best:
To have *dsolve()* (page 755) try all methods and return the simplest one. This takes into account whether the solution is solvable in the function, whether it contains any Integral classes (i.e. unevaluatable integrals), and which one is the shortest in size.

See also the *classify_ode()* (page 760) docstring for more info on hints, and the *ode* (page 755) docstring for a list of all supported hints.

Tips

- You can declare the derivative of an unknown function this way:

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x # x is the independent variable
>>> f = Function("f")(x) # f is a function of x
>>> # f_ will be the derivative of f with respect to x
>>> f_ = Derivative(f, x)
```

- See `test_ode.py` for many tests, which serves also as a set of examples for how to use *dsolve()* (page 755).
- *dsolve()* (page 755) always returns an *Equality* (page 1023) class (except for the case when the hint is `all` or `all_Integral`). If possible, it solves the solution explicitly for the function being solved for. Otherwise, it returns an implicit solution.
- Arbitrary constants are symbols named `C1`, `C2`, and so on.
- Because all solutions should be mathematically equivalent, some hints may return the exact same result for an ODE. Often, though, two different hints will return the same solution formatted differently. The two should be equivalent. Also note that sometimes the values of the arbitrary constants in two different solutions may not be the same, because one constant may have "absorbed" other constants into it.
- Do `help(ode.ode_<hintname>)` to get help more information on a specific hint, where `<hintname>` is the name of a hint without `_Integral`.

For System Of Ordinary Differential Equations

Usage

`dsolve(eq, func)` -> Solve a system of ordinary differential equations `eq` for `func` being list of functions including $x(t)$, $y(t)$, $z(t)$ where number of functions in the list depends upon the number of equations provided in `eq`.

Details

`eq` can be any supported system of ordinary differential equations This can either be an *Equality* (page 1023), or an expression, which is assumed to be equal to `0`.

`func` holds `x(t)` and `y(t)` being functions of one variable which together with some of their derivatives make up the system of ordinary differential equation `eq`. It is not necessary to provide this; it will be autodetected (and an error raised if it could not be detected).

Hints

The hints are formed by parameters returned by classify_sysode, combining them give hints name used later for forming method name.

Examples

```
>>> from sympy import Function, dsolve, Eq, Derivative, sin, cos, symbols
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(Derivative(f(x), x, x) + 9*f(x), f(x))
Eq(f(x), C1*sin(3*x) + C2*cos(3*x))
```

```
>>> eq = sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x)
>>> dsolve(eq, hint='1st_exact')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> dsolve(eq, hint='almost_linear')
[Eq(f(x), -acos(C1/cos(x)) + 2*pi), Eq(f(x), acos(C1/cos(x)))]
>>> t = symbols('t')
>>> x, y = symbols('x, y', cls=Function)
>>> eq = (Eq(Derivative(x(t),t), 12*t*x(t) + 8*y(t)), Eq(Derivative(y(t),
↪t), 21*x(t) + 7*t*y(t)))
>>> dsolve(eq)
[Eq(x(t), C1*x0(t) + C2*x0(t)*Integral(8*exp(Integral(7*t,␣
↪t))*exp(Integral(12*t, t))/x0(t)**2, t)),
Eq(y(t), C1*y0(t) + C2*(y0(t)*Integral(8*exp(Integral(7*t,␣
↪t))*exp(Integral(12*t, t))/x0(t)**2, t) +
exp(Integral(7*t, t))*exp(Integral(12*t, t))/x0(t)))]
>>> eq = (Eq(Derivative(x(t),t),x(t)*y(t)*sin(t)), Eq(Derivative(y(t),t),
↪y(t)**2*sin(t)))
>>> dsolve(eq)
{Eq(x(t), -exp(C1)/(C2*exp(C1) - cos(t))), Eq(y(t), -1/(C1 - cos(t)))}
```


dsolve_system

sympy.solvers.ode.systems.**dsolve_system**(*eqs*, *funcs=None*, *t=None*, *ics=None*, *doit=False*, *simplify=True*)

Solves any(supported) system of Ordinary Differential Equations

Parameters

eqs : List

system of ODEs to be solved

funcs : List or None

List of dependent variables that make up the system of ODEs

t : Symbol or None

Independent variable in the system of ODEs

ics : Dict or None

Set of initial boundary/conditions for the system of ODEs

doit : Boolean

Evaluate the solutions if True. Default value is True. Can be set to false if the integral evaluation takes too much time and/or is not required.

simplify: Boolean

Simplify the solutions for the systems. Default value is True. Can be set to false if simplification takes too much time and/or is not required.

Returns

List of List of Equations

Raises

NotImplementedError

When the system of ODEs is not solvable by this function.

ValueError

When the parameters passed are not in the required form.

Explanation

This function takes a system of ODEs as an input, determines if the it is solvable by this function, and returns the solution if found any.

This function can handle: 1. Linear, First Order, Constant coefficient homogeneous system of ODEs 2. Linear, First Order, Constant coefficient non-homogeneous system of ODEs 3. Linear, First Order, non-constant coefficient homogeneous system of ODEs 4. Linear, First Order, non-constant coefficient non-homogeneous system of ODEs 5. Any implicit system which can be divided into system of ODEs which is of the above 4 forms 6. Any higher order linear system of ODEs that can be reduced to one of the 5 forms of systems described above.

The types of systems described above are not limited by the number of equations, i.e. this function can solve the above types irrespective of the number of equations in the system passed. But, the bigger the system, the more time it will take to solve the system.

This function returns a list of solutions. Each solution is a list of equations where LHS is the dependent variable and RHS is an expression in terms of the independent variable.

Among the non constant coefficient types, not all the systems are solvable by this function. Only those which have either a coefficient matrix with a commutative antiderivative or those systems which may be divided further so that the divided systems may have coefficient matrix with commutative antiderivative.

Examples

```
>>> from sympy import symbols, Eq, Function
>>> from sympy.solvers.ode.systems import dsolve_system
>>> f, g = symbols("f g", cls=Function)
>>> x = symbols("x")
```

```
>>> eqs = [Eq(f(x).diff(x), g(x)), Eq(g(x).diff(x), f(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]
```

You can also pass the initial conditions for the system of ODEs:

```
>>> dsolve_system(eqs, ics={f(0): 1, g(0): 0})
[[Eq(f(x), exp(x)/2 + exp(-x)/2), Eq(g(x), exp(x)/2 - exp(-x)/2)]]
```

Optionally, you can pass the dependent variables and the independent variable for which the system is to be solved:

```
>>> funcs = [f(x), g(x)]
>>> dsolve_system(eqs, funcs=funcs, t=x)
[[Eq(f(x), -C1*exp(-x) + C2*exp(x)), Eq(g(x), C1*exp(-x) + C2*exp(x))]]
```

Lets look at an implicit system of ODEs:

```
>>> eqs = [Eq(f(x).diff(x)**2, g(x)**2), Eq(g(x).diff(x), g(x))]
>>> dsolve_system(eqs)
[[Eq(f(x), C1 - C2*exp(x)), Eq(g(x), C2*exp(x))], [Eq(f(x), C1 +␣
↪C2*exp(x)), Eq(g(x), C2*exp(x))]]
```

classify_ode

sympy.solvers.ode.**classify_ode**(*eq*, *func=None*, *dict=False*, *ics=None*, ***, *prep=True*, *xi=None*, *eta=None*, *n=None*, ***kwargs*)

Returns a tuple of possible *dsolve()* (page 755) classifications for an ODE.

The tuple is ordered so that first item is the classification that *dsolve()* (page 755) uses to solve the ODE by default. In general, classifications at the near the beginning of the list will produce better solutions faster than those near the end, thought there are always exceptions. To make *dsolve()* (page 755) use a different classification, use `dsolve(ODE, func, hint=<classification>)`. See also the *dsolve()* (page 755) docstring for different meta-hints you can use.

If `dict` is true, *classify_ode()* (page 760) will return a dictionary of `hint:match` expression terms. This is intended for internal use by *dsolve()* (page 755). Note that because dictionaries are ordered arbitrarily, this will most likely not be in the same order as the tuple.

You can get help on different hints by executing `help(ode.ode_hintname)`, where `hintname` is the name of the hint without `_Integral`.

See *allhints* (page 767) or the *ode* (page 755) docstring for a list of all supported hints that can be returned from *classify_ode()* (page 760).

Notes

These are remarks on hint names.

`_Integral`

If a classification has `_Integral` at the end, it will return the expression with an unevaluated *Integral* (page 601) class in it. Note that a hint may do this anyway if *integrate()* (page 964) cannot do the integral, though just using an `_Integral` will do so much faster. Indeed, an `_Integral` hint will always be faster than its corresponding hint without `_Integral` because *integrate()* (page 964) is an expensive routine. If *dsolve()* (page 755) hangs, it is probably

because *integrate()* (page 964) is hanging on a tough or impossible integral. Try using an `_Integral` hint or `all_Integral` to get it return something.

Note that some hints do not have `_Integral` counterparts. This is because *integrate()* (page 598) is not used in solving the ODE for those method. For example, nth order linear homogeneous ODEs with constant coefficients do not require integration to solve, so there is no `nth_linear_homogeneous_constant_coeff_Integrate` hint. You can easily evaluate any unevaluated *Integral* (page 601)s in an expression by doing `expr.doit()`.

Ordinals

Some hints contain an ordinal such as `1st_linear`. This is to help differentiate them from other hints, as well as from other methods that may not be implemented yet. If a hint has `nth` in it, such as the `nth_linear` hints, this means that the method used to applies to ODEs of any order.

`indep` and `dep`

Some hints contain the words `indep` or `dep`. These reference the independent variable and the dependent function, respectively. For example, if an ODE is in terms of $f(x)$, then `indep` will refer to x and `dep` will refer to f.

`subs`

If a hints has the word `subs` in it, it means that the ODE is solved by substituting the expression given after the word `subs` for a single dummy variable. This is usually in terms of `indep` and `dep` as above. The substituted expression will be written only in characters allowed for names of Python objects, meaning operators will be spelled out. For example, `indep/dep` will be written as `indep_div_dep`.

`coeff`

The word `coeff` in a hint refers to the coefficients of something in the ODE, usually of the derivative terms. See the docstring for the individual methods for more info (`help(ode)`). This is contrast to `coefficients`, as in `undetermined_coefficients`, which refers to the common name of a method.

`_best`

Methods that have more than one fundamental way to solve will have a hint for each sub-method and a `_best` meta-classification. This will evaluate all hints and return the best, using the same considerations as the normal `best` meta-hint.

Examples

```
>>> from sympy import Function, classify_ode, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> classify_ode(Eq(f(x).diff(x), 0), f(x))
('nth_algebraic',
'separable',
'1st_exact',
'1st_linear',
```

(continues on next page)

(continued from previous page)

```
'Bernoulli',
'1st_homogeneous_coeff_best',
'1st_homogeneous_coeff_subs_indep_div_dep',
'1st_homogeneous_coeff_subs_dep_div_indep',
'1st_power_series', 'lie_group', 'nth_linear_constant_coeff_homogeneous',
'nth_linear_euler_eq_homogeneous',
'nth_algebraic_Integral', 'separable_Integral', '1st_exact_Integral',
'1st_linear_Integral', 'Bernoulli_Integral',
'1st_homogeneous_coeff_subs_indep_div_dep_Integral',
'1st_homogeneous_coeff_subs_dep_div_indep_Integral')
>>> classify_ode(f(x).diff(x, 2) + 3*f(x).diff(x) + 2*f(x) - 4)
('factorable', 'nth_linear_constant_coeff_undetermined_coefficients',
'nth_linear_constant_coeff_variation_of_parameters',
'nth_linear_constant_coeff_variation_of_parameters_Integral')
```

checkodesol

sympy.solvers.ode.**checkodesol**(*ode*, *sol*, *func=None*, *order='auto'*, *solve_for_func=True*)

Substitutes `sol` into `ode` and checks that the result is `0`.

This works when `func` is one function, like $f(x)$ or a list of functions like $[f(x), g(x)]$ when ode is a system of ODEs. `sol` can be a single solution or a list of solutions. Each solution may be an *Equality* (page 1023) that the solution satisfies, e.g. `Eq(f(x), C1), Eq(f(x) + C1, 0)`; or simply an *Expr* (page 947), e.g. `f(x) - C1`. In most cases it will not be necessary to explicitly identify the function, but if the function cannot be inferred from the original equation it can be supplied through the `func` argument.

If a sequence of solutions is passed, the same sort of container will be used to return the result for each solution.

It tries the following methods, in order, until it finds zero equivalence:

1. Substitute the solution for f in the original equation. This only works if `ode` is solved for f. It will attempt to solve it first unless `solve_for_func == False`.
2. Take n derivatives of the solution, where n is the order of `ode`, and check to see if that is equal to the solution. This only works on exact ODEs.
3. Take the 1st, 2nd, ..., nth derivatives of the solution, each time solving for the derivative of f of that order (this will always be possible because f is a linear operator). Then back substitute each derivative into `ode` in reverse order.

This function returns a tuple. The first item in the tuple is `True` if the substitution results in `0`, and `False` otherwise. The second item in the tuple is what the substitution results in. It should always be `0` if the first item is `True`. Sometimes this function will return `False` even when an expression is identically equal to `0`. This happens when *simplify()* (page 661) does not reduce the expression to `0`. If an expression returned by this function vanishes identically, then `sol` really is a solution to the `ode`.

If this function seems to hang, it is probably because of a hard simplification.

To use this function to test, test the first item of the tuple.

Examples

```
>>> from sympy import (Eq, Function, checkodesol, symbols,
...     Derivative, exp)
>>> x, C1, C2 = symbols('x,C1,C2')
>>> f, g = symbols('f g', cls=Function)
>>> checkodesol(f(x).diff(x), Eq(f(x), C1))
(True, 0)
>>> assert checkodesol(f(x).diff(x), C1)[0]
>>> assert not checkodesol(f(x).diff(x), x)[0]
>>> checkodesol(f(x).diff(x, 2), x**2)
(False, 2)
```

```
>>> eqs = [Eq(Derivative(f(x), x), f(x)), Eq(Derivative(g(x), x), g(x))]
>>> sol = [Eq(f(x), C1*exp(x)), Eq(g(x), C2*exp(x))]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

homogeneous_order

sympy.solvers.ode.**homogeneous_order**(*eq*, **symbols*)

Returns the order n if g is homogeneous and None if it is not homogeneous.

Determines if a function is homogeneous and if so of what order. A function $f(x, y, \cdots)$ is homogeneous of order n if $f(tx, ty, \cdots) = t^n f(x, y, \cdots)$.

If the function is of two variables, $F(x, y)$, then f being homogeneous of any order is equivalent to being able to rewrite $F(x, y)$ as $G(x/y)$ or $H(y/x)$. This fact is used to solve 1st order ordinary differential equations whose coefficients are homogeneous of the same order (see the docstrings of *HomogeneousCoeffSubsDepDivIndep* (page 773) and *HomogeneousCoeffSubsIndepDivDep* (page 774)).

Symbols can be functions, but every argument of the function must be a symbol, and the arguments of the function that appear in the expression must match those given in the list of symbols. If a declared function appears with different arguments than given in the list of symbols, None is returned.

Examples

```
>>> from sympy import Function, homogeneous_order, sqrt
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> homogeneous_order(f(x), f(x)) is None
True
>>> homogeneous_order(f(x,y), f(y, x), x, y) is None
True
>>> homogeneous_order(f(x), f(x), x)
1
>>> homogeneous_order(x**2*f(x)/sqrt(x**2+f(x)**2), x, f(x))
2
```

(continues on next page)

(continued from previous page)

```
>>> homogeneous_order(x**2+f(x), x, f(x)) is None
True
```

infinitesimals

sympy.solvers.ode.**infinitesimals**(*eq*, *func=None*, *order=None*, *hint='default'*, *match=None*)

The infinitesimal functions of an ordinary differential equation, $\xi(x,y)$ and $\eta(x,y)$, are the infinitesimals of the Lie group of point transformations for which the differential equation is invariant. So, the ODE $y' = f(x,y)$ would admit a Lie group $x^* = X(x,y;\varepsilon) = x + \varepsilon\xi(x,y)$, $y^* = Y(x,y;\varepsilon) = y + \varepsilon\eta(x,y)$ such that $(y^*)' = f(x^*,y^*)$. A change of coordinates, to $r(x,y)$ and $s(x,y)$, can be performed so this Lie group becomes the translation group, $r^* = r$ and $s^* = s + \varepsilon$. They are tangents to the coordinate curves of the new system.

