exclude=[] (default)
Do not 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, do not do any testing of solutions. This can be useful if you want 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, ect.

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; `particular` must be True)
Selects a fast heuristic to find a solution with many zeros whereas a value of False uses the very slow method guaranteed to find the largest number of zeros possible.

cubics=True (default)
Return explicit solutions when cubic expressions are encountered. When False, quartics and quintics are disabled, too.

quartics=True (default)
Return explicit solutions when quartic expressions are encountered. When False, quintics are disabled, too.

quintics=True (default)
Return explicit solutions (if possible) when quintic expressions are encountered.

Explanation

Currently supported:

- polynomial
- transcendental
- piecewise combinations of the above
- systems of linear and polynomial equations
- systems containing relational expressions

Examples

The output varies according to the input and can be seen by example:

```
>>> from sympy import solve, Poly, Eq, Function, exp
>>> from sympy.abc import x, y, z, a, b
>>> f = Function('f')
```

Boolean or univariate Relational:

```
>>> solve(x < 3)
(-oo < x) & (x < 3)
```

To always get a list of solution mappings, use flag dict=True:

```
>>> solve(x - 3, dict=True)
[{x: 3}]
>>> sol = solve([x - 3, y - 1], dict=True)
>>> sol
[{x: 3, y: 1}]
>>> sol[0][x]
3
>>> sol[0][y]
1
```

To get a list of *symbols* and set of solution(s) use flag set=True:

```
>>> solve([x**2 - 3, y - 1], set=True)
([x, y], {(-sqrt(3), 1), (sqrt(3), 1)})
```

Single expression and single symbol that is in the expression:

```
>>> solve(x - y, x)
[y]
>>> solve(x - 3, x)
[3]
```

(continues on next page)

(continued from previous page)

```
>>> solve(Eq(x, 3), x)
[3]
>>> solve(Poly(x - 3), x)
[3]
>>> solve(x**2 - y**2, x, set=True)
([x], {(-y,), (y,)})
>>> solve(x**4 - 1, x, set=True)
([x], {(-1,), (1,), (-I,), (I,)})
```

Single expression with no symbol that is in the expression:

```
>>> solve(3, x)
[]
>>> solve(x - 3, y)
[]
```

Single expression with no symbol given. In this case, all free *symbols* will be selected as potential *symbols* to solve for. If the equation is univariate then a list of solutions is returned; otherwise - as is the case when *symbols* are given as an iterable of length greater than 1 - a list of mappings will be returned:

```
>>> solve(x - 3)
[3]
>>> solve(x**2 - y**2)
[{x: -y}, {x: y}]
>>> solve(z**2*x**2 - z**2*y**2)
[{x: -y}, {x: y}, {z: 0}]
>>> solve(z**2*x - z**2*y**2)
[{x: y**2}, {z: 0}]
```

When an object other than a Symbol is given as a symbol, it is isolated algebraically and an implicit solution may be obtained. This is mostly provided as a convenience to save you from replacing the object with a Symbol and solving for that Symbol. It will only work if the specified object can be replaced with a Symbol using the subs method:

```
>>> solve(f(x) - x, f(x))
[x]
>>> solve(f(x).diff(x) - f(x) - x, f(x).diff(x))
[x + f(x)]
>>> solve(f(x).diff(x) - f(x) - x, f(x))
[-x + Derivative(f(x), x)]
>>> solve(x + exp(x)**2, exp(x), set=True)
([exp(x)], {(-sqrt(-x),), (sqrt(-x),)})
```

```
>>> from sympy import Indexed, IndexedBase, Tuple, sqrt
>>> A = IndexedBase('A')
>>> eqs = Tuple(A[1] + A[2] - 3, A[1] - A[2] + 1)
>>> solve(eqs, eqs.atoms(Indexed))
{A[1]: 1, A[2]: 2}
```

- To solve for a symbol implicitly, use implicit=True:

```
>>> solve(x + exp(x), x)
[-LambertW(1)]
>>> solve(x + exp(x), x, implicit=True)
[-exp(x)]
```

- It is possible to solve for anything that can be targeted with subs:

```
>>> solve(x + 2 + sqrt(3), x + 2)
[-sqrt(3)]
>>> solve((x + 2 + sqrt(3), x + 4 + y), y, x + 2)
{y: -2 + sqrt(3), x + 2: -sqrt(3)}
```

- Nothing heroic is done in this implicit solving so you may end up with a symbol still in the solution:

```
>>> eqs = (x*y + 3*y + sqrt(3), x + 4 + y)
>>> solve(eqs, y, x + 2)
{y: -sqrt(3)/(x + 3), x + 2: -2*x/(x + 3) - 6/(x + 3) + sqrt(3)/(x +␣
↪3)}
>>> solve(eqs, y*x, x)
{x: -y - 4, x*y: -3*y - sqrt(3)}
```

- If you attempt to solve for a number remember that the number you have obtained does not necessarily mean that the value is equivalent to the expression obtained:

```
>>> solve(sqrt(2) - 1, 1)
[sqrt(2)]
>>> solve(x - y + 1, 1)  # /!\ -1 is targeted, too
[x/(y - 1)]
>>> [_.subs(z, -1) for _ in solve((x - y + 1).subs(-1, z), 1)]
[-x + y]
```

- To solve for a function within a derivative, use `dsolve`.

Single expression and more than one symbol:

- When there is a linear solution:

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

- When undetermined coefficients are identified:
  - That are linear:

```
>>> solve((a + b)*x - b + 2, a, b)
{a: -2, b: 2}
```

  - That are nonlinear:

```
>>> solve((a + b)*x - b**2 + 2, a, b, set=True)
([a, b], {(-sqrt(2), sqrt(2)), (sqrt(2), -sqrt(2))})
```

- If there is no linear solution, then the first successful attempt for a nonlinear solution will be returned:

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

Iterable of one or more of the above:

- Involving relationals or bools:

```
>>> solve([x < 3, x - 2])
Eq(x, 2)
>>> solve([x > 3, x - 2])
False
```

- When the system is linear:
  - With a solution:

```
>>> solve([x - 3], x)
{x: 3}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - 15), x, y, z)
{x: -3, y: 1}
>>> solve((x + 5*y - 2, -3*x + 6*y - z), z, x, y)
{x: 2 - 5*y, z: 21*y - 6}
```

  - Without a solution:

```
>>> solve([x + 3, x - 3])
[]
```

- When the system is not linear:

```
>>> solve([x**2 + y -2, y**2 - 4], x, y, set=True)
([x, y], {(-2, -2), (0, 2), (2, -2)})
```

- If no *symbols* are given, all free *symbols* will be selected and a list of mappings returned:

```
>>> solve([x - 2, x**2 + y])
[{x: 2, y: -4}]
>>> solve([x - 2, x**2 + f(x)], {f(x), x})
[{x: 2, f(x): -4}]
```

- If any equation does not 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}
```

Additional Examples

`solve()` with check=True (default) will run through the symbol tags to elimate unwanted solutions. If no assumptions are included, all possible solutions will be returned:

```
>>> from sympy import Symbol, solve
>>> x = Symbol("x")
>>> solve(x**2 - 1)
[-1, 1]
```

By using the positive tag, only one solution will be returned:

```
>>> pos = Symbol("pos", positive=True)
>>> solve(pos**2 - 1)
[1]
```

Assumptions are not checked when `solve()` input involves relationals or bools.

When the solutions are checked, those that make any denominator zero are automatically excluded. If you do not want to exclude such solutions, then use the check=False option:

```
>>> from sympy import sin, limit
>>> solve(sin(x)/x)  # 0 is excluded
[pi]
```

If check=False, then a solution to the numerator being zero is found: x = 0. In this case, this is a spurious solution since $\sin(x)/x$ has the well known limit (without dicontinuity) of 1 at x = 0:

```
>>> solve(sin(x)/x, check=False)
[0, pi]
```

In the following case, however, the limit exists and is equal to the 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, you might not want explicit solutions (which can be quite long). If the expression is univariate, `CRootOf` instances will be returned instead:

```
>>> solve(x**3 - x + 1)
[-1/((-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)) -
```

(continues on next page)

(continued from previous page)

