

Reference

diop_general_sum_of_even_powers

sympy.solvers.diophantine.diophantine.**diop_general_sum_of_even_powers**(*eq*, *limit=1*)

Solves the equation $x_1^e + x_2^e + ... + x_n^e - k = 0$ where e is an even, integer power.

Returns at most `limit` number of solutions.

Usage

`general_sum_of_even_powers(eq, limit)` : Here `eq` is an expression which is assumed to be zero. Also, `eq` should be in the form, $x_1^e + x_2^e + ... + x_n^e - k = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import diop_general_sum_
↪of_even_powers
>>> from sympy.abc import a, b
>>> diop_general_sum_of_even_powers(a**4 + b**4 - (2**4 + 3**4))
{(2, 3)}
```

See also:

power_representation (page 733)

power_representation

sympy.solvers.diophantine.diophantine.**power_representation**(*n*, *p*, *k*, *zeros=False*)

Returns a generator for finding k-tuples of integers, $(n_1, n_2, ...n_k)$, such that $n = n_1^p + n_2^p + ...n_k^p$.

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number `n` as a sum of `k` `p`th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import power_
↪representation
```

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag $zeros$ is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the $permute_sign$ function can be used to get all signed values:

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-
↪12, -1)]
```

partition

sympy.solvers.diophantine.diophantine.**partition**(*n*, *k=None*, *zeros=False*)

Returns a generator that can be used to generate partitions of an integer n.

Explanation

A partition of n is a set of positive integers which add up to n. For example, partitions of 3 are 3, 1 + 2, 1 + 1 + 1. A partition is returned as a tuple. If k equals None, then all possible partitions are returned irrespective of their size, otherwise only the partitions of size k are returned. If the `zero` parameter is set to True then a suitable number of zeros are added at the end of every partition of size less than k.

`zero` parameter is considered only if k is not None. When the partitions are over, the last $next()$ call throws the `StopIteration` exception, so this function should always be used inside a try - except block.

Details

`partition(n, k)`: Here n is a positive integer and k is the size of the partition which is also positive integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import partition
>>> f = partition(5)
>>> next(f)
(1, 1, 1, 1, 1)
>>> next(f)
(1, 1, 1, 2)
>>> g = partition(5, 3)
>>> next(g)
(1, 1, 3)
>>> next(g)
(1, 2, 2)
>>> g = partition(5, 3, zeros=True)
>>> next(g)
(0, 0, 5)
```

sum_of_three_squares

sympy.solvers.diophantine.diophantine.**sum_of_three_squares**(*n*)

Returns a 3-tuple (a, b, c) such that $a^2 + b^2 + c^2 = n$ and $a, b, c \geq 0$.

Returns None if $n = 4^a(8m + 7)$ for some $a, m \in \mathbb{Z}$. See [R775] for more details.

Usage

`sum_of_three_squares(n)`: Here `n` is a non-negative integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_three_
↪squares
>>> sum_of_three_squares(44542)
(18, 37, 207)
```

See also:

sum_of_squares (page 737)

References

[R775]

sum_of_four_squares

sympy.solvers.diophantine.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 $a, b, c, d \geq 0$.

Usage

`sum_of_four_squares(n)`: Here `n` is a non-negative integer.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_four_squares
>>> sum_of_four_squares(3456)
(8, 8, 32, 48)
>>> sum_of_four_squares(1294585930293)
(0, 1234, 2161, 1137796)
```

See also:

sum_of_squares (page 737)

References

[R776]

sum_of_powers

sympy.solvers.diophantine.diophantine.**sum_of_powers**(*n*, *p*, *k*, *zeros=False*)

Returns a generator for finding k-tuples of integers, $(n_1, n_2, ...n_k)$, such that $n = n_1^p + n_2^p + ...n_k^p$.