Consider the transformation $(x,y) \to (X,Y)$ such that the differential equation remains invariant. ξ and η are the tangents to the transformed coordinates X and Y, at $\varepsilon = 0$.

$$\left(\frac{\partial X(x,y;\varepsilon)}{\partial\varepsilon}\right)|_{\varepsilon=0} = \xi, \left(\frac{\partial Y(x,y;\varepsilon)}{\partial\varepsilon}\right)|_{\varepsilon=0} = \eta,$$

The infinitesimals can be found by solving the following PDE:

```
>>> from sympy import Function, Eq, pprint
>>> from sympy.abc import x, y
>>> xi, eta, h = map(Function, ['xi', 'eta', 'h'])
>>> h = h(x, y)  # dy/dx = h
>>> eta = eta(x, y)
>>> xi = xi(x, y)
>>> genform = Eq(eta.diff(x) + (eta.diff(y) - xi.diff(x))*h
... - (xi.diff(y))*h**2 - xi*(h.diff(x)) - eta*(h.diff(y)), 0)
>>> pprint(genform)
/d              d          \                    d              2      ␣
↪ d
|--(eta(x, y)) - --(xi(x, y))|*h(x, y) - eta(x, y)*--(h(x, y)) - h (x,␣
↪y)*--(x
\dy             dx         /                    dy                    ␣
↪ dy

                   d             d
i(x, y)) - xi(x, y)*--(h(x, y)) + --(eta(x, y)) = 0
                   dx            dx
```

Solving the above mentioned PDE is not trivial, and can be solved only by making intelligent assumptions for ξ and η (heuristics). Once an infinitesimal is found, the attempt to find more heuristics stops. This is done to optimise the speed of solving the differential equation. If a list of all the infinitesimals is needed, `hint` should be flagged as `all`, which gives the complete list of infinitesimals. If the infinitesimals for a particular heuristic needs to be found, it can be passed as a flag to `hint`.

Examples

```
>>> from sympy import Function
>>> from sympy.solvers.ode.lie_group import infinitesimals
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x) - x**2*f(x)
>>> infinitesimals(eq)
[{eta(x, f(x)): exp(x**3/3), xi(x, f(x)): 0}]
```

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

checkinfsol

sympy.solvers.ode.**checkinfsol**(*eq*, *infinitesimals*, *func=None*, *order=None*)

This function is used to check if the given infinitesimals are the actual infinitesimals of the given first order differential equation. This method is specific to the Lie Group Solver of ODEs.

As of now, it simply checks, by substituting the infinitesimals in the partial differential equation.

$$\frac{\partial \eta}{\partial x} + \left(\frac{\partial \eta}{\partial y} - \frac{\partial \xi}{\partial x}\right) * h - \frac{\partial \xi}{\partial y} * h^2 - \xi \frac{\partial h}{\partial x} - \eta \frac{\partial h}{\partial y} = 0$$

where η, and ξ are the infinitesimals and $h(x, y) = \frac{dy}{dx}$

The infinitesimals should be given in the form of a list of dicts `[{xi(x, y): inf, eta(x, y): inf}]`, corresponding to the output of the function infinitesimals. It returns a list of values of the form `[(True/False, sol)]` where `sol` is the value obtained after substituting the infinitesimals in the PDE. If it is `True`, then `sol` would be 0.

constantsimp

sympy.solvers.ode.**constantsimp**(*expr*, *constants*)

Simplifies an expression with arbitrary constants in it.

This function is written specifically to work with *dsolve()* (page 755), and is not intended for general use.

Simplification is done by "absorbing" the arbitrary constants into other arbitrary constants, numbers, and symbols that they are not independent of.

The symbols must all have the same name with numbers after it, for example, `C1`, `C2`, `C3`. The `symbolname` here would be '`C`', the `startnumber` would be 1, and the `endnumber` would be 3. If the arbitrary constants are independent of the variable `x`, then the independent symbol would be `x`. There is no need to specify the dependent function, such as `f(x)`, because it already has the independent symbol, `x`, in it.

Because terms are "absorbed" into arbitrary constants and because constants are renumbered after simplifying, the arbitrary constants in expr are not necessarily equal to the ones of the same name in the returned result.

If two or more arbitrary constants are added, multiplied, or raised to the power of each other, they are first absorbed together into a single arbitrary constant. Then the new constant is combined into other terms if necessary.

Absorption of constants is done with limited assistance:

1. terms of *Add* (page 1013)s are collected to try join constants so $e^x(C_1 \cos(x) + C_2 \cos(x))$ will simplify to $e^x C_1 \cos(x)$;
2. powers with exponents that are *Add* (page 1013)s are expanded so e^{C_1+x} will be simplified to $C_1 e^x$.

Use *constant_renumber()* (page 768) to renumber constants after simplification or else arbitrary numbers on constants may appear, e.g. $C_1 + C_3 x$.

In rare cases, a single constant can be "simplified" into two constants. Every differential equation solution should have as many arbitrary constants as the order of the differential equation. The result here will be technically correct, but it may, for example, have C_1 and C_2 in an expression, when C_1 is actually equal to C_2. Use your discretion in such situations, and also take advantage of the ability to use hints in *dsolve()* (page 755).

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode.ode import constantsimp
>>> C1, C2, C3, x, y = symbols('C1, C2, C3, x, y')
>>> constantsimp(2*C1*x, {C1, C2, C3})
C1*x
>>> constantsimp(C1 + 2 + x, {C1, C2, C3})
C1 + x
>>> constantsimp(C1*C2 + 2 + C2 + C3*x, {C1, C2, C3})
C1 + C3*x
```

Hint Functions

These functions are intended for internal use by *dsolve()* (page 755) and others. Unlike *User Functions* (page 755), above, these are not intended for every-day use by ordinary SymPy users. Instead, functions such as *dsolve()* (page 755) should be used. Nonetheless, these functions contain useful information in their docstrings on the various ODE solving methods. For this reason, they are documented here.

allhints

sympy.solvers.ode.**allhints = ('factorable', 'nth_algebraic', 'separable', 'exact', '1st_linear', 'Bernoulli', '1st_rational_riccati', 'Riccati_special_minus2', '1st_homogeneous_coeff_best', '1st_homogeneous_coeff_subs_indep_div_dep', '1st_homogeneous_coeff_subs_dep_div_indep', 'almost_linear', 'linear_coefficients', 'separable_reduced', '1st_power_series', 'lie_group', 'nth_linear_constant_coeff_homogeneous', 'nth_linear_euler_eq_homogeneous', 'nth_linear_constant_coeff_undetermined_coefficients', 'nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients', 'nth_linear_constant_coeff_variation_of_parameters', 'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters', 'Liouville', '2nd_linear_airy', '2nd_linear_bessel', '2nd_hypergeometric', '2nd_hypergeometric_Integral', 'nth_order_reducible', '2nd_power_series_ordinary', '2nd_power_series_regular', 'nth_algebraic_Integral', 'separable_Integral', '1st_exact_Integral', '1st_linear_Integral', 'Bernoulli_Integral', '1st_homogeneous_coeff_subs_indep_div_dep_Integral', '1st_homogeneous_coeff_subs_dep_div_indep_Integral', 'almost_linear_Integral', 'linear_coefficients_Integral', 'separable_reduced_Integral', 'nth_linear_constant_coeff_variation_of_parameters_Integral', 'nth_linear_euler_eq_nonhomogeneous_variation_of_parameters_Integral', 'Liouville_Integral', '2nd_nonlinear_autonomous_conserved', '2nd_nonlinear_autonomous_conserved_Integral')**

Built-in immutable sequence.

If no argument is given, the constructor returns an empty tuple. If iterable is specified the tuple is initialized from iterable's items.

If the argument is a tuple, the return value is the same object.

odesimp

sympy.solvers.ode.ode.**odesimp**(*ode*, *eq*, *func*, *hint*)

Simplifies solutions of ODEs, including trying to solve for `func` and running *constantsimp()* (page 765).

It may use knowledge of the type of solution that the hint returns to apply additional simplifications.

It also attempts to integrate any *Integral* (page 601)s in the expression, if the hint is not an `_Integral` hint.

This function should have no effect on expressions returned by *dsolve()* (page 755), as *dsolve()* (page 755) already calls *odesimp()* (page 767), but the individual hint functions do not call *odesimp()* (page 767) (because the *dsolve()* (page 755) wrapper does). Therefore, this function is designed for mainly internal use.

Examples