```
(-1/2 - sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3,
-(-1/2 + sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)/3 -
1/((-1/2 + sqrt(3)*I/2)*(3*sqrt(69)/2 + 27/2)**(1/3)),
-(3*sqrt(69)/2 + 27/2)**(1/3)/3 -
1/(3*sqrt(69)/2 + 27/2)**(1/3)]
>>> solve(x**3 - x + 1, cubics=False)
[CRootOf(x**3 - x + 1, 0),
 CRootOf(x**3 - x + 1, 1),
 CRootOf(x**3 - x + 1, 2)]
```

If the expression is multivariate, no solution might be returned:

```
>>> solve(x**3 - x + a, x, cubics=False)
[]
```

Sometimes solutions will be obtained even when a flag is False because the expression could be factored. In the following example, the equation can be factored as the product of a linear and a quadratic factor so explicit solutions (which did not require solving a cubic expression) are obtained:

```
>>> eq = x**3 + 3*x**2 + x - 1
>>> solve(eq, cubics=False)
[-1, -1 + sqrt(2), -sqrt(2) - 1]
```

Solving Equations Involving Radicals

Because of SymPy's use of the principle root, some solutions to radical equations will be missed unless check=False:

```
>>> from sympy import root
>>> eq = root(x**3 - 3*x**2, 3) + 1 - x
>>> solve(eq)
[]
>>> solve(eq, check=False)
[1/3]
```

In the above example, there is only a single solution to the equation. Other expressions will yield spurious roots which must be checked manually; roots which give a negative argument to odd-powered radicals will also need special checking:

```
>>> from sympy import real_root, S
>>> eq = root(x, 3) - root(x, 5) + S(1)/7
>>> solve(eq)  # this gives 2 solutions but misses a 3rd
[CRootOf(7*x**5 - 7*x**3 + 1, 1)**15,
CRootOf(7*x**5 - 7*x**3 + 1, 2)**15]
>>> sol = solve(eq, check=False)
>>> [abs(eq.subs(x,i).n(2)) for i in sol]
[0.48, 0.e-110, 0.e-110, 0.052, 0.052]
```

The first solution is negative so `real_root` must be used to see that it satisfies the expression:

```
>>> abs(real_root(eq.subs(x, sol[0])).n(2))
0.e-110
```

If the roots of the equation are not real then more care will be necessary to find the roots, especially for higher order equations. Consider the following expression:

```
>>> expr = root(x, 3) - root(x, 5)
```

We will construct a known value for this expression at x = 3 by selecting the 1-th root for each radical:

```
>>> expr1 = root(x, 3, 1) - root(x, 5, 1)
>>> v = expr1.subs(x, -3)
```

The `solve` function is unable to find any exact roots to this equation:

```
>>> eq = Eq(expr, v); eq1 = Eq(expr1, v)
>>> solve(eq, check=False), solve(eq1, check=False)
([], [])
```

The function `unrad`, however, can be used to get a form of the equation for which numerical roots can be found:

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy import nroots
>>> e, (p, cov) = unrad(eq)
>>> pvals = nroots(e)
>>> inversion = solve(cov, x)[0]
>>> xvals = [inversion.subs(p, i) for i in pvals]
```

Although `eq` or `eq1` could have been used to find `xvals`, the solution can only be verified with `expr1`:

```
>>> z = expr - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z.subs(x, xi).n()) < 1e-9]
[]
>>> z1 = expr1 - v
>>> [xi.n(chop=1e-9) for xi in xvals if abs(z1.subs(x, xi).n()) < 1e-9]
[-3.0]
```

See also:

rsolve **(page 853)**
: For solving recurrence relationships

dsolve **(page 755)**
: For solving differential equations

sympy.solvers.solvers.**solve_linear**(*lhs*, *rhs=0*, *symbols=[]*, *exclude=[]*)

Return a tuple derived from `f = lhs - rhs` that is one of the following: `(0, 1)`, `(0, 0)`, `(symbol, solution)`, `(n, d)`.

Explanation

`(0, 1)` meaning that `f` is independent of the symbols in *symbols* that are not in *exclude*.

`(0, 0)` meaning that there is no solution to the equation amongst the symbols given. If the first element of the tuple is not zero, then the function is guaranteed to be dependent on a symbol in *symbols*.

`(symbol, solution)` where symbol appears linearly in the numerator of `f`, is in *symbols* (if given), and is not in *exclude* (if given). No simplification is done to `f` other than a `mul=True` expansion, so the solution will correspond strictly to a unique solution.

`(n, d)` where `n` and `d` are the numerator and denominator of `f` when the numerator was not linear in any symbol of interest; `n` will never be a symbol unless a solution for that symbol was found (in which case the second element is the solution, not the denominator).

Examples

```
>>> from sympy import cancel, Pow
```

`f` is independent of the symbols in *symbols* that are not in *exclude*:

```
>>> from sympy import cos, sin, solve_linear
>>> from sympy.abc import x, y, z
>>> eq = y*cos(x)**2 + y*sin(x)**2 - y  # = y*(1 - 1) = 0
>>> solve_linear(eq)
(0, 1)
>>> eq = cos(x)**2 + sin(x)**2  # = 1
>>> solve_linear(eq)
(0, 1)
>>> solve_linear(x, exclude=[x])
(0, 1)
```

The variable `x` appears as a linear variable in each of the following:

```
>>> solve_linear(x + y**2)
(x, -y**2)
>>> solve_linear(1/x - y**2)
(x, y**(-2))
```

When not linear in `x` or `y` then the numerator and denominator are returned:

```
>>> solve_linear(x**2/y**2 - 3)
(x**2 - 3*y**2, y**2)
```

If the numerator of the expression is a symbol, then `(0, 0)` is returned if the solution for that symbol would have set any denominator to 0:

```
>>> eq = 1/(1/x - 2)
>>> eq.as_numer_denom()
(x, 1 - 2*x)
>>> solve_linear(eq)
(0, 0)
```

But automatic rewriting may cause a symbol in the denominator to appear in the numerator so a solution will be returned:

```
>>> (1/x)**-1
x
>>> solve_linear((1/x)**-1)
(x, 0)
```

Use an unevaluated expression to avoid this:

```
>>> solve_linear(Pow(1/x, -1, evaluate=False))
(0, 0)
```

If x is allowed to cancel in the following expression, then it appears to be linear in x, but this sort of cancellation is not done by `solve_linear` so the solution will always satisfy the original expression without causing a division by zero error.

```
>>> eq = x**2*(1/x - z**2/x)
>>> solve_linear(cancel(eq))
(x, 0)
>>> solve_linear(eq)
(x**2*(1 - z**2), x)
```

A list of symbols for which a solution is desired may be given:

```
>>> solve_linear(x + y + z, symbols=[y])
(y, -x - z)
```

A list of symbols to ignore may also be given:

```
>>> solve_linear(x + y + z, exclude=[x])
(y, -x - z)
```

(A solution for y is obtained because it is the first variable from the canonically sorted list of symbols that had a linear solution.)

sympy.solvers.solvers.**solve_linear_system**(*system*, **symbols*, ***flags*)

Solve system of N linear equations with M variables, which means both under- and overdetermined systems are supported.

Explanation

The possible number of solutions is zero, one, or infinite. Respectively, this procedure will return None or a dictionary with solutions. In the case of underdetermined systems, all arbitrary parameters are skipped. This may cause a situation in which an empty dictionary is returned. In that case, all symbols can be assigned arbitrary values.

Input to this function is a $N \times M + 1$ matrix, which means it has to be in augmented form. If you prefer to enter N equations and M unknowns then use `solve(Neqs, *Msymbols)` instead. Note: a local copy of the matrix is made by this routine so the matrix that is passed will not be modified.

The algorithm used here is fraction-free Gaussian elimination, which results, after elimination, in an upper-triangular matrix. Then solutions are found using back-substitution. This approach is more efficient and compact than the Gauss-Jordan method.

Examples

```
>>> from sympy import Matrix, solve_linear_system
>>> from sympy.abc import x, y
```

Solve the following system:

```
   x + 4 y ==  2
-2 x +   y == 14
```

```
>>> system = Matrix(( (1, 4, 2), (-2, 1, 14)))
>>> solve_linear_system(system, x, y)
{x: -6, y: 2}
```

A degenerate system returns an empty dictionary:

```
>>> system = Matrix(( (0,0,0), (0,0,0) ))
>>> solve_linear_system(system, x, y)
{}
```

sympy.solvers.solvers.**solve_linear_system_LU**(*matrix*, *syms*)

Solves the augmented matrix system using `LUsolve` and returns a dictionary in which solutions are keyed to the symbols of *syms* as ordered.