Usage

`power_representation(n, p, k, zeros)`: Represent non-negative number `n` as a sum of `k` `p`th powers. If `zeros` is true, then the solutions is allowed to contain zeros.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import power_
↪representation
```

Represent 1729 as a sum of two cubes:

```
>>> f = power_representation(1729, 3, 2)
>>> next(f)
(9, 10)
>>> next(f)
(1, 12)
```

If the flag $zeros$ is True, the solution may contain tuples with zeros; any such solutions will be generated after the solutions without zeros:

```
>>> list(power_representation(125, 2, 3, zeros=True))
[(5, 6, 8), (3, 4, 10), (0, 5, 10), (0, 2, 11)]
```

For even p the $permute_sign$ function can be used to get all signed values:

```
>>> from sympy.utilities.iterables import permute_signs
>>> list(permute_signs((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12)]
```

All possible signed permutations can also be obtained:

```
>>> from sympy.utilities.iterables import signed_permutations
>>> list(signed_permutations((1, 12)))
[(1, 12), (-1, 12), (1, -12), (-1, -12), (12, 1), (-12, 1), (12, -1), (-
↪12, -1)]
```

sum_of_squares

sympy.solvers.diophantine.diophantine.**sum_of_squares**(*n*, *k*, *zeros=False*)

Return a generator that yields the k-tuples of nonnegative values, the squares of which sum to n. If zeros is False (default) then the solution will not contain zeros. The nonnegative elements of a tuple are sorted.

- If k == 1 and n is square, (n,) is returned.
- If k == 2 then n can only be written as a sum of squares if every prime in the factorization of n that has the form 4*k + 3 has an even multiplicity. If n is prime then it can only be written as a sum of two squares if it is in the form 4*k + 1.
- if k == 3 then n can be written as a sum of squares if it does not have the form 4**m*(8*k + 7).
- all integers can be written as the sum of 4 squares.
- if k > 4 then n can be partitioned and each partition can be written as a sum of 4 squares; if n is not evenly divisible by 4 then n can be written as a sum of squares only if the an additional partition can be written as sum of squares. For example, if k = 6 then n is partitioned into two parts, the first being written as a sum of 4 squares and the second being written as a sum of 2 squares – which can only be done if the condition above for k = 2 can be met, so this will automatically reject certain partitions of n.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sum_of_squares
>>> list(sum_of_squares(25, 2))
[(3, 4)]
>>> list(sum_of_squares(25, 2, True))
[(3, 4), (0, 5)]
>>> list(sum_of_squares(25, 4))
[(1, 2, 2, 4)]
```

See also:

sympy.utilities.iterables.signed_permutations (page 2094)

merge_solution

sympy.solvers.diophantine.diophantine.**merge_solution**(*var*, *var_t*, *solution*)

This is used to construct the full solution from the solutions of sub equations.

Explanation

For example when solving the equation $(x - y)(x^2 + y^2 - z^2) = 0$, solutions for each of the equations $x - y = 0$ and $x^2 + y^2 - z^2$ are found independently. Solutions for $x - y = 0$ are $(x, y) = (t, t)$. But we should introduce a value for z when we output the solution for the original equation. This function converts (t, t) into (t, t, n_1) where n_1 is an integer parameter.

divisible

sympy.solvers.diophantine.diophantine.**divisible**(*a*, *b*)

Returns $True$ if a is divisible by b and $False$ otherwise.

PQa

sympy.solvers.diophantine.diophantine.**PQa**(*P_0*, *Q_0*, *D*)

Returns useful information needed to solve the Pell equation.

Explanation

There are six sequences of integers defined related to the continued fraction representation of
$fracP + \sqrt{D}Q$, 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 [R777] for more detailed information.

Usage

`PQa(P_0, Q_0, D)`: `P_0`, `Q_0` and `D` are integers corresponding to P_0, Q_0 and D in the continued fraction
$fracP_0 + \sqrt{D}Q_0$. Also it's assumed that $P_0^2 == Dmod(|Q_0|)$ and D is square free.

Examples

```
>>> from sympy.solvers.diophantine.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)
```

References

[R777]

equivalent

sympy.solvers.diophantine.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.