```
>>> from sympy import sin, symbols, dsolve, pprint, Function
>>> from sympy.solvers.ode.ode import odesimp
>>> x, u2, C1= symbols('x,u2,C1')
>>> f = Function('f')
```

```
>>> eq = dsolve(x*f(x).diff(x) - f(x) - x*sin(f(x)/x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral',
... simplify=False)
>>> pprint(eq, wrap_line=False)
                        x
                       ----
                       f(x)
                         /
                        |
                        |   /        1   \
                        |  -|u1 + -------|
                        |   |        /1 \|
                        |   |     sin|--||
                        |   \        \u1//
log(f(x)) = log(C1) +   |  ---------------- d(u1)
                        |           2
                        |         u1
                        |
                       /
```

```
>>> pprint(odesimp(eq, f(x), 1, {C1},
... hint='1st_homogeneous_coeff_subs_indep_div_dep'
... ))
    x
--------- = C1
   /f(x)\
tan|----|
   \2*x /
```

constant_renumber

sympy.solvers.ode.ode.**constant_renumber**(*expr*, *variables=None*, *newconstants=None*)

Renumber arbitrary constants in `expr` to use the symbol names as given in `newconstants`. In the process, this reorders expression terms in a standard way.

If `newconstants` is not provided then the new constant names will be `C1`, `C2` etc. Otherwise `newconstants` should be an iterable giving the new symbols to use for the constants in order.

The `variables` argument is a list of non-constant symbols. All other free symbols found in `expr` are assumed to be constants and will be renumbered. If `variables` is not given then any numbered symbol beginning with `C` (e.g. `C1`) is assumed to be a constant.

Symbols are renumbered based on `.sort_key()`, so they should be numbered roughly in the order that they appear in the final, printed expression. Note that this ordering is based in part on hashes, so it can produce different results on different machines.

The structure of this function is very similar to that of *constantsimp()* (page 765).

Examples

```
>>> from sympy import symbols
>>> from sympy.solvers.ode.ode import constant_renumber
>>> x, C1, C2, C3 = symbols('x,C1:4')
>>> expr = C3 + C2*x + C1*x**2
>>> expr
C1*x**2  + C2*x + C3
>>> constant_renumber(expr)
C1 + C2*x + C3*x**2
```

The `variables` argument specifies which are constants so that the other symbols will not be renumbered:

```
>>> constant_renumber(expr, [C1, x])
C1*x**2  + C2 + C3*x
```

The `newconstants` argument is used to specify what symbols to use when replacing the constants:

```
>>> constant_renumber(expr, [x], newconstants=symbols('E1:4'))
E1 + E2*x + E3*x**2
```


sol_simplicity

sympy.solvers.ode.ode.**ode_sol_simplicity**(*sol*, *func*, *trysolving=True*)

Returns an extended integer representing how simple a solution to an ODE is.

The following things are considered, in order from most simple to least:

- `sol` is solved for `func`.
- `sol` is not solved for `func`, but can be if passed to solve (e.g., a solution returned by `dsolve(ode, func, simplify=False)`.
- If `sol` is not solved for `func`, then base the result on the length of `sol`, as computed by `len(str(sol))`.
- If `sol` has any unevaluated *Integral* (page 601)s, this will automatically be considered less simple than any of the above.

This function returns an integer such that if solution A is simpler than solution B by above metric, then `ode_sol_simplicity(sola, func) < ode_sol_simplicity(solb, func)`.

Currently, the following are the numbers returned, but if the heuristic is ever improved, this may change. Only the ordering is guaranteed.

Simplicity	Return
`sol` solved for `func`	`-2`
`sol` not solved for `func` but can be	`-1`
`sol` is not solved nor solvable for `func`	`len(str(sol))`
`sol` contains an *Integral* (page 601)	`oo`

oo here means the SymPy infinity, which should compare greater than any integer.

If you already know *solve()* (page 836) cannot solve sol, you can use trysolving=False to skip that step, which is the only potentially slow step. For example, *dsolve()* (page 755) with the simplify=False flag should do this.

If sol is a list of solutions, if the worst solution in the list returns oo it returns that, otherwise it returns len(str(sol)), that is, the length of the string representation of the whole list.

Examples

This function is designed to be passed to min as the key argument, such as min(listofsolutions, key=lambda i: ode_sol_simplicity(i, f(x))).

```
>>> from sympy import symbols, Function, Eq, tan, Integral
>>> from sympy.solvers.ode.ode import ode_sol_simplicity
>>> x, C1, C2 = symbols('x, C1, C2')
>>> f = Function('f')
```

```
>>> ode_sol_simplicity(Eq(f(x), C1*x**2), f(x))
-2
>>> ode_sol_simplicity(Eq(x**2 + f(x), C1), f(x))
-1
>>> ode_sol_simplicity(Eq(f(x), C1*Integral(2*x, x)), f(x))
oo
>>> eq1 = Eq(f(x)/tan(f(x)/(2*x)), C1)
>>> eq2 = Eq(f(x)/tan(f(x)/(2*x) + f(x)), C2)
>>> [ode_sol_simplicity(eq, f(x)) for eq in [eq1, eq2]]
[28, 35]
>>> min([eq1, eq2], key=lambda i: ode_sol_simplicity(i, f(x)))
Eq(f(x)/tan(f(x)/(2*x)), C1)
```

factorable

class sympy.solvers.ode.single.**Factorable**(*ode_problem*)

Solves equations having a solvable factor.

This function is used to solve the equation having factors. Factors may be of type algebraic or ode. It will try to solve each factor independently. Factors will be solved by calling dsolve. We will return the list of solutions.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (f(x)**2-4)*(f(x).diff(x)+f(x))
>>> pprint(dsolve(eq, f(x)))
                                  -x
[f(x) = 2, f(x) = -2, f(x) = C1*e  ]
```

1st_exact

class sympy.solvers.ode.single.**FirstExact**(*ode_problem*)

Solves 1st order exact ordinary differential equations.

A 1st order differential equation is called exact if it is the total differential of a function. That is, the differential equation

$$P(x, y)\,\partial x + Q(x, y)\,\partial y = 0$$

is exact if there is some function $F(x, y)$ such that $P(x, y) = \partial F/\partial x$ and $Q(x, y) = \partial F/\partial y$. It can be shown that a necessary and sufficient condition for a first order ODE to be exact is that $\partial P/\partial y = \partial Q/\partial x$. Then, the solution will be as given below:

```
>>> from sympy import Function, Eq, Integral, symbols, pprint
>>> x, y, t, x0, y0, C1= symbols('x,y,t,x0,y0,C1')
>>> P, Q, F= map(Function, ['P', 'Q', 'F'])
>>> pprint(Eq(Eq(F(x, y), Integral(P(t, y), (t, x0, x)) +
... Integral(Q(x0, t), (t, y0, y))), C1))
            x                 y
            /                 /
           |                 |
F(x, y) =  |  P(t, y) dt +   |  Q(x0, t) dt = C1
           |                 |
          /                 /
          x0                y0
```

Where the first partials of P and Q exist and are continuous in a simply connected region.

A note: SymPy currently has no way to represent inert substitution on an expression, so the hint `1st_exact_Integral` will return an integral with dy. This is supposed to represent the function that you are solving for.