Explanation

The matrix must be invertible.

Examples

```
>>> from sympy import Matrix, solve_linear_system_LU
>>> from sympy.abc import x, y, z
```

```
>>> solve_linear_system_LU(Matrix([
... [1, 2, 0, 1],
... [3, 2, 2, 1],
... [2, 0, 0, 1]]), [x, y, z])
{x: 1/2, y: 1/4, z: -1/2}
```

See also:

LUsolve (page 1287)

sympy.solvers.solvers.**solve_undetermined_coeffs**(*equ*, *coeffs*, *sym*, ***flags*)

Solve equation of a type $p(x; a_1, \ldots, a_k) = q(x)$ where both p and q are univariate polynomials that depend on k parameters.

Explanation

The result of this function is a dictionary with symbolic values of those parameters with respect to coefficients in q.

This function accepts both equations class instances and ordinary SymPy expressions. Specification of parameters and variables is obligatory for efficiency and simplicity reasons.

Examples

```
>>> from sympy import Eq, solve_undetermined_coeffs
>>> from sympy.abc import a, b, c, x
```

```
>>> solve_undetermined_coeffs(Eq(2*a*x + a+b, x), [a, b], x)
{a: 1/2, b: -1/2}
```

```
>>> solve_undetermined_coeffs(Eq(a*c*x + a+b, x), [a, b], x)
{a: 1/c, b: -1/c}
```

sympy.solvers.solvers.**nsolve**(**args, dict=False, **kwargs*)

Solve a nonlinear equation system numerically: `nsolve(f, [args,] x0, modules=['mpmath'], **kwargs)`.

Explanation

`f` is a vector function of symbolic expressions representing the system. *args* are the variables. If there is only one variable, this argument can be omitted. `x0` is a starting vector close to a solution.

Use the modules keyword to specify which modules should be used to evaluate the function and the Jacobian matrix. Make sure to use a module that supports matrices. For more information on the syntax, please see the docstring of `lambdify`.

If the keyword arguments contain `dict=True` (default is False) `nsolve` will return a list (perhaps empty) of solution mappings. This might be especially useful if you want to use `nsolve` as a fallback to solve since using the dict argument for both methods produces return values of consistent type structure. Please note: to keep this consistent with `solve`, the solution will be returned in a list even though `nsolve` (currently at least) only finds one solution at a time.

Overdetermined systems are supported.

Examples

```
>>> from sympy import Symbol, nsolve
>>> import mpmath
>>> mpmath.mp.dps = 15
>>> x1 = Symbol('x1')
>>> x2 = Symbol('x2')
>>> f1 = 3 * x1**2 - 2 * x2**2 - 1
>>> f2 = x1**2 - 2 * x1 + x2**2 + 2 * x2 - 8
>>> print(nsolve((f1, f2), (x1, x2), (-1, 1)))
Matrix([[-1.19287309935246], [1.27844411169911]])
```

For one-dimensional functions the syntax is simplified:

```
>>> from sympy import sin, nsolve
>>> from sympy.abc import x
>>> nsolve(sin(x), x, 2)
3.14159265358979
>>> nsolve(sin(x), 2)
3.14159265358979
```

To solve with higher precision than the default, use the prec argument:

```
>>> from sympy import cos
>>> nsolve(cos(x) - x, 1)
0.739085133215161
>>> nsolve(cos(x) - x, 1, prec=50)
0.73908513321516064165531208767387340401341175890076
>>> cos(_)
0.73908513321516064165531208767387340401341175890076
```

To solve for complex roots of real functions, a nonreal initial point must be specified:

```
>>> from sympy import I
>>> nsolve(x**2 + 2, I)
1.4142135623731*I
```

`mpmath.findroot` is used and you can find their more extensive documentation, especially concerning keyword parameters and available solvers. Note, however, that functions which are very steep near the root, the verification of the solution may fail. In this case you should use the flag `verify=False` and independently verify the solution.

```
>>> from sympy import cos, cosh
>>> 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); ans
312.588469032184
>>> f.subs(x, ans).n(2)
2.1e+121
```

(continues on next page)

(continued from previous page)

```
>>> (f/f.diff(x)).subs(x, ans).n(2)
7.4e-15
```

One might safely skip the verification if bounds of the root are known and a bisection method is used:

```
>>> bounds = lambda i: (3.14*i, 3.14*(i + 1))
>>> nsolve(f, bounds(100), solver='bisect', verify=False)
315.730061685774
```

Alternatively, a function may be better behaved when the denominator is ignored. Since this is not always the case, however, the decision of what function to use is left to the discretion of the user.

```
>>> eq = x**2/(1 - x)/(1 - 2*x)**2 - 100
>>> nsolve(eq, 0.46)
Traceback (most recent call last):
...
ValueError: Could not find root within given tolerance. (10000 > 2.1684e-
↪19)
Try another starting point or tweak arguments.
>>> nsolve(eq.as_numer_denom()[0], 0.46)
0.46792545969349058
```

sympy.solvers.solvers.**checksol**(*f*, *symbol*, *sol=None*, ***flags*)

Checks whether sol is a solution of equation f == 0.

Explanation

Input can be either a single symbol and corresponding value or a dictionary of symbols and values. When given as a dictionary and flag `simplify=True`, the values in the dictionary will be simplified. *f* can be a single equation or an iterable of equations. A solution must satisfy all equations in *f* to be considered valid; if a solution does not satisfy any equation, False is returned; if one or more checks are inconclusive (and none are False) then None is returned.

Examples

```
>>> from sympy import checksol, symbols
>>> 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.

`sympy.solvers.solvers.`**`unrad`**(*eq*, **syms*, ***flags*)

Remove radicals with symbolic arguments and return (eq, cov), None, or raise an error.

Explanation

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 rewritten 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 four terms with sqrt() factors or there are less than four terms having sqrt() factors.
- There are only two terms with radicals.

Examples

```
>>> from sympy.solvers.solvers import unrad
>>> from sympy.abc import x
>>> from sympy import sqrt, Rational, root
```

```
>>> 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, _p**6 - x])
```

Ordinary Differential equations (ODEs)

See *ODE* (page 755).

Partial Differential Equations (PDEs)

See *PDE* (page 826).

Deutils (Utilities for solving ODE's and PDE's)

sympy.solvers.deutils.**ode_order**(*expr*, *func*)

Returns the order of a given differential equation with respect to func.

This function is implemented recursively.

Examples

```
>>> from sympy import Function
>>> from sympy.solvers.deutils import ode_order
>>> from sympy.abc import x
>>> f, g = map(Function, ['f', 'g'])
>>> ode_order(f(x).diff(x, 2) + f(x).diff(x)**2 +
... f(x).diff(x), f(x))
2
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), f(x))
2
```

(continues on next page)

(continued from previous page)

```
>>> ode_order(f(x).diff(x, 2) + g(x).diff(x, 3), g(x))
3
```

Recurrence Equations

sympy.solvers.recurr.**rsolve**(*f*, *y*, *init=None*)

Solve univariate recurrence with rational coefficients.

Given k-th order linear recurrence $\mathrm{L}\,y = f$, or equivalently:

$$a_k(n)y(n+k) + a_{k-1}(n)y(n+k-1) + \cdots + a_0(n)y(n) = f(n)$$

where $a_i(n)$, for $i = 0, \ldots, k$, are polynomials or rational functions in n, and f is a hypergeometric function or a sum of a fixed number of pairwise dissimilar hypergeometric terms in n, finds all solutions or returns `None`, if none were found.

Initial conditions can be given as a dictionary in two forms:

(1) `{ n_0 : v_0, n_1 : v_1, ..., n_m : v_m}`

(2) `{y(n_0) : v_0, y(n_1) : v_1, ..., y(n_m) : v_m}`

or as a list `L` of values:

```
L = [v_0, v_1, ..., v_m]
```

where `L[i] = v_i`, for $i = 0, \ldots, m$, maps to $y(n_i)$.

Examples

Lets consider the following recurrence:

$$(n-1)y(n+2) - (n^2+3n-2)y(n+1) + 2n(n+1)y(n) = 0$$

```
>>> from sympy import Function, rsolve
>>> from sympy.abc import n
>>> y = Function('y')
```

```
>>> f = (n - 1)*y(n + 2) - (n**2 + 3*n - 2)*y(n + 1) + 2*n*(n + 1)*y(n)
```

```
>>> rsolve(f, y(n))
2**n*C0 + C1*factorial(n)
```

```
>>> rsolve(f, y(n), {y(0):0, y(1):3})
3*2**n - 3*factorial(n)
```

See also:

rsolve_poly (page 853), *rsolve_ratio* (page 854), *rsolve_hyper* (page 855)

sympy.solvers.recurr.**rsolve_poly**(*coeffs*, *f*, *n*, *shift=0*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $\mathrm{L}\,y = f$, 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 $\mathrm{L}\,y = f$.

There are two methods for computing the polynomial solutions. If the degree bound is relatively small, i.e. it's smaller than or equal to the order of the recurrence, then naive method of undetermined coefficients is being used. This gives system of algebraic equations with $N+1$ unknowns.

In the other case, the algorithm performs transformation of the initial equation to an equivalent one, for which the system of algebraic equations has only r indeterminates. This method is quite sophisticated (in comparison with the naive one) and was invented together by Abramov, Bronstein and Petkovsek.

It is possible to generalize the algorithm implemented here to the case of linear q-difference and differential equations.

Lets say that we would like to compute m-th Bernoulli polynomial up to a constant. For this we can use $b(n+1) - b(n) = mn^{m-1}$ recurrence, which has solution $b(n) = B_m + C$. For example:

```
>>> from sympy import Symbol, rsolve_poly
>>> n = Symbol('n', integer=True)
```

```
>>> rsolve_poly([-1, 1], 4*n**3, n)
C0 + n**4 - 2*n**3 + n**2
```

References

[R797], [R798], [R799]

sympy.solvers.recurr.**rsolve_ratio**(*coeffs*, *f*, *n*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $\mathrm{L}\,y = f$, 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 $\mathrm{L}\,y = f$.

(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.

Examples

```
>>> from sympy.abc import x
>>> from sympy.solvers.recurr import rsolve_ratio
>>> rsolve_ratio([-2*x**3 + x**2 + 2*x - 1, 2*x**3 + x**2 - 6*x,
... - 2*x**3 - 11*x**2 - 18*x - 9, 2*x**3 + 13*x**2 + 22*x + 8], 0, x)
C2*(2*x - 3)/(2*(x**2 - 1))
```

See also:

rsolve_hyper (page 855)

References

[R800]

sympy.solvers.recurr.**rsolve_hyper**(*coeffs*, *f*, *n*, ***hints*)

Given linear recurrence operator L of order k with polynomial coefficients and inhomogeneous equation $\mathrm{L}y = f$ 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 $\mathrm{L}y = f$, 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

Examples

```
>>> from sympy.solvers import rsolve_hyper
>>> from sympy.abc import x
```

```
>>> rsolve_hyper([-1, -1, 1], 0, x)
C0*(1/2 - sqrt(5)/2)**x + C1*(1/2 + sqrt(5)/2)**x
```

```
>>> rsolve_hyper([-1, 1], 1 + x, x)
C0 + x*(x + 1)/2
```

References

[R801], [R802]

Systems of Polynomial Equations

sympy.solvers.polysys.**solve_poly_system**(*seq*, **gens*, *strict=False*, ***args*)

Solve a system of polynomial equations.

Parameters

seq: a list/tuple/set

Listing all the equations that are needed to be solved

gens: generators

generators of the equations in seq for which we want the solutions

strict: a boolean (default is False)

if strict is True, NotImplementedError will be raised if the solution is known to be incomplete (which can occur if not all solutions are expressible in radicals)

args: Keyword arguments

Special options for solving the equations.

Returns

List[Tuple]

A List of tuples. Solutions for symbols that satisfy the equations listed in seq

Examples

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

```
>>> solve_poly_system([x*y - 2*y, 2*y**2 - x**2], x, y)
[(0, 0), (2, -sqrt(2)), (2, sqrt(2))]
```

```
>>> solve_poly_system([x**5 - x + y**3, y**2 - 1], x, y, strict=True)
Traceback (most recent call last):
...
UnsolvableFactorError
```

sympy.solvers.polysys.**solve_triangulated**(*polys*, **gens*, ***args*)

Solve a polynomial system using Gianni-Kalkbrenner algorithm.

The algorithm proceeds by computing one Groebner basis in the ground domain and then by iteratively computing polynomial factorizations in appropriately constructed algebraic extensions of the ground domain.

Parameters

polys: a list/tuple/set

Listing all the equations that are needed to be solved

gens: generators

generators of the equations in polys for which we want the solutions

args: Keyword arguments

Special options for solving the equations

Returns

List[Tuple]

A List of tuples. Solutions for symbols that satisfy the equations listed in polys

Examples

```
>>> from sympy import solve_triangulated
>>> from sympy.abc import x, y, z
```

```
>>> F = [x**2 + y + z - 1, x + y**2 + z - 1, x + y + z**2 - 1]
```

```
>>> solve_triangulated(F, x, y, z)
[(0, 0, 1), (0, 1, 0), (1, 0, 0)]
```

References

1. Patrizia Gianni, Teo Mora, Algebraic Solution of System of Polynomial Equations using Groebner Bases, AAECC-5 on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, LNCS 356 247–257, 1989

Diophantine Equations (DEs)

See *Diophantine* (page 714)

Inequalities

See *Inequality Solvers* (page 750)

Solveset

This is the official documentation of the `solveset` module in solvers. It contains the frequently asked questions about our new module to solve equations.

What's wrong with solve():

SymPy already has a pretty powerful `solve` function. But it has some deficiencies. For example:

1. It doesn't have a consistent output for various types of solutions It needs to return a lot of types of solutions consistently:
   - Single solution : $x = 1$
   - Multiple solutions: $x^2 = 1$
   - No Solution: $x^2 + 1 = 0; x \in \mathbb{R}$
   - Interval of solution: $\lfloor x \rfloor = 0$
   - Infinitely many solutions: $\sin(x) = 0$
   - Multivariate functions with point solutions: $x^2 + y^2 = 0$
   - Multivariate functions with non-point solution: $x^2 + y^2 = 1$
   - System of equations: $x + y = 1$ and $x - y = 0$
   - Relational: $x > 0$
   - And the most important case: "We don't Know"
2. The input API has a lot of parameters and it can be difficult to use.
3. There are cases like finding the maxima and minima of function using critical points where it is important to know if it has returned all the solutions. `solve` does not guarantee this.

Why Solveset?

- `solveset` has an alternative consistent input and output interface: `solveset` returns a set object and a set object takes care of all types of output. For cases where it does not "know" all the solutions a `ConditionSet` with a partial solution is returned. For input it only takes the equation, the variables to solve for and the optional argument `domain` over which the equation is to be solved.
- `solveset` can return infinitely many solutions. For example solving for $\sin(x) = 0$ returns $\{2n\pi | n \in \mathbb{Z}\} \cup \{2n\pi + \pi | n \in \mathbb{Z}\}$, whereas `solve` only returns $[0, \pi]$.
- There is a clear code level and interface level separation between solvers for equations in the complex domain and the real domain. For example solving $e^x = 1$ when x is to be solved in the complex domain, returns the set of all solutions, that is $\{2ni\pi | n \in \mathbb{Z}\}$, whereas if x is to be solved in the real domain then only $\{0\}$ is returned.

Why do we use Sets as an output type?

SymPy has a well developed sets module, which can represent most of the set containers in mathematics such as:

- *FiniteSet* (page 1197)

  Represents a finite set of discrete numbers.

- *Interval* (page 1194)

  Represents a real interval as a set.

- *ProductSet* (page 1199)

  Represents a Cartesian product of sets.

- *ImageSet* (page 1206)

  Represents the image of a set under a mathematical function