Explanation

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 [R778]. No test is performed to test whether (u, v) and (r, s) are actually solutions to the equation. User should take care of this.

Usage

`equivalent(u, v, r, s, D, N)`: (u, v) and (r, s) are two solutions of the equation $x^2 - Dy^2 = N$ and all parameters involved are integers.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import equivalent
>>> equivalent(18, 5, -18, -5, 13, -1)
True
>>> equivalent(3, 1, -18, 393, 109, -4)
False
```

References

[R778]

parametrize_ternary_quadratic

sympy.solvers.diophantine.diophantine.**parametrize_ternary_quadratic**(*eq*)

Returns the parametrized general solution for the ternary quadratic equation `eq` which has the form $ax^2 + by^2 + cz^2 + fxy + gyz + hxz = 0$.

Examples

```
>>> from sympy import Tuple, ordered
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import parametrize_
↪ternary_quadratic
```

The parametrized solution may be returned with three parameters:

```
>>> parametrize_ternary_quadratic(2*x**2 + y**2 - 2*z**2)
(p**2 - 2*q**2, -2*p**2 + 4*p*q - 4*p*r - 4*q**2, p**2 - 4*p*q + 2*q**2 -
↪ 4*q*r)
```

There might also be only two parameters:

```
>>> parametrize_ternary_quadratic(4*x**2 + 2*y**2 - 3*z**2)
(2*p**2 - 3*q**2, -4*p**2 + 12*p*q - 6*q**2, 4*p**2 - 8*p*q + 6*q**2)
```

Notes

Consider p and q in the previous 2-parameter solution and observe that more than one solution can be represented by a given pair of parameters. If p and q are not coprime, this is trivially true since the common factor will also be a common factor of the solution values. But it may also be true even when p and q are coprime:

```
>>> sol = Tuple(*_)
>>> p, q = ordered(sol.free_symbols)
>>> sol.subs([(p, 3), (q, 2)])
(6, 12, 12)
>>> sol.subs([(q, 1), (p, 1)])
(-1, 2, 2)
>>> sol.subs([(q, 0), (p, 1)])
(2, -4, 4)
>>> sol.subs([(q, 1), (p, 0)])
(-3, -6, 6)
```

Except for sign and a common factor, these are equivalent to the solution of (1, 2, 2).

References

[R779]

diop_ternary_quadratic_normal

sympy.solvers.diophantine.diophantine.**diop_ternary_quadratic_normal**(*eq*, *parameterize=False*)

Solves the quadratic ternary diophantine equation, $ax^2 + by^2 + cz^2 = 0$.

Explanation

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.

Usage

`diop_ternary_quadratic_normal(eq)`: where `eq` is an equation of the form $ax^2 + by^2 + cz^2 = 0$.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.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

sympy.solvers.diophantine.diophantine.**ldescent**(*A*, *B*)

Return a non-trivial solution to $w^2 = Ax^2 + By^2$ using Lagrange's method; return None if there is no such solution. .

Here, $A \neq 0$ and $B \neq 0$ and A and B are square free. Output a tuple (w_0, x_0, y_0) which is a solution to the above equation.

Examples

```
>>> from sympy.solvers.diophantine.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)
```

References

[R780], [R781]

gaussian_reduce

sympy.solvers.diophantine.diophantine.**gaussian_reduce**(*w*, *a*, *b*)

Returns a reduced solution (x, z) to the congruence $X^2 - aZ^2 \equiv 0 \ (mod\ b)$ so that $x^2 + |a|z^2$ is minimal.

Details

Here `w` is a solution of the congruence $x^2 \equiv a \ (mod\ b)$

References

[R782], [R783]

holzer

sympy.solvers.diophantine.diophantine.**holzer**(*x*, *y*, *z*, *a*, *b*, *c*)

Simplify the solution (x, y, z) of the equation $ax^2 + by^2 = cz^2$ with $a, b, c > 0$ and $z^2 \geq |\ ab\ |$ to a new reduced solution (x', y', z') such that $z'^2 \leq |\ ab\ |$.

The algorithm is an interpretation of Mordell's reduction as described on page 8 of Cremona and Rusin's paper [R784] and the work of Mordell in reference [R785].

References

[R784], [R785]

prime_as_sum_of_two_squares

sympy.solvers.diophantine.diophantine.**prime_as_sum_of_two_squares**(*p*)

Represent a prime p as a unique sum of two squares; this can only be done if the prime is congruent to 1 mod 4.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import prime_as_sum_of_
↪two_squares
>>> prime_as_sum_of_two_squares(7)  # can't be done
>>> prime_as_sum_of_two_squares(5)
(1, 2)
```