Examples

```
>>> from sympy import Function, dsolve, cos, sin
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(cos(f(x)) - (x*sin(f(x)) - f(x)**2)*f(x).diff(x),
... f(x), hint='1st_exact')
Eq(x*cos(f(x)) + f(x)**3/3, C1)
```

References

- https://en.wikipedia.org/wiki/Exact_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 73

indirect doctest

1st_homogeneous_coeff_best

class sympy.solvers.ode.single.**HomogeneousCoeffBest**(*ode_problem*)

Returns the best solution to an ODE from the two hints `1st_homogeneous_coeff_subs_dep_div_indep` and `1st_homogeneous_coeff_subs_indep_div_dep`.

This is as determined by *ode_sol_simplicity()* (page 769).

See the *HomogeneousCoeffSubsIndepDivDep* (page 774) and *HomogeneousCoeffSubsDepDivIndep* (page 773) docstrings for more information on these hints. Note that there is no `ode_1st_homogeneous_coeff_best_Integral` hint.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_best', simplify=False))
                        /     2    \
                        | 3*x     |
                    log|----- + 1|
                        | 2       |
                        \f (x)    /
log(f(x)) = log(C1) - --------------
                             3
```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

1st_homogeneous_coeff_subs_dep_div_indep

class sympy.solvers.ode.single.**HomogeneousCoeffSubsDepDivIndep**(*ode_problem*)

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_1 = \frac{\text{<dependent variable>}}{\text{<independent variable>}}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of *homogeneous_order()* (page 763).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $y = u_1 x$ (i.e. $u_1 = y/x$) will turn the differential equation into an equation separable in the variables x and u. If $h(u_1)$ is the function that results from making the substitution $u_1 = f(x)/x$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(f(x)/x) + h(f(x)/x)*f(x).diff(x)
>>> pprint(genform)
 /f(x)\    /f(x)\ d
g|----| + h|----|*--(f(x))
 \ x  /    \ x  / dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep_Integral'))
               f(x)
               ----
                x
                 /
                |
                |       -h(u1)
log(x) = C1 +   |  ---------------- d(u1)
                |  u1*h(u1) + g(u1)
                |
               /
```

Where $u_1 h(u_1) + g(u_1) \neq 0$ and $x \neq 0$.

See also the docstrings of *HomogeneousCoeffBest* (page 772) and *HomogeneousCoeffSubsIndepDivDep* (page 774).

Examples

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_dep_div_indep', simplify=False))
                      /          3   \
                      |3*f(x)   f (x)|
                   log|------ + -----|
                      |  x         3 |
                      \           x  /
log(x) = log(C1) - -------------------
                             3
```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

1st_homogeneous_coeff_subs_indep_div_dep

class sympy.solvers.ode.single.**HomogeneousCoeffSubsIndepDivDep**(*ode_problem*)

Solves a 1st order differential equation with homogeneous coefficients using the substitution $u_2 = \frac{\text{<independent variable>}}{\text{<dependent variable>}}$.

This is a differential equation

$$P(x, y) + Q(x, y)dy/dx = 0$$

such that P and Q are homogeneous and of the same order. A function $F(x, y)$ is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, $F(x, y)$ can be rewritten as $G(y/x)$ or $H(x/y)$. See also the docstring of *homogeneous_order()* (page 763).

If the coefficients P and Q in the differential equation above are homogeneous functions of the same order, then it can be shown that the substitution $x = u_2 y$ (i.e. $u_2 = x/y$) will turn the differential equation into an equation separable in the variables y and u_2. If $h(u_2)$ is the function that results from making the substitution $u_2 = x/f(x)$ on $P(x, f(x))$ and $g(u_2)$ is the function that results from the substitution on $Q(x, f(x))$ in the differential equation $P(x, f(x)) + Q(x, f(x))f'(x) = 0$, then the general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = g(x/f(x)) + h(x/f(x))*f(x).diff(x)
>>> pprint(genform)
 / x  \    / x  \ d
g|----| + h|----|*--(f(x))
```

(continues on next page)

(continued from previous page)

```
 \f(x)/    \f(x)/ dx
>>> pprint(dsolve(genform, f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep_Integral'))
             x
            ----
            f(x)
              /
             |
             |       -g(u1)
             |  ---------------- d(u1)
             |  u1*g(u1) + h(u1)
             |
            /

f(x) = C1*e
```

Where $u_1 g(u_1) + h(u_1) \neq 0$ and $f(x) \neq 0$.

See also the docstrings of *HomogeneousCoeffBest* (page 772) and *HomogeneousCoeffSubsDepDivIndep* (page 773).

Examples

```
>>> from sympy import Function, pprint, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(2*x*f(x) + (x**2 + f(x)**2)*f(x).diff(x), f(x),
... hint='1st_homogeneous_coeff_subs_indep_div_dep',
... simplify=False))
                          /    2    \
                          | 3*x     |
                       log|----- + 1|
                          | 2       |
                          \f (x)    /
log(f(x)) = log(C1) - --------------
                             3
```

References

- https://en.wikipedia.org/wiki/Homogeneous_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 59

indirect doctest

1st_linear

class sympy.solvers.ode.single.**FirstLinear**(*ode_problem*)

Solves 1st order linear differential equations.

These are differential equations of the form

$$dy/dx + P(x)y = Q(x).$$

These kinds of differential equations can be solved in a general way. The integrating factor $e^{\int P(x)\,dx}$ will turn the equation into a separable equation. The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff, sin
>>> from sympy.abc import x
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x))
>>> pprint(genform)
            d
P(x)*f(x) + --(f(x)) = Q(x)
            dx
>>> pprint(dsolve(genform, f(x), hint='1st_linear_Integral'))
        /       /                    \
        |       |                    |
        |       |         /          |     /
        |       |        |           |    |
        |       |        | P(x) dx   |  - | P(x) dx
        |       |        |           |    |
        |       |       /            |   /
f(x) = |C1 +   | Q(x)*e               dx|*e
        |       |                    |
        \     /                      /
```

Examples

```
>>> f = Function('f')
>>> pprint(dsolve(Eq(x*diff(f(x), x) - f(x), x**2*sin(x)),
... f(x), '1st_linear'))
f(x) = x*(C1 - cos(x))
```

References

- https://en.wikipedia.org/wiki/Linear_differential_equation#First_order_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 92

indirect doctest

1st_rational_riccati

class sympy.solvers.ode.single.**RationalRiccati**(*ode_problem*)

Gives general solutions to the first order Riccati differential equations that have atleast one rational particular solution.

$$y' = b_0(x) + b_1(x)y + b_2(x)y^2$$

where b_0, b_1 and b_2 are rational functions of x with $b_2 \neq 0$ ($b_2 = 0$ would make it a Bernoulli equation).

Examples

```
>>> from sympy import Symbol, Function, dsolve, checkodesol
>>> f = Function('f')
>>> x = Symbol('x')
```

```
>>> eq = -x**4*f(x)**2 + x**3*f(x).diff(x) + x**2*f(x) + 20
>>> sol = dsolve(eq, hint="1st_rational_riccati")
>>> sol
Eq(f(x), (4*C1 - 5*x**9 - 4)/(x**2*(C1 + x**9 - 1)))
>>> checkodesol(eq, sol)
(True, 0)
```

References

- Riccati ODE: https://en.wikipedia.org/wiki/Riccati_equation
- N. Thieu Vo - Rational and Algebraic Solutions of First-Order Algebraic ODEs: Algorithm 11, pp. 78 - https://www3.risc.jku.at/publications/download/risc_5387/PhDThesisThieu.pdf

2nd_linear_airy

class sympy.solvers.ode.single.**SecondLinearAiry**(*ode_problem*)

Gives solution of the Airy differential equation

$$\frac{d^2y}{dx^2} + (a + bx)y(x) = 0$$

in terms of Airy special functions airyai and airybi.

Examples

```
>>> from sympy import dsolve, Function
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) - x*f(x)
>>> dsolve(eq)
Eq(f(x), C1*airyai(x) + C2*airybi(x))
```

2nd_linear_bessel

class sympy.solvers.ode.single.**SecondLinearBessel**(*ode_problem*)

Gives solution of the Bessel differential equation

$$x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} y(x) + (x^2 - n^2) y(x)$$

if n is integer then the solution is of the form `Eq(f(x), C0 besselj(n,x) + C1 bessely(n,x))` as both the solutions are linearly independent else if n is a fraction then the solution is of the form `Eq(f(x), C0 besselj(n,x) + C1 besselj(-n,x))` which can also transform into `Eq(f(x), C0 besselj(n,x) + C1 bessely(n,x))`.

Examples