```
>>> from sympy import ImageSet, S, Lambda
>>> from sympy.abc import x
>>> squares = ImageSet(Lambda(x, x**2), S.Naturals)  # {x**2 for x in N}
>>> 4 in squares
True
```

- *ComplexRegion* (page 1209)

  Represents the set of all complex numbers in a region in the Argand plane.

- *ConditionSet* (page 1215)

  Represents the set of elements, which satisfies a given condition.

Also, the predefined set classes such as:

- *Naturals* (page 1203), $mathbb{N}

  Represents the natural numbers (or counting numbers), which are all positive integers starting from 1.

- *Naturals0* (page 1204), $\mathbb{N}_0$

  Represents the whole numbers, which are all the non-negative integers, inclusive of 0.

- *Integers* (page 1204), $\mathbb{Z}$

  Represents all integers: positive, negative and zero.

- *Reals* (page 1205), $\mathbb{R}$

  Represents the set of all real numbers.

- *Complexes* (page 1206), $\mathbb{C}$

  Represents the set of all complex numbers.

- *EmptySet* (page 1202), $\emptyset$

  Represents the empty set.

The above six sets are available as Singletons, like `S.Integers`.

It is capable of most of the set operations in mathematics:

- `Union`
- `Intersection`
- `Complement`
- `SymmetricDifference`

The main reason for using sets as output to solvers is that it can consistently represent many types of solutions. For the single variable case it can represent:

- No solution (by the empty set).
- Finitely many solutions (by `FiniteSet`).
- Infinitely many solutions, both countably and uncountably infinite solutions (using the `ImageSet` module).
- `Interval`
- There can also be bizarre solutions to equations like the set of rational numbers.

No other Python object (list, dictionary, generator, Python sets) provides the flexibility of mathematical sets which our sets module tries to emulate. The second reason to use sets is that they are close to the entities which mathematicians deal with and it makes it easier to reason about them. Set objects conform to Pythonic conventions when possible, i.e., `x in A` and `for i in A` both work when they can be computed. Another advantage of using objects closer to mathematical entities is that the user won't have to "learn" our representation and she can have her expectations transferred from her mathematical experience.

For the multivariate case we represent solutions as a set of points in a n-dimensional space and a point is represented by a `FiniteSet` of ordered tuples, which is a point in $\mathbb{R}^n$ or $\mathbb{C}^n$.

Please note that, the general `FiniteSet` is unordered, but a `FiniteSet` with a tuple as its only argument becomes ordered, since a tuple is ordered. So the order in the tuple is mapped to a pre-defined order of variables while returning solutions.

For example:

```
>>> from sympy import FiniteSet
>>> FiniteSet(1, 2, 3)   # Unordered
{1, 2, 3}
>>> FiniteSet((1, 2, 3))  # Ordered
{(1, 2, 3)}
```

Why not use dicts as output?

Dictionary are easy to deal with programmatically but mathematically they are not very precise and use of them can quickly lead to inconsistency and a lot of confusion. For example:

- There are a lot of cases where we don't know the complete solution and we may like to output a partial solution, consider the equation $fg = 0$. The solution of this equation is the union of the solution of the following two equations: $f = 0$, $g = 0$. Let's say that we are able to solve $f = 0$ but solving $g = 0$ isn't supported yet. In this case we cannot represent partial solution of the given equation $fg = 0$ using dicts. This problem is solved with sets using a `ConditionSet` object:

  $sol_f \cup \{x | x \in \mathbb{R} \wedge g = 0\}$, where sol_f is the solution of the equation $f = 0$.

- Using a dict may lead to surprising results like:

  - `solve(Eq(x**2, 1), x) != solve(Eq(y**2, 1), y)`

    Mathematically, this doesn't make sense. Using `FiniteSet` here solves the problem.

- It also cannot represent solutions for equations like $|x| < 1$, which is a disk of radius 1 in the Argand Plane. This problem is solved using complex sets implemented as `ComplexRegion`.

Input API of `solveset`

`solveset` has simpler input API, unlike `solve`. It takes a maximum of three arguments:

`solveset(equation, variable=None, domain=S.Complexes)`

Equation(s)
The equation(s) to solve.

Variable(s)
The variable(s) for which the equation is to be solved.

Domain
The domain in which the equation is to be solved.

`solveset` removes the `flags` argument of `solve`, which had made the input API more complicated and output API inconsistent.

What is this domain argument about?

Solveset is designed to be independent of the assumptions on the variable being solved for and instead, uses the `domain` argument to decide the solver to dispatch the equation to, namely `solveset_real` or `solveset_complex`. It's unlike the old `solve` which considers the assumption on the variable.

```
>>> from sympy import solveset, S
>>> from sympy.abc import x
>>> solveset(x**2 + 1, x) # domain=S.Complexes is default
{-I, I}
>>> solveset(x**2 + 1, x, domain=S.Reals)
EmptySet
```

What are the general methods employed by solveset to solve an equation?

Solveset uses various methods to solve an equation, here is a brief overview of the methodology:

- The `domain` argument is first considered to know the domain in which the user is interested to get the solution.
- If the given function is a relational (`>=`, `<=`, `>`, `<`), and the domain is real, then `solve_univariate_inequality` and solutions are returned. Solving for complex solutions of inequalities, like $x^2 < 0$ is not yet supported.
- Based on the `domain`, the equation is dispatched to one of the two functions `solveset_real` or `solveset_complex`, which solves the given equation in the complex or real domain, respectively.
- If the given expression is a product of two or more functions, like say $gh = 0$, then the solution to the given equation is the Union of the solution of the equations $g = 0$ and $h = 0$, if and only if both g and h are finite for a finite input. So, the solution is built up recursively.
- If the function is trigonometric or hyperbolic, the function `_solve_real_trig` is called, which solves it by converting it to complex exponential form.
- The function is now checked if there is any instance of a `Piecewise` expression, if it is, then it's converted to explicit expression and set pairs and then solved recursively.
- The respective solver now tries to invert the equation using the routines `invert_real` and `invert_complex`. These routines are based on the concept of mathematical inverse (though not exactly). It reduces the real/complex valued equation $f(x) = y$ to a set of equations: $\{g(x) = h_1(y), g(x) = h_2(y), ..., g(x) = h_n(y)\}$ where $g(x)$ is a simpler function than $f(x)$. There is some work needed to be done in this to find invert of more complex expressions.
- After the invert, the equations are checked for radical or Abs (Modulus), then the method `_solve_radical` tries to simplify the radical, by removing it using techniques like squaring, cubing etc, and `_solve_abs` solves nested Modulus by considering the positive and negative variants, iteratively.
- If none of the above method is successful, then methods of polynomial is used as follows:
  - The method to solve the rational function, `_solve_as_rational`, is called. Based on the domain, the respective poly solver `_solve_as_poly_real` or `_solve_as_poly_complex` is called to solve `f` as a polynomial.
  - The underlying method `_solve_as_poly` solves the equation using polynomial techniques if it's already a polynomial equation or, with a change of variables, can be made so.
- The final solution set returned by `solveset` is the intersection of the set of solutions found above and the input domain.

How do we manipulate and return an infinite solution?

- In the real domain, we use our `ImageSet` class in the sets module to return infinite solutions. `ImageSet` is an image of a set under a mathematical function. For example, to represent the solution of the equation $\sin(x) = 0$, we can use the `ImageSet` as:

```
>>> from sympy import ImageSet, Lambda, pi, S, Dummy, pprint
>>> n = Dummy('n')
>>> pprint(ImageSet(Lambda(n, 2*pi*n), S.Integers), use_unicode=True)
{2·n·π | n ∈ ℤ}
```

  Where `n` is a dummy variable. It is basically the image of the set of integers under the function $2\pi n$.

- In the complex domain, we use complex sets, which are implemented as the `ComplexRegion` class in the sets module, to represent infinite solution in the Argand plane. For example to represent the solution of the equation $|z| = 1$, which is a unit circle, we can use the `ComplexRegion` as:

```
>>> from sympy import ComplexRegion, FiniteSet, Interval, pi, pprint
>>> pprint(ComplexRegion(FiniteSet(1)*Interval(0, 2*pi), polar=True),␣
↪use_unicode=True)
{r·(i·sin(θ) + cos(θ)) | r, θ ∈ {1} × [0, 2·π)}
```

  Where the `FiniteSet` in the `ProductSet` is the range of the value of r, which is the radius of the circle and the `Interval` is the range of θ, the angle from the x axis representing a unit circle in the Argand plane.

  Note: We also have non-polar form notation for representing solution in rectangular form. For example, to represent first two quadrants in the Argand plane, we can write the `ComplexRegion` as:

```
>>> from sympy import ComplexRegion, Interval, pi, oo, pprint
>>> pprint(ComplexRegion(Interval(-oo, oo)*Interval(0, oo)), use_
↪unicode=True)
{x + y·i | x, y ∈ (-∞, ∞) × [0, ∞)}
```

  where the Intervals are the range of x and y for the set of complex numbers $x + iy$.

How does `solveset` ensure that it is not returning any wrong solution?

Solvers in a Computer Algebra System are based on heuristic algorithms, so it's usually very hard to ensure 100% percent correctness, in every possible case. However there are still a lot of cases where we can ensure correctness. Solveset tries to verify correctness wherever it can. For example:

Consider the equation $|x| = n$. A naive method to solve this equation would return `{-n, n}` as its solution, which is not correct since `{-n, n}` can be its solution if and only if `n` is positive. Solveset returns this information as well to ensure correctness.