Reference

See also:

sum_of_squares (page 737)

sqf_normal

sympy.solvers.diophantine.diophantine.**sqf_normal**(*a*, *b*, *c*, *steps=False*)

Return a', b', c', the coefficients of the square-free normal form of $ax^2 + by^2 + cz^2 = 0$, where a', b', c' are pairwise prime. If *steps* is True then also return three tuples: sq, sqf, and (a', b', c') where sq contains the square factors of a, b and c after removing the $gcd(a, b, c)$; sqf contains the values of a, b and c after removing both the $gcd(a, b, c)$ and the square factors.

The solutions for $ax^2 + by^2 + cz^2 = 0$ can be recovered from the solutions of $a'x^2 + b'y^2 + c'z^2 = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import sqf_normal
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11)
(11, 1, 5)
>>> sqf_normal(2 * 3**2 * 5, 2 * 5 * 11, 2 * 7**2 * 11, True)
((3, 1, 7), (5, 55, 11), (11, 1, 5))
```

See also:

reconstruct (page 744)

References

[R787]

reconstruct

sympy.solvers.diophantine.diophantine.**reconstruct**(*A*, *B*, *z*)

Reconstruct the z value of an equivalent solution of $ax^2 + by^2 + cz^2$ from the z value of a solution of the square-free normal form of the equation, $a' * x^2 + b' * y^2 + c' * z^2$, where a', b' and c' are square free and $gcd(a', b', c') == 1$.

Internal Classes

These classes are intended for internal use in the Diophantine module.

DiophantineSolutionSet

class sympy.solvers.diophantine.diophantine.**DiophantineSolutionSet**(*symbols_seq*, *parameters*)

Container for a set of solutions to a particular diophantine equation.

The base representation is a set of tuples representing each of the solutions.

Parameters

symbols : list

List of free symbols in the original equation.

parameters: list

List of parameters to be used in the solution.

Examples

Adding solutions:

```
>>> from sympy.solvers.diophantine.diophantine import␣
↪DiophantineSolutionSet
>>> from sympy.abc import x, y, t, u
>>> s1 = DiophantineSolutionSet([x, y], [t, u])
>>> s1
set()
>>> s1.add((2, 3))
>>> s1.add((-1, u))
>>> s1
{(-1, u), (2, 3)}
>>> s2 = DiophantineSolutionSet([x, y], [t, u])
>>> s2.add((3, 4))
>>> s1.update(*s2)
>>> s1
{(-1, u), (2, 3), (3, 4)}
```

Conversion of solutions into dicts:

```
>>> list(s1.dict_iterator())
[{x: -1, y: u}, {x: 2, y: 3}, {x: 3, y: 4}]
```

Substituting values:

```
>>> s3 = DiophantineSolutionSet([x, y], [t, u])
>>> s3.add((t**2, t + u))
>>> s3
{(t**2, t + u)}
>>> s3.subs({t: 2, u: 3})
{(4, 5)}
>>> s3.subs(t, -1)
{(1, u - 1)}
>>> s3.subs(t, 3)
{(9, u + 3)}
```

Evaluation at specific values. Positional arguments are given in the same order as the parameters:

```
>>> s3(-2, 3)
{(4, 1)}
>>> s3(5)
{(25, u + 5)}
>>> s3(None, 2)
{(t**2, t + 2)}
```


DiophantineEquationType

class sympy.solvers.diophantine.diophantine.**DiophantineEquationType**(*equation*, *free_symbols=None*)

Internal representation of a particular diophantine equation type.

Parameters

equation :

The diophantine equation that is being solved.

free_symbols : list (optional)

The symbols being solved for.

Attributes

total_degree :	The maximum of the degrees of all terms in the equation
homogeneous :	Does the equation contain a term of degree 0
homogeneous_order :	Does the equation contain any coefficient that is in the symbols being solved for
dimension :	The number of symbols being solved for

matches()

Determine whether the given equation can be matched to the particular equation type.

Univariate

class sympy.solvers.diophantine.diophantine.**Univariate**(*equation*, *free_symbols=None*)

Representation of a univariate diophantine equation.

A univariate diophantine equation is an equation of the form $a_0 + a_1 x + a_2 x^2 + .. + a_n x^n = 0$ where $a_1, a_2, ..a_n$ are integer constants and x is an integer variable.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import Univariate
>>> from sympy.abc import x
>>> Univariate((x - 2)*(x - 3)**2).solve() # solves equation (x - 2)*(x -
↪ 3)**2 == 0
{(2,), (3,)}
```

Linear

class sympy.solvers.diophantine.diophantine.**Linear**(*equation*, *free_symbols=None*)

Representation of a linear diophantine equation.

A linear diophantine equation is an equation of the form $a_1 x_1 + a_2 x_2 + .. + a_n x_n = 0$ where $a_1, a_2, ..a_n$ are integer constants and $x_1, x_2, ..x_n$ are integer variables.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import Linear
>>> from sympy.abc import x, y, z
>>> l1 = Linear(2*x - 3*y - 5)
>>> l1.matches() # is this equation linear
True
>>> l1.solve() # 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

```
>>> Linear(2*x - 3*y - 4*z -3).solve()
{(t_0, 2*t_0 + 4*t_1 + 3, -t_0 - 3*t_1 - 3)}
```

BinaryQuadratic

class sympy.solvers.diophantine.diophantine.**BinaryQuadratic**(*equation*, *free_symbols=None*)

Representation of a binary quadratic diophantine equation.

A binary quadratic diophantine equation is an equation of the form $Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$, where A, B, C, D, E, F are integer constants and x and y are integer variables.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import BinaryQuadratic
>>> b1 = BinaryQuadratic(x**3 + y**2 + 1)
>>> b1.matches()
False
>>> b2 = BinaryQuadratic(x**2 + y**2 + 2*x + 2*y + 2)
>>> b2.matches()
True
>>> b2.solve()
{(-1, -1)}
```

References

[R788], [R789]

InhomogeneousTernaryQuadratic

class sympy.solvers.diophantine.diophantine.**InhomogeneousTernaryQuadratic**(*equation*, *free_symbols=Non*

Representation of an inhomogeneous ternary quadratic.

No solver is currently implemented for this equation type.

HomogeneousTernaryQuadraticNormal

class sympy.solvers.diophantine.diophantine.**HomogeneousTernaryQuadraticNormal**(*equation*, *free_symbols=*

Representation of a homogeneous ternary quadratic normal diophantine equation.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import␣
↪HomogeneousTernaryQuadraticNormal
>>> HomogeneousTernaryQuadraticNormal(4*x**2 - 5*y**2 + z**2).solve()
{(1, 2, 4)}
```

HomogeneousTernaryQuadratic

class sympy.solvers.diophantine.diophantine.**HomogeneousTernaryQuadratic**(*equation*, *free_symbols=None*)