```
>>> from sympy.abc import x
>>> from sympy import Symbol
>>> v = Symbol('v', positive=True)
>>> from sympy import dsolve, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = x**2*y.diff(x, 2) + x*y.diff(x) + (x**2 - v**2)*y
>>> dsolve(genform)
Eq(f(x), C1*besselj(v, x) + C2*bessely(v, x))
```

References

https://www.math24.net/bessel-differential-equation/

Bernoulli

class sympy.solvers.ode.single.**Bernoulli**(*ode_problem*)

Solves Bernoulli differential equations.

These are equations of the form

$$dy/dx + P(x)y = Q(x)y^n\text{, } n \neq 1\text{`}.$$

The substitution $w = 1/y^{1-n}$ will transform an equation of this form into one that is linear (see the docstring of *FirstLinear* (page 776)). The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x, n
>>> f, P, Q = map(Function, ['f', 'P', 'Q'])
>>> genform = Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)**n)
>>> pprint(genform)
          d                 n
P(x)*f(x) + --(f(x)) = Q(x)*f (x)
          dx
>>> pprint(dsolve(genform, f(x), hint='Bernoulli_Integral'), num_
↪columns=110)
                                                                                                ␣
↪                                 -1                                                            ␣
                                                                                                ␣
↪                                -----                                                          ␣
                                                                                                ␣
↪                                n - 1                                                         ␣
      //          /                                      /                                      ␣
↪        \                    \                                                              ␣
      ||         |                                     |                                       ␣
↪        |                    |                                                              ␣
      ||         |                  /                  |                   /                  ␣
↪        |            /       |                                                              ␣
      ||         |                    |                |                   |                  ␣
↪        |          |         |                                                              ␣
      ||         |       -(n - 1)* | P(x) dx          |        -(n - 1)* |␣
↪P(x) dx   |  (n - 1)* | P(x) dx|
      ||         |                    |                |                   |                  ␣
↪        |          |         |                                                              ␣
      ||         |                   /                 |                 /                  ␣
↪        |         /          |                                                              ␣
f(x) = ||C1 - n* | Q(x)*e                      dx +   | Q(x)*e                                    ␣
↪      dx|*e                |                                                              ␣
      ||         |                                     |                                       ␣
↪        |                    |                                                              ␣
      \\       /                                     /                                       ␣
↪        /                   /                                                              ␣
```

Note that the equation is separable when $n = 1$ (see the docstring of *Separable* (page 789)).

```
>>> pprint(dsolve(Eq(f(x).diff(x) + P(x)*f(x), Q(x)*f(x)), f(x),
... hint='separable_Integral'))
f(x)
    /
|                /
|  1             |
|  - dy = C1 +   | (-P(x) + Q(x)) dx
|  y             |
|              /
/
```

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint, log
>>> from sympy.abc import x
>>> f = Function('f')
```

```
>>> pprint(dsolve(Eq(x*f(x).diff(x) + f(x), log(x)*f(x)**2),
... f(x), hint='Bernoulli'))
                1
f(x) =  -----------------
        C1*x + log(x) + 1
```

References

- https://en.wikipedia.org/wiki/Bernoulli_differential_equation
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 95

indirect doctest

Liouville

class sympy.solvers.ode.single.**Liouville**(*ode_problem*)

Solves 2nd order Liouville differential equations.

The general form of a Liouville ODE is

$$\frac{d^2y}{dx^2} + g(y)\left(\frac{dy}{dx}\right)^2 + h(x)\frac{dy}{dx}.$$

The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint, diff
>>> from sympy.abc import x
>>> f, g, h = map(Function, ['f', 'g', 'h'])
>>> genform = Eq(diff(f(x),x,x) + g(f(x))*diff(f(x),x)**2 +
... h(x)*diff(f(x),x), 0)
>>> pprint(genform)
                    2                     2
        /d       \          d           d
g(f(x))*|--(f(x))|  + h(x)*--(f(x)) + ---(f(x)) = 0
        \dx      /          dx            2
                                        dx
>>> pprint(dsolve(genform, f(x), hint='Liouville_Integral'))
                                  f(x)
          /                     /
         |                     |
         |     /               |     /
         |    |                |    |
         |  - | h(x) dx        |    | g(y) dy
         |    |                |    |
```

(continues on next page)

(continued from previous page)

```
          |   /                 |   /
C1 + C2* | e           dx +    |  e          dy = 0
         |                     |
        /                     /
```

Examples

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(diff(f(x), x, x) + diff(f(x), x)**2/f(x) +
... diff(f(x), x)/x, f(x), hint='Liouville'))

                  ________________          ________________
[f(x) = -\/ C1 + C2*log(x) , f(x) = \/ C1 + C2*log(x) ]
```

References

- Goldstein and Braun, “Advanced Methods for the Solution of Differential Equations”, pp. 98
- http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Liouville

indirect doctest

Riccati_special_minus2

class sympy.solvers.ode.single.**RiccatiSpecial**(*ode_problem*)

The general Riccati equation has the form

$$dy/dx = f(x)y^2 + g(x)y + h(x).$$

While it does not have a general solution [1], the “special” form, $dy/dx = ay^2 - bx^c$, does have solutions in many cases [2]. This routine returns a solution for $a(dy/dx) = by^2 + cy/x + d/x^2$ that is obtained by using a suitable change of variables to reduce it to the special form and is valid when neither a nor b are zero and either c or d is zero.

```
>>> from sympy.abc import x, a, b, c, d
>>> from sympy import dsolve, checkodesol, pprint, Function
>>> f = Function('f')
>>> y = f(x)
>>> genform = a*y.diff(x) - (b*y**2 + c*y/x + d/x**2)
>>> sol = dsolve(genform, y, hint="Riccati_special_minus2")
>>> pprint(sol, wrap_line=False)
       /                                  /        ____________________    ␣
↪   \\
       |          ____________________    |       /                2    ␣
↪   ||
```

(continues on next page)

(continued from previous page)

```
       |          /                2     |     \/  4*b*d - (a + c) ␣
↪*log(x)||
      -|a + c - \/  4*b*d - (a + c)  *tan|C1 + ------------------------
↪---||
       \                                 \                 2*a        ␣
↪   //
f(x) = ------------------------------------------------------------------
↪------
                                    2*b*x
```

```
>>> checkodesol(genform, sol, order=1)[0]
True
```

References

- http://www.maplesoft.com/support/help/Maple/view.aspx?path=odeadvisor/Riccati
- http://eqworld.ipmnet.ru/en/solutions/ode/ode0106.pdf - http://eqworld.ipmnet.ru/en/solutions/ode/ode0123.pdf

nth_linear_constant_coeff_homogeneous

class sympy.solvers.ode.single.**NthLinearConstantCoeffHomogeneous**(*ode_problem*)

Solves an nth order linear homogeneous differential equation with constant coefficients.

This is an equation of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$$

These equations can be solved in a general manner, by taking the roots of the characteristic equation $a_n m^n + a_{n-1} m^{n-1} + \cdots + a_1 m + a_0 = 0$. The solution will then be the sum of $C_n x^i e^{rx}$ terms, for each where C_n is an arbitrary constant, r is a root of the characteristic equation and i is one of each from 0 to the multiplicity of the root - 1 (for example, a root 3 of multiplicity 2 would create the terms $C_1 e^{3x} + C_2 x e^{3x}$). The exponential is usually expanded for complex roots using Euler's equation $e^{Ix} = \cos(x) + I\sin(x)$. Complex roots always come in conjugate pairs in polynomials with real coefficients, so the two roots will be represented (after simplifying the constants) as $e^{ax}\left(C_1 \cos(bx) + C_2 \sin(bx)\right)$.

If SymPy cannot find exact roots to the characteristic equation, a *ComplexRootOf* (page 2431) instance will be return instead.

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(f(x).diff(x, 5) + 10*f(x).diff(x) - 2*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous')
...
Eq(f(x), C5*exp(x*CRootOf(_x**5 + 10*_x - 2, 0))
+ (C1*sin(x*im(CRootOf(_x**5 + 10*_x - 2, 1)))
+ C2*cos(x*im(CRootOf(_x**5 + 10*_x - 2, 1))))*exp(x*re(CRootOf(_x**5 +␣
```