```
>>> from sympy import symbols, S, pprint, solveset
>>> x, n = symbols('x, n')
>>> pprint(solveset(abs(x) - n, x, domain=S.Reals), use_unicode=True)
{x | x ∈ {-n, n} ∧ (n ∈ [0, ∞))}
```

Though, there still a lot of work needs to be done in this regard.

Search based solver and step-by-step solution

Note: This is under Development.

After the introduction of *ConditionSet* (page 1215), the solving of equations can be seen as set transformations. Here is an abstract view of the things we can do to solve equations.

- Apply various set transformations on the given set.
- Define a metric of the usability of solutions, or a notion of some solutions being better than others.
- Different transformations would be the nodes of a tree.
- Suitable searching techniques could be applied to get the best solution.

`ConditionSet` gives us the ability to represent unevaluated equations and inequalities in forms like $\{x|f(x) = 0; x \in S\}$ and $\{x|f(x) > 0; x \in S\}$ but a more powerful thing about `ConditionSet` is that it allows us to write the intermediate steps as set to set transformation. Some of the transformations are:

- Composition: $\{x|f(g(x)) = 0; x \in S\} \Rightarrow \{x|g(x) = y; x \in S, y \in \{z|f(z) = 0; z \in S\}\}$
- **Polynomial Solver:** $\{x|P(x) = 0; x \in S\} \Rightarrow \{x_1, x_2, ..., x_n\} \cap S$**,**
  where x_i are roots of $P(x)$.
- Invert solver: $\{x|f(x) = 0; x \in S\} \Rightarrow \{g(0)|$ all g such that $f(g(x)) = x\}$
- **logcombine:** $\{x|\log(f(x)) + \log(g(x)); x \in S\}$
  $\Rightarrow \{x|\log(f(x).g(x)); x \in S\}$ if $f(x) > 0$ and $g(x) > 0$ $\Rightarrow$ $\{x|\log(f(x)) + \log(g(x)); x \in S\}$ otherwise
- **product solve:** $\{x|f(x)g(x) = 0; x \in S\}$
  $\Rightarrow \{x|f(x) = 0; x \in S\} U \{x|g(x) = 0; x \in S\}$ given $f(x)$ and $g(x)$ are bounded.
  $\Rightarrow \{x|f(x)g(x) = 0; x \in S\},$ otherwise

Since the output type is same as the input type any composition of these transformations is also a valid transformation. And our aim is to find the right sequence of compositions (given the atoms) which transforms the given condition set to a set which is not a condition set i.e., FiniteSet, Interval, Set of Integers and their Union, Intersection, Complement or ImageSet. We can assign a cost function to each set, such that, the more desirable that form of set is to us, the less the value of the cost function. This way our problem is now reduced to finding the path from the initial ConditionSet to the lowest valued set on a graph where the atomic transformations forms the edges.

How do we deal with cases where only some of the solutions are known?

Creating a universal equation solver, which can solve each and every equation we encounter in mathematics is an ideal case for solvers in a Computer Algebra System. When cases which are not solved or can only be solved incompletely, a `ConditionSet` is used and acts as an unevaluated solveset object.

Note that, mathematically, finding a complete set of solutions for an equation is undecidable. See Richardson's theorem.

`ConditionSet` is basically a Set of elements which satisfy a given condition. For example, to represent the solutions of the equation in the real domain:

$$(x^2 - 4)(\sin(x) + x)$$

We can represent it as:

$\{-2, 2\} \cup \{x | x \in \mathbb{R} \wedge x + \sin(x) = 0\}$

What is the plan for solve and solveset?

There are still a few things `solveset` can't do, which `solve` can, such as solving nonlinear multivariate & LambertW type equations. Hence, it's not yet a perfect replacement for `solve`. As the algorithms in `solveset` mature, `solveset` may be able to be used within `solve` to replace some of its algorithms.

How are symbolic parameters handled in solveset?

Solveset is in its initial phase of development, so the symbolic parameters aren't handled well for all the cases, but some work has been done in this regard to depict our ideology towards symbolic parameters. As an example, consider the solving of $|x| = n$ for real x, where n is a symbolic parameter. Solveset returns the value of x considering the domain of the symbolic parameter n as well:

$$([0, \infty) \cap \{n\}) \cup ((-\infty, 0] \cap \{-n\}).$$

This simply means n is the solution only when it belongs to the `Interval` $[0, \infty)$ and $-n$ is the solution only when $-n$ belongs to the `Interval` $(-\infty, 0]$.

There are other cases to address too, like solving $2^x + (a - 2)$ for x where a is a symbolic parameter. As of now, It returns the solution as an intersection with $\mathbb{R}$, which is trivial, as it doesn't reveal the domain of a in the solution.

Recently, we have also implemented a function to find the domain of the expression in a FiniteSet (Intersection with the interval) in which it is not-empty. It is a useful addition for dealing with symbolic parameters. For example:

```
>>> from sympy import Symbol, FiniteSet, Interval, not_empty_in,␣
↪sqrt, oo
>>> from sympy.abc import x
>>> not_empty_in(FiniteSet(x/2).intersect(Interval(0, 1)), x)
Interval(0, 2)
>>> not_empty_in(FiniteSet(x, x**2).intersect(Interval(1, 2)), x)
Union(Interval(1, 2), Interval(-sqrt(2), -1))
```

References

Solveset Module Reference

Use *solveset()* (page 866) to solve equations or expressions (assumed to be equal to 0) for a single variable. Solving an equation like $x^2 == 1$ can be done as follows:

```
>>> from sympy import solveset
>>> from sympy import Symbol, Eq
>>> x = Symbol('x')
>>> solveset(Eq(x**2, 1), x)
{-1, 1}
```

Or one may manually rewrite the equation as an expression equal to 0:

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

The first argument for *solveset()* (page 866) is an expression (equal to zero) or an equation and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solveset.`**solveset**(*f*, *symbol=None*, *domain=Complexes*)

Solves a given inequality or equation with set as output

Parameters

f : Expr or a relational.

The target equation or inequality

symbol : Symbol

The variable for which the equation is solved

domain : Set

The domain over which the equation is solved

Returns

Set

A set of values for $symbol$ for which f is True or is equal to zero. An *EmptySet* (page 1202) is returned if f is False or nonzero. A *ConditionSet* (page 1215) is returned as unsolved object if algorithms to evaluate complete solution are not yet implemented.

`solveset` claims to be complete in the solution set that it returns.

Raises

NotImplementedError

The algorithms to solve inequalities in complex domain are not yet implemented.

ValueError

The input is not valid.

RuntimeError

It is a bug, please report to the github issue tracker.

Notes

Python interprets 0 and 1 as False and True, respectively, but in this function they refer to solutions of an expression. So 0 and 1 return the domain and EmptySet, respectively, while True and False return the opposite (as they are assumed to be solutions of relational expressions).

Examples

```
>>> from sympy import exp, sin, Symbol, pprint, S, Eq
>>> from sympy.solvers.solveset import solveset, solveset_real
```

- The default domain is complex. Not specifying a domain will lead to the solving of the equation in the complex domain (and this is not affected by the assumptions on the symbol):