Representation of a homogeneous ternary quadratic diophantine equation.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.solvers.diophantine.diophantine import␣
↪HomogeneousTernaryQuadratic
>>> HomogeneousTernaryQuadratic(x**2 + y**2 - 3*z**2 + x*y).solve()
{(-1, 2, 1)}
>>> HomogeneousTernaryQuadratic(3*x**2 + y**2 - 3*z**2 + 5*x*y + y*z).
↪solve()
{(3, 12, 13)}
```

InhomogeneousGeneralQuadratic

class sympy.solvers.diophantine.diophantine.**InhomogeneousGeneralQuadratic**(*equation*, *free_symbols=Non*

Representation of an inhomogeneous general quadratic.

No solver is currently implemented for this equation type.

HomogeneousGeneralQuadratic

class sympy.solvers.diophantine.diophantine.**HomogeneousGeneralQuadratic**(*equation*, *free_symbols=None*)

Representation of a homogeneous general quadratic.

No solver is currently implemented for this equation type.

GeneralSumOfSquares

class sympy.solvers.diophantine.diophantine.**GeneralSumOfSquares**(*equation*, *free_symbols=None*)

Representation of the diophantine equation

$x_1^2 + x_2^2 + ... + x_n^2 - k = 0$.

Details

When $n = 3$ if $k = 4^a(8m + 7)$ for some $a, m \in Z$ then there will be no solutions. Refer [R790] for more details.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import GeneralSumOfSquares
>>> from sympy.abc import a, b, c, d, e
>>> GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).solve()
{(15, 22, 22, 24, 24)}
```

By default only 1 solution is returned. Use the *limit* keyword for more:

```
>>> sorted(GeneralSumOfSquares(a**2 + b**2 + c**2 + d**2 + e**2 - 2345).
↪solve(limit=3))
[(15, 22, 22, 24, 24), (16, 19, 24, 24, 24), (16, 20, 22, 23, 26)]
```

References

[R790]

GeneralPythagorean

class sympy.solvers.diophantine.diophantine.**GeneralPythagorean**(*equation*, *free_symbols=None*)

Representation of the general pythagorean equation, $a_1^2x_1^2 + a_2^2x_2^2 + ... + a_n^2x_n^2 - a_{n+1}^2x_{n+1}^2 = 0$.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import GeneralPythagorean
>>> from sympy.abc import a, b, c, d, e, x, y, z, t
>>> GeneralPythagorean(a**2 + b**2 + c**2 - d**2).solve()
{(t_0**2 + t_1**2 - t_2**2, 2*t_0*t_2, 2*t_1*t_2, t_0**2 + t_1**2 + t_
↪2**2)}
>>> GeneralPythagorean(9*a**2 - 4*b**2 + 16*c**2 + 25*d**2 + e**2).
↪solve(parameters=[x, y, z, t])
{(-10*t**2 + 10*x**2 + 10*y**2 + 10*z**2, 15*t**2 + 15*x**2 + 15*y**2 +␣
↪15*z**2, 15*t*x, 12*t*y, 60*t*z)}
```

CubicThue

class sympy.solvers.diophantine.diophantine.**CubicThue**(*equation*, *free_symbols=None*)

Representation of a cubic Thue diophantine equation.

A cubic Thue diophantine equation is a polynomial of the form $f(x, y) = r$ of degree 3, where x and y are integers and r is a rational number.

No solver is currently implemented for this equation type.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.solvers.diophantine.diophantine import CubicThue
>>> c1 = CubicThue(x**3 + y**2 + 1)
>>> c1.matches()
True
```

GeneralSumOfEvenPowers

class sympy.solvers.diophantine.diophantine.**GeneralSumOfEvenPowers**(*equation*, *free_symbols=None*)

Representation of the diophantine equation

$x_1^e + x_2^e + ... + x_n^e - k = 0$

where e is an even, integer power.

Examples

```
>>> from sympy.solvers.diophantine.diophantine import␣
↪GeneralSumOfEvenPowers
>>> from sympy.abc import a, b
>>> GeneralSumOfEvenPowers(a**4 + b**4 - (2**4 + 3**4)).solve()
{(2, 3)}
```

Inequality Solvers

sympy.solvers.inequalities.**solve_rational_inequalities**(*eqs*)

Solve a system of rational inequalities with rational coefficients.