(continues on next page)

(continued from previous page)

```
↪10*_x - 2, 1)))
+ (C3*sin(x*im(CRootOf(_x**5 + 10*_x - 2, 3)))
+ C4*cos(x*im(CRootOf(_x**5 + 10*_x - 2, 3))))*exp(x*re(CRootOf(_x**5 +␣
↪10*_x - 2, 3))))
```

Note that because this method does not involve integration, there is no `nth_linear_constant_coeff_homogeneous_Integral` hint.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 4) + 2*f(x).diff(x, 3) -
... 2*f(x).diff(x, 2) - 6*f(x).diff(x) + 5*f(x), f(x),
... hint='nth_linear_constant_coeff_homogeneous'))
                    x                                  -2*x
f(x) = (C1 + C2*x)*e  + (C3*sin(x) + C4*cos(x))*e
```

References

- https://en.wikipedia.org/wiki/Linear_differential_equation section: Nonhomogeneous_equation_with_constant_coefficients
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 211

indirect doctest

nth_linear_constant_coeff_undetermined_coefficients

class sympy.solvers.ode.single.**NthLinearConstantCoeffUndeterminedCoefficients**(*ode_problem*)

Solves an nth order linear differential equation with constant coefficients using the method of undetermined coefficients.

This method works on differential equations of the form

$$a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x),$$

where $P(x)$ is a function that has a finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $ax^i e^{bx} \sin(cx + d)$ or $ax^i e^{bx} \cos(cx + d)$, where i is a non-negative integer and a, b, c, and d are constants. For example any polynomial in x, functions like $x^2 e^{2x}$, $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert $\sin^2(x)$ into $(1 + \cos(2x))/2$ to properly apply the method of undetermined coefficients on it.

This method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for

each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient x to make them linearly independent.

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 2) + 2*f(x).diff(x) + f(x) -
... 4*exp(-x)*x**2 + cos(2*x), f(x),
... hint='nth_linear_constant_coeff_undetermined_coefficients'))
       /       /      3\\
       |       |     x ||  -x   4*sin(2*x)   3*cos(2*x)
f(x) = |C1 + x*|C2 + --||*e   - ---------- + ----------
       \       \     3 //          25           25
```

References

- https://en.wikipedia.org/wiki/Method_of_undetermined_coefficients
- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 221

indirect doctest

nth_linear_constant_coeff_variation_of_parameters

class sympy.solvers.ode.single.**NthLinearConstantCoeffVariationOfParameters**(*ode_problem*)

Solves an nth order linear differential equation with constant coefficients using the method of variation of parameters.

This method works on any differential equations of the form

$$f^{(n)}(x) + a_{n-1}f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = P(x).$$

This method works by assuming that the particular solution takes the form

$$\sum_{x=1}^{n} c_i(x) y_i(x),$$

where y_i is the ith solution to the homogeneous equation. The solution is then solved using Wronskian’s and Cramer’s Rule. The particular solution is given by

$$\sum_{x=1}^{n} \left(\int \frac{W_i(x)}{W(x)}\, dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the ith column replaced with $[0, 0, \cdots, 0, P(x)]$.

This method is general enough to solve any nth order inhomogeneous linear differential equation with constant coefficients, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it does not use integration, making it faster and more reliable.

Warning, using simplify=False with 'nth_linear_constant_coeff_variation_of_parameters' in *dsolve()* (page 755) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use simplify=False with 'nth_linear_constant_coeff_variation_of_parameters_Integral' for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

Examples

```
>>> from sympy import Function, dsolve, pprint, exp, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x, 3) - 3*f(x).diff(x, 2) +
... 3*f(x).diff(x) - f(x) - exp(x)*log(x), f(x),
... hint='nth_linear_constant_coeff_variation_of_parameters'))
       /       /       /     x*log(x)   11*x\\\  x
f(x) = |C1 + x*|C2 + x*|C3 + -------- - ----|||*e
       \       \       \        6         36 ///
```

References

- https://en.wikipedia.org/wiki/Variation_of_parameters
- http://planetmath.org/VariationOfParameters
- M. Tenenbaum & H. Pollard, "Ordinary Differential Equations", Dover 1963, pp. 233

indirect doctest

nth_linear_euler_eq_homogeneous

class sympy.solvers.ode.single.**NthLinearEulerEqHomogeneous**(*ode_problem*)

Solves an nth order linear homogeneous variable-coefficient Cauchy-Euler equidimensional ordinary differential equation.

This is an equation with form $0 = a_0 f(x) + a_1 x f'(x) + a_2 x^2 f''(x) \cdots$.

These equations can be solved in a general manner, by substituting solutions of the form $f(x) = x^r$, and deriving a characteristic equation for r. When there are repeated roots, we include extra terms of the form $C_{rk} \ln^k(x) x^r$, where C_{rk} is an arbitrary integration constant, r is a root of the characteristic equation, and k ranges over the multiplicity of r. In the cases where the roots are complex, solutions of the form $C_1 x^a \sin(b \log(x)) + C_2 x^a \cos(b \log(x))$ are returned, based on expansions with Euler's formula. The general solution is the sum of the terms found. If SymPy cannot find exact

roots to the characteristic equation, a *ComplexRootOf* (page 2431) instance will be returned instead.

```
>>> from sympy import Function, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> dsolve(4*x**2*f(x).diff(x, 2) + f(x), f(x),
... hint='nth_linear_euler_eq_homogeneous')
...
Eq(f(x), sqrt(x)*(C1 + C2*log(x)))
```

Note that because this method does not involve integration, there is no `nth_linear_euler_eq_homogeneous_Integral` hint.

The following is for internal use:

- `returns = 'sol'` returns the solution to the ODE.
- `returns = 'list'` returns a list of linearly independent solutions, corresponding to the fundamental solution set, for use with non homogeneous solution methods like variation of parameters and undetermined coefficients. Note that, though the solutions should be linearly independent, this function does not explicitly check that. You can do `assert simplify(wronskian(sollist)) != 0` to check for linear independence. Also, `assert len(sollist) == order` will need to pass.
- `returns = 'both'`, return a dictionary `{'sol': <solution to ODE>, 'list': <list of linearly independent solutions>}`.

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = f(x).diff(x, 2)*x**2 - 4*f(x).diff(x)*x + 6*f(x)
>>> pprint(dsolve(eq, f(x),
... hint='nth_linear_euler_eq_homogeneous'))
        2
f(x) = x *(C1 + C2*x)
```

References

- https://en.wikipedia.org/wiki/Cauchy%E2%80%93Euler_equation
- C. Bender & S. Orszag, “Advanced Mathematical Methods for Scientists and Engineers”, Springer 1999, pp. 12

indirect doctest

nth_linear_euler_eq_nonhomogeneous_variation_of_parameters

class sympy.solvers.ode.single.**NthLinearEulerEqNonhomogeneousVariationOfParameters**(*ode_prob*

Solves an nth order linear non homogeneous Cauchy-Euler equidimensional ordinary differential equation using variation of parameters.

This is an equation with form $g(x) = a_0 f(x) + a_1 x f'(x) + a_2 x^2 f''(x) \cdots$.

This method works by assuming that the particular solution takes the form

$$\sum_{x=1}^{n} c_i(x) y_i(x) a_n x^n,$$

where y_i is the ith solution to the homogeneous equation. The solution is then solved using Wronskian's and Cramer's Rule. The particular solution is given by multiplying eq given below with $a_n x^n$

$$\sum_{x=1}^{n} \left(\int \frac{W_i(x)}{W(x)}\, dx \right) y_i(x),$$

where $W(x)$ is the Wronskian of the fundamental system (the system of n linearly independent solutions to the homogeneous equation), and $W_i(x)$ is the Wronskian of the fundamental system with the ith column replaced with $[0, 0, \cdots, 0, \frac{x^{-n}}{a_n} g(x)]$.

This method is general enough to solve any nth order inhomogeneous linear differential equation, but sometimes SymPy cannot simplify the Wronskian well enough to integrate it. If this method hangs, try using the `nth_linear_constant_coeff_variation_of_parameters_Integral` hint and simplifying the integrals manually. Also, prefer using `nth_linear_constant_coeff_undetermined_coefficients` when it applies, because it does not use integration, making it faster and more reliable.

Warning, using simplify=False with 'nth_linear_constant_coeff_variation_of_parameters' in *dsolve()* (page 755) may cause it to hang, because it will not attempt to simplify the Wronskian before integrating. It is recommended that you only use simplify=False with 'nth_linear_constant_coeff_variation_of_parameters_Integral' for this method, especially if the solution to the homogeneous equation has trigonometric functions in it.

Examples