```
>>> x = Symbol('x')
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

```
>>> x = Symbol('x', real=True)
>>> pprint(solveset(exp(x) - 1, x), use_unicode=False)
{2*n*I*pi | n in Integers}
```

- If you want to use `solveset` to solve the equation in the real domain, provide a real domain. (Using `solveset_real` does this automatically.)

```
>>> R = S.Reals
>>> x = Symbol('x')
>>> solveset(exp(x) - 1, x, R)
{0}
>>> solveset_real(exp(x) - 1, x)
{0}
```

The solution is unaffected by assumptions on the symbol:

```
>>> p = Symbol('p', positive=True)
>>> pprint(solveset(p**2 - 4))
{-2, 2}
```

When a *ConditionSet* (page 1215) is returned, symbols with assumptions that would alter the set are replaced with more generic symbols:

```
>>> i = Symbol('i', imaginary=True)
>>> solveset(Eq(i**2 + i*sin(i), 1), i, domain=S.Reals)
ConditionSet(_R, Eq(_R**2 + _R*sin(_R) - 1, 0), Reals)
```

- Inequalities can be solved over the real domain only. Use of a complex domain leads to a NotImplementedError.

```
>>> solveset(exp(x) > 1, x, R)
Interval.open(0, oo)
```

See also:

solveset_real **(page 868)**
solver for real domain

solveset_complex **(page 868)**
solver for complex domain

sympy.solvers.solveset.**solveset_real**(*f*, *symbol*)

sympy.solvers.solveset.**solveset_complex**(*f*, *symbol*)

sympy.solvers.solveset.**invert_real**(*f_x*, *y*, *x*)
Inverts a real-valued function. Same as *invert_complex()* (page 868), but sets the domain to `S.Reals` before inverting.

sympy.solvers.solveset.**invert_complex**(*f_x*, *y*, *x*, *domain=Complexes*)
Reduce the complex valued equation $f(x) = y$ to a set of equations

$$\{g(x) = h_1(y),\ g(x) = h_2(y),\ \ldots,\ g(x) = h_n(y)\}$$

where $g(x)$ is a simpler function than $f(x)$. The return value is a tuple $(g(x), \text{set}_h)$, where $g(x)$ is a function of x and set_h is the set of function $\{h_1(y), h_2(y), \ldots, h_n(y)\}$. Here, y is not necessarily a symbol.

set_h contains the functions, along with the information about the domain in which they are valid, through set operations. For instance, if $y = |x| - n$ is inverted in the real domain, then set_h is not simply $\{-n, n\}$ as the nature of n is unknown; rather, it is:

$$ left(left[0, inftyright) cap left{nright}right) cup
left(left(-infty, 0right] cap left{- nright}right)$$

By default, the complex domain is used which means that inverting even seemingly simple functions like $\exp(x)$ will give very different results from those obtained in the real domain. (In the case of $\exp(x)$, the inversion via log is multi-valued in the complex domain, having infinitely many branches.)

If you are working with real values only (or you are not sure which function to use) you should probably set the domain to `S.Reals` (or use `invert_real` which does that automatically).

Examples

```
>>> from sympy.solvers.solveset import invert_complex, invert_real
>>> from sympy.abc import x, y
>>> from sympy import exp
```

When does exp(x) == y?

```
>>> invert_complex(exp(x), y, x)
(x, ImageSet(Lambda(_n, I*(2*_n*pi + arg(y)) + log(Abs(y))), Integers))
>>> invert_real(exp(x), y, x)
(x, Intersection({log(y)}, Reals))
```

When does exp(x) == 1?

```
>>> invert_complex(exp(x), 1, x)
(x, ImageSet(Lambda(_n, 2*_n*I*pi), Integers))
>>> invert_real(exp(x), 1, x)
(x, {0})
```

See also:

invert_real (page 868), *invert_complex* (page 868)

sympy.solvers.solveset.**domain_check**(*f*, *symbol*, *p*)

Returns False if point p is infinite or any subexpression of f is infinite or becomes so after replacing symbol with p. If none of these conditions is met then True will be returned.

Examples

```
>>> from sympy import Mul, oo
>>> from sympy.abc import x
>>> from sympy.solvers.solveset import domain_check
>>> g = 1/(1 + (1/(x + 1))**2)
>>> domain_check(g, x, -1)
False
>>> domain_check(x**2, x, 0)
True
>>> domain_check(1/x, x, oo)
False
```

- The function relies on the assumption that the original form of the equation has not been changed by automatic simplification.

```
>>> domain_check(x/x, x, 0) # x/x is automatically simplified to 1
True
```

- To deal with automatic evaluations use evaluate=False:

```
>>> domain_check(Mul(x, 1/x, evaluate=False), x, 0)
False
```

sympy.solvers.solveset.**solvify**(*f*, *symbol*, *domain*)

Solves an equation using solveset and returns the solution in accordance with the *solve* output API.

Returns

We classify the output based on the type of solution returned by *solveset*.

Raises

NotImplementedError

A ConditionSet is the input.

Solution | Output

FiniteSet | list

ImageSet, | list (if f is periodic) Union |

Union | list (with FiniteSet)

EmptySet | empty list

Others | None

Examples

```
>>> from sympy.solvers.solveset import solvify
>>> from sympy.abc import x
>>> from sympy import S, tan, sin, exp
>>> solvify(x**2 - 9, x, S.Reals)
[-3, 3]
>>> solvify(sin(x) - 1, x, S.Reals)
[pi/2]
>>> solvify(tan(x), x, S.Reals)
[0]
>>> solvify(exp(x) - 1, x, S.Complexes)
```

```
>>> solvify(exp(x) - 1, x, S.Reals)
[0]
```

linear_eq_to_matrix

sympy.solvers.solveset.**linear_eq_to_matrix**(*equations*, **symbols*)

Converts a given System of Equations into Matrix form. Here *equations* must be a linear system of equations in *symbols*. Element `M[i, j]` corresponds to the coefficient of the jth symbol in the ith equation.

The Matrix form corresponds to the augmented matrix form. For example:

$$4x + 2y + 3z = 1$$

$$3x + y + z = -6$$

$$2x + 4y + 9z = 2$$

This system will return A and b as:

$$ A = left[begin{array}{ccc}
4 & 2 & 3 \ 3 & 1 & 1 \ 2 & 4 & 9 end{array}right] b = left[begin{array}{c} 1 \ -6 \ 2 end{array}right] $$

The only simplification performed is to convert `Eq(a, b)` $\Rightarrow a - b$.

Raises

NonlinearError

The equations contain a nonlinear term.

ValueError

The symbols are not given or are not unique.

Examples

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

The coefficients (numerical or symbolic) of the symbols will be returned as matrices:

```
>>> eqns = [c*x + z - 1 - c, y + z, x - y]
>>> A, b = linear_eq_to_matrix(eqns, [x, y, z])
>>> A
Matrix([
[c,  0, 1],
[0,  1, 1],
[1, -1, 0]])
>>> b
Matrix([
[c + 1],
[    0],
[    0]])
```

This routine does not simplify expressions and will raise an error if nonlinearity is encountered:

```
>>> eqns = [
...     (x**2 - 3*x)/(x - 3) - 3,
...     y**2 - 3*y - y*(y - 4) + x - 4]
>>> linear_eq_to_matrix(eqns, [x, y])
Traceback (most recent call last):
...
NonlinearError:
The term (x**2 - 3*x)/(x - 3) is nonlinear in {x, y}
```

Simplifying these equations will discard the removable singularity in the first, reveal the linear structure of the second:

```
>>> [e.simplify() for e in eqns]
[x - 3, x + y - 4]
```

Any such simplification needed to eliminate nonlinear terms must be done before calling this routine.

linsolve

sympy.solvers.solveset.**linsolve**(*system*, **symbols*)

Solve system of N linear equations with M variables; both underdetermined and overdetermined systems are supported. The possible number of solutions is zero, one or infinite. Zero solutions throws a ValueError, whereas infinite solutions are represented parametrically in terms of the given symbols. For unique solution a *FiniteSet* (page 1197) of ordered tuples is returned.