Examples

```
>>> from sympy.abc import x
>>> from sympy import solve_rational_inequalities, Poly
```

```
>>> solve_rational_inequalities([[
... ((Poly(-x + 1), Poly(1, x)), '>='),
... ((Poly(-x + 1), Poly(1, x)), '<=')]])
{1}
```

```
>>> solve_rational_inequalities([[
... ((Poly(x), Poly(1, x)), '!='),
... ((Poly(-x + 1), Poly(1, x)), '>=')]])
Union(Interval.open(-oo, 0), Interval.Lopen(0, 1))
```

See also:

solve_poly_inequality (page 751)

sympy.solvers.inequalities.**solve_poly_inequality**(*poly*, *rel*)

Solve a polynomial inequality with rational coefficients.

Examples

```
>>> from sympy import solve_poly_inequality, Poly
>>> from sympy.abc import x
```

```
>>> solve_poly_inequality(Poly(x, x, domain='ZZ'), '==')
[{0}]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '!=')
[Interval.open(-oo, -1), Interval.open(-1, 1), Interval.open(1, oo)]
```

```
>>> solve_poly_inequality(Poly(x**2 - 1, x, domain='ZZ'), '==')
[{-1}, {1}]
```

See also:

solve_poly_inequalities (page 751)

sympy.solvers.inequalities.**solve_poly_inequalities**(*polys*)

Solve polynomial inequalities with rational coefficients.

Examples

```
>>> from sympy import Poly
>>> from sympy.solvers.inequalities import solve_poly_inequalities
>>> from sympy.abc import x
>>> solve_poly_inequalities(((
... Poly(x**2 - 3), ">"), (
... Poly(-x**2 + 1), ">")))
Union(Interval.open(-oo, -sqrt(3)), Interval.open(-1, 1), Interval.
↪open(sqrt(3), oo))
```

sympy.solvers.inequalities.**reduce_rational_inequalities**(*exprs*, *gen*, *relational=True*)

Reduce a system of rational inequalities with rational coefficients.

Examples

```
>>> from sympy import Symbol
>>> from sympy.solvers.inequalities import reduce_rational_inequalities
```

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

```
>>> reduce_rational_inequalities([[x**2 <= 0]], x)
Eq(x, 0)
```

```
>>> reduce_rational_inequalities([[x + 2 > 0]], x)
-2 < x
>>> reduce_rational_inequalities([[(x + 2, ">")]], x)
-2 < x
>>> reduce_rational_inequalities([[x + 2]], x)
Eq(x, -2)
```

This function find the non-infinite solution set so if the unknown symbol is declared as extended real rather than real then the result may include finiteness conditions:

```
>>> y = Symbol('y', extended_real=True)
>>> reduce_rational_inequalities([[y + 2 > 0]], y)
(-2 < y) & (y < oo)
```

sympy.solvers.inequalities.**reduce_abs_inequality**(*expr*, *rel*, *gen*)

Reduce an inequality with nested absolute values.

Examples

```
>>> from sympy import reduce_abs_inequality, Abs, Symbol
>>> x = Symbol('x', real=True)
```

```
>>> reduce_abs_inequality(Abs(x - 5) - 3, '<', x)
(2 < x) & (x < 8)
```

```
>>> reduce_abs_inequality(Abs(x + 2)*3 - 13, '<', x)
(-19/3 < x) & (x < 7/3)
```

See also:

reduce_abs_inequalities (page 753)

sympy.solvers.inequalities.**reduce_abs_inequalities**(*exprs, gen*)

Reduce a system of inequalities with nested absolute values.

Examples

```
>>> from sympy import reduce_abs_inequalities, Abs, Symbol
>>> x = Symbol('x', extended_real=True)
```

```
>>> reduce_abs_inequalities([(Abs(3*x - 5) - 7, '<'),
... (Abs(x + 25) - 13, '>')], x)
(-2/3 < x) & (x < 4) & (((-oo < x) & (x < -38)) | ((-12 < x) & (x < oo)))
```

```
>>> reduce_abs_inequalities([(Abs(x - 4) + Abs(3*x - 5) - 7, '<')], x)
(1/2 < x) & (x < 4)
```

See also:

reduce_abs_inequality (page 752)

sympy.solvers.inequalities.**reduce_inequalities**(*inequalities, symbols=[]*)

Reduce a system of inequalities with rational coefficients.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy import reduce_inequalities
```

```
>>> reduce_inequalities(0 <= x + 3, [])
(-3 <= x) & (x < oo)
```

```
>>> reduce_inequalities(0 <= x + y*2 - 1, [x])
(x < oo) & (x >= 1 - 2*y)
```

sympy.solvers.inequalities.**solve_univariate_inequality**(*expr*, *gen*, *relational=True*, *domain=Reals*, *continuous=False*)

Solves a real univariate inequality.

Parameters

expr : Relational

The target inequality

gen : Symbol

The variable for which the inequality is solved

relational : bool

A Relational type output is expected or not

domain : Set

The domain over which the equation is solved

continuous: bool

True if expr is known to be continuous over the given domain (and so continuous_domain() does not need to be called on it)

Raises

NotImplementedError

The solution of the inequality cannot be determined due to limitation in *sympy.solvers.solveset.solvify()* (page 869).

Notes

Currently, we cannot solve all the inequalities due to limitations in *sympy.solvers.solveset.solvify()* (page 869). Also, the solution returned for trigonometric inequalities are restricted in its periodic interval.

Examples

```
>>> from sympy import solve_univariate_inequality, Symbol, sin, Interval,
 ↪ S
>>> x = Symbol('x')
```

```
>>> solve_univariate_inequality(x**2 >= 4, x)
((2 <= x) & (x < oo)) | ((-oo < x) & (x <= -2))
```

```
>>> solve_univariate_inequality(x**2 >= 4, x, relational=False)
Union(Interval(-oo, -2), Interval(2, oo))
```

```
>>> domain = Interval(0, S.Infinity)
>>> solve_univariate_inequality(x**2 >= 4, x, False, domain)
Interval(2, oo)
```

```
>>> solve_univariate_inequality(sin(x) > 0, x, relational=False)
Interval.open(0, pi)
```

See also:

sympy.solvers.solveset.solvify **(page 869)**
solver returning solveset solutions with solve's output API

ODE

User Functions

These are functions that are imported into the global namespace with `from sympy import *`. These functions (unlike *Hint Functions* (page 766), below) are intended for use by ordinary users of SymPy.

dsolve

sympy.solvers.ode.**dsolve**(*eq*, *func=None*, *hint='default'*, *simplify=True*, *ics=None*, *xi=None*, *eta=None*, *x0=0*, *n=6*, ***kwargs*)

Solves any (supported) kind of ordinary differential equation and system of ordinary differential equations.

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 755) docstring for supported methods). This can either be an *Equality* (page 1023), 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 could not 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 760). See Hints below for more options that you can use for hint.

`simplify` enables simplification by *odesimp()* (page 767). 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 764) with the help of various heuristics.

`ics` is the set of initial/boundary conditions for the differential equation.
It should be given in the form of `{f(x0): x1, f(x).diff(x).subs(x, x2): x3}` and so on. For power series solutions, if no initial conditions are specified `f(0)` is assumed to be `C0` and the power series solution is calculated about 0.

`x0` is the point about which the power series solution of a differential
equation is to be evaluated.

`n` gives the exponent of the dependent variable up to which the power series
solution of a differential equation is to be evaluated.

Hints

Aside from the various solving methods, there are also some meta-hints that you can pass to *dsolve()* (page 755):

`default`:
This uses whatever hint is returned first by *classify_ode()* (page 760). This is the default argument to *dsolve()* (page 755).

`all`:
To make *dsolve()* (page 755) 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 852) in `deutils.py`.
- `best`: The simplest hint; what would be returned by `best` below.
- `best_hint`: The hint that would produce the solution given by `best`. If more than one hint produces the best solution, the first one in the tuple returned by *classify_ode()* (page 760) 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 760).

`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 755) to hang because of a difficult or impossible integral. This meta-hint will also be much faster than `all`, because *integrate()* (page 964) is an expensive routine.