All standard input formats are supported: For the given set of equations, the respective input types are given below:

$$3x + 2y - z = 1$$

$$2x - 2y + 4z = -2$$

$$2x - y + 2z = 0$$

- Augmented matrix form, `system` given below:

$$ text{system} = left[{array}{cccc}
3 & 2 & -1 & 1\ 2 & -2 & 4 & -2\ 2 & -1 & 2 & 0 end{array}right] $$

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

- List of equations form

```
system  =  [3x + 2y - z - 1, 2x - 2y + 4z + 2, 2x - y + 2z]
```

- Input A and b in matrix form (from $Ax = b$) are given as:

$$ A = left[begin{array}{ccc}
3 & 2 & -1 \ 2 & -2 & 4 \ 2 & -1 & 2 end{array}right] b = left[begin{array}{c} 1 \
-2 \ 0 end{array}right] $$

```
A = Matrix([[3, 2, -1], [2, -2, 4], [2, -1, 2]])
b = Matrix([[1], [-2], [0]])
system = (A, b)
```

Symbols can always be passed but are actually only needed when 1) a system of equations is being passed and 2) the system is passed as an underdetermined matrix and one wants to control the name of the free variables in the result. An error is raised if no symbols are used for case 1, but if no symbols are provided for case 2, internally generated symbols will be provided. When providing symbols for case 2, there should be at least as many symbols are there are columns in matrix A.

The algorithm used here is Gauss-Jordan elimination, which results, after elimination, in a row echelon form matrix.

Returns

A FiniteSet containing an ordered tuple of values for the unknowns for which the *system* has a solution. (Wrapping the tuple in FiniteSet is used to maintain a consistent output format throughout solveset.)

Returns EmptySet, if the linear system is inconsistent.

Raises

ValueError

The input is not valid. The symbols are not given.

Examples

```
>>> from sympy import Matrix, linsolve, symbols
>>> x, y, z = symbols("x, y, z")
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> A
Matrix([
[1, 2,  3],
[4, 5,  6],
[7, 8, 10]])
>>> b
Matrix([
[3],
[6],
[9]])
>>> linsolve((A, b), [x, y, z])
{(-1, 2, 0)}
```

- Parametric Solution: In case the system is underdetermined, the function will return a parametric solution in terms of the given symbols. Those that are free will be returned unchanged. e.g. in the system below, z is returned as the solution for variable z; it can take on any value.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
{(z - 1, 2 - 2*z, z)}
```

If no symbols are given, internally generated symbols will be used. The `tau0` in the third position indicates (as before) that the third variable – whatever it is named – can take on any value:

```
>>> linsolve((A, b))
{(tau0 - 1, 2 - 2*tau0, tau0)}
```

- List of equations as input

```
>>> Eqns = [3*x + 2*y - z - 1, 2*x - 2*y + 4*z + 2, - x + y/2 - z]
>>> linsolve(Eqns, x, y, z)
{(1, -2, -2)}
```

- Augmented matrix as input

```
>>> aug = Matrix([[2, 1, 3, 1], [2, 6, 8, 3], [6, 8, 18, 5]])
>>> aug
Matrix([
[2, 1,  3, 1],
[2, 6,  8, 3],
[6, 8, 18, 5]])
>>> linsolve(aug, x, y, z)
{(3/10, 2/5, 0)}
```

- Solve for symbolic coefficients

```
>>> a, b, c, d, e, f = symbols('a, b, c, d, e, f')
>>> eqns = [a*x + b*y - c, d*x + e*y - f]
>>> linsolve(eqns, x, y)
{((-b*f + c*e)/(a*e - b*d), (a*f - c*d)/(a*e - b*d))}
```

- A degenerate system returns solution as set of given symbols.

```
>>> system = Matrix(([0, 0, 0], [0, 0, 0], [0, 0, 0]))
>>> linsolve(system, x, y)
{(x, y)}
```

- For an empty system linsolve returns empty set

```
>>> linsolve([], x)
EmptySet
```

- An error is raised if, after expansion, any nonlinearity is detected:

```
>>> linsolve([x*(1/x - 1), (y - 1)**2 - y**2 + 1], x, y)
{(1, 1)}
>>> linsolve([x**2 - 1], x)
Traceback (most recent call last):
...
NonlinearError:
nonlinear term encountered: x**2
```

nonlinsolve

sympy.solvers.solveset.**nonlinsolve**(*system*, **symbols*)

Solve system of N nonlinear equations with M variables, which means both under and overdetermined systems are supported. Positive dimensional system is also supported (A system with infinitely many solutions is said to be positive-dimensional). In a positive dimensional system the solution will be dependent on at least one symbol. Returns both real solution and complex solution (if they exist).

Parameters

system : list of equations

The target system of equations

symbols : list of Symbols

symbols should be given as a sequence eg. list

Returns

A *FiniteSet* (page 1197) of ordered tuple of values of $symbols$ for which the $system$

has solution. Order of values in the tuple is same as symbols present in the parameter $symbols$.

Please note that general *FiniteSet* (page 1197) is unordered, the solution returned here is not simply a *FiniteSet* (page 1197) of solutions, rather it is a *FiniteSet* (page 1197) of ordered tuple, i.e. the first and only argument to *FiniteSet* (page 1197) is a tuple of solutions, which is ordered, and, hence ,the returned solution is ordered.

Also note that solution could also have been returned as an ordered tuple, FiniteSet is just a wrapper `{}` around the tuple. It has no other significance except for the fact it is just used to maintain a consistent output format throughout the solveset.

For the given set of equations, the respective input types are given below:

$$xy - 1 = 0$$

$$4x^2 + y^2 - 5 = 0$$

```
system  = [x*y - 1, 4*x**2 + y**2 - 5]
symbols = [x, y]
```

Raises

ValueError

The input is not valid. The symbols are not given.

AttributeError

The input symbols are not $Symbol$ type.

Examples

```
>>> from sympy import symbols, nonlinsolve
>>> x, y, z = symbols('x, y, z', real=True)
>>> nonlinsolve([x*y - 1, 4*x**2 + y**2 - 5], [x, y])
{(-1, -1), (-1/2, -2), (1/2, 2), (1, 1)}
```

1. Positive dimensional system and complements:

```
>>> from sympy import pprint
>>> from sympy.polys.polytools import is_zero_dimensional
>>> a, b, c, d = symbols('a, b, c, d', extended_real=True)
>>> eq1 =  a + b + c + d
>>> eq2 = a*b + b*c + c*d + d*a
>>> eq3 = a*b*c + b*c*d + c*d*a + d*a*b
>>> eq4 = a*b*c*d - 1
>>> system = [eq1, eq2, eq3, eq4]
>>> is_zero_dimensional(system)
False
>>> pprint(nonlinsolve(system, [a, b, c, d]), use_unicode=False)
  -1       1               1      -1
{(---, -d, -, {d} \ {0}), (-, -d, ---, {d} \ {0})}
   d       d               d       d
>>> nonlinsolve([(x+y)**2 - 4, x + y - 2], [x, y])
{(2 - y, y)}
```

2. If some of the equations are non-polynomial then $nonlinsolve$ will call the `substitution` function and return real and complex solutions, if present.

```
>>> from sympy import exp, sin
>>> nonlinsolve([exp(x) - sin(y), y**2 - 4], [x, y])
{(ImageSet(Lambda(_n, I*(2*_n*pi + pi) + log(sin(2))), Integers), -2),
 (ImageSet(Lambda(_n, 2*_n*I*pi + log(sin(2))), Integers), 2)}
```

3. If system is non-linear polynomial and zero-dimensional then it returns both solution (real and complex solutions, if present) using *solve_poly_system()* (page 856):

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

4. `nonlinsolve` can solve some linear (zero or positive dimensional) system (because it uses the *sympy.polys.polytools.groebner()* (page 2377) function to get the groebner basis and then uses the `substitution` function basis as the new $system$). But it is not recommended to solve linear system using `nonlinsolve`, because *linsolve()* (page 872) is better for general linear systems.

```
>>> nonlinsolve([x + 2*y -z - 3, x - y - 4*z + 9, y + z - 4], [x, y, z])
{(3*z - 5, 4 - z, z)}
```

5. System having polynomial equations and only real solution is solved using *solve_poly_system()* (page 856):