```
>>> from sympy import Function, dsolve, Derivative
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = x**2*Derivative(f(x), x, x) - 2*x*Derivative(f(x), x) + 2*f(x) -
 ↪ x**4
>>> dsolve(eq, f(x),
... hint='nth_linear_euler_eq_nonhomogeneous_variation_of_parameters').
 ↪expand()
Eq(f(x), C1*x + C2*x**2 + x**4/6)
```

nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients

class sympy.solvers.ode.single.**NthLinearEulerEqNonhomogeneousUndeterminedCoefficients**(*ode_*

Solves an nth order linear non homogeneous Cauchy-Euler equidimensional ordinary differential equation using undetermined coefficients.

This is an equation with form $g(x) = a_0 f(x) + a_1 x f'(x) + a_2 x^2 f''(x) \cdots$.

These equations can be solved in a general manner, by substituting solutions of the form $x = exp(t)$, and deriving a characteristic equation of form $g(exp(t)) = b_0 f(t) + b_1 f'(t) + b_2 f''(t) \cdots$ which can be then solved by nth_linear_constant_coeff_undetermined_coefficients if g(exp(t)) has finite number of linearly independent derivatives.

Functions that fit this requirement are finite sums functions of the form $a x^i e^{bx} \sin(cx + d)$ or $a x^i e^{bx} \cos(cx + d)$, where i is a non-negative integer and a, b, c, and d are constants. For example any polynomial in x, functions like $x^2 e^{2x}$, $x \sin(x)$, and $e^x \cos(x)$ can all be used. Products of sin's and cos's have a finite number of derivatives, because they can be expanded into $\sin(ax)$ and $\cos(bx)$ terms. However, SymPy currently cannot do that expansion, so you will need to manually rewrite the expression in terms of the above to use this method. So, for example, you will need to manually convert $\sin^2(x)$ into $(1 + \cos(2x))/2$ to properly apply the method of undetermined coefficients on it.

After replacement of x by exp(t), this method works by creating a trial function from the expression and all of its linear independent derivatives and substituting them into the original ODE. The coefficients for each term will be a system of linear equations, which are be solved for and substituted, giving the solution. If any of the trial functions are linearly dependent on the solution to the homogeneous equation, they are multiplied by sufficient x to make them linearly independent.

Examples

```
>>> from sympy import dsolve, Function, Derivative, log
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = x**2*Derivative(f(x), x, x) - 2*x*Derivative(f(x), x) + 2*f(x) -
 ↪ log(x)
>>> dsolve(eq, f(x),
... hint='nth_linear_euler_eq_nonhomogeneous_undetermined_coefficients').
 ↪expand()
Eq(f(x), C1*x + C2*x**2 + log(x)/2 + 3/4)
```

nth_algebraic

class sympy.solvers.ode.single.**NthAlgebraic**(*ode_problem*)

Solves an nth order ordinary differential equation using algebra and integrals.

There is no general form for the kind of equation that this can solve. The the equation is solved algebraically treating differentiation as an invertible algebraic function.

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(f(x) * (f(x).diff(x)**2 - 1), 0)
>>> dsolve(eq, f(x), hint='nth_algebraic')
[Eq(f(x), 0), Eq(f(x), C1 - x), Eq(f(x), C1 + x)]
```

Note that this solver can return algebraic solutions that do not have any integration constants (f(x) = 0 in the above example).

nth_order_reducible

class sympy.solvers.ode.single.**NthOrderReducible**(*ode_problem*)

Solves ODEs that only involve derivatives of the dependent variable using a substitution of the form $f^n(x) = g(x)$.

For example any second order ODE of the form $f''(x) = h(f'(x), x)$ can be transformed into a pair of 1st order ODEs $g'(x) = h(g(x), x)$ and $f'(x) = g(x)$. Usually the 1st order ODE for g is easier to solve. If that gives an explicit solution for g then f is found simply by integration.

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = Eq(x*f(x).diff(x)**2 + f(x).diff(x, 2), 0)
>>> dsolve(eq, f(x), hint='nth_order_reducible')
...
Eq(f(x), C1 - sqrt(-1/C2)*log(-C2*sqrt(-1/C2) + x) + sqrt(-1/
↪C2)*log(C2*sqrt(-1/C2) + x))
```

separable

class sympy.solvers.ode.single.**Separable**(*ode_problem*)

Solves separable 1st order differential equations.

This is any differential equation that can be written as $P(y)\frac{dy}{dx} = Q(x)$. The solution can then just be found by rearranging terms and integrating: $\int P(y)\,dy = \int Q(x)\,dx$. This hint uses *sympy.simplify.simplify.separatevars()* (page 664) as its back end, so if a separable equation is not caught by this solver, it is most likely the fault of that function. *separatevars()* (page 664) is smart enough to do most expansion and factoring necessary to convert a separable equation $F(x, y)$ into the proper form $P(x) \cdot Q(y)$. The general solution is:

```
>>> from sympy import Function, dsolve, Eq, pprint
>>> from sympy.abc import x
>>> a, b, c, d, f = map(Function, ['a', 'b', 'c', 'd', 'f'])
>>> genform = Eq(a(x)*b(f(x))*f(x).diff(x), c(x)*d(f(x)))
>>> pprint(genform)
             d
a(x)*b(f(x))*--(f(x)) = c(x)*d(f(x))
             dx
>>> pprint(dsolve(genform, f(x), hint='separable_Integral'))
     f(x)
   /                  /
  |                  |
  |  b(y)            | c(x)
  |  ---- dy = C1 +  | ---- dx
  |  d(y)            | a(x)
  |                  |
 /                  /
```

Examples

```
>>> from sympy import Function, dsolve, Eq
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(Eq(f(x)*f(x).diff(x), x, 3*x*f(x)**2), f(x),
... hint='separable', simplify=False))
   /   2       \          2
log\3*f (x) - 1/         x
---------------- = C1 + --
       6                 2
```

References

- M. Tenenbaum & H. Pollard, “Ordinary Differential Equations”, Dover 1963, pp. 52

indirect doctest

almost_linear

class sympy.solvers.ode.single.**AlmostLinear**(*ode_problem*)

Solves an almost-linear differential equation.

The general form of an almost linear differential equation is

$$a(x)g'(f(x))f'(x) + b(x)g(f(x)) + c(x)$$

Here $f(x)$ is the function to be solved for (the dependent variable). The substitution $g(f(x)) = u(x)$ leads to a linear differential equation for $u(x)$ of the form $a(x)u' + b(x)u + c(x) = 0$. This can be solved for $u(x)$ by the $first_linear$ hint and then $f(x)$ is found by solving $g(f(x)) = u(x)$.

Examples

```
>>> from sympy import dsolve, Function, pprint, sin, cos
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = x*d + x*f(x) + 1
>>> dsolve(eq, f(x), hint='almost_linear')
Eq(f(x), (C1 - Ei(x))*exp(-x))
>>> pprint(dsolve(eq, f(x), hint='almost_linear'))
                     -x
f(x) = (C1 - Ei(x))*e
>>> example = cos(f(x))*f(x).diff(x) + sin(f(x)) + 1
>>> pprint(example)
                     d
sin(f(x)) + cos(f(x))*--(f(x)) + 1
                     dx
>>> pprint(dsolve(example, f(x), hint='almost_linear'))
                /    -x    \              /    -x    \
[f(x) = pi - asin\C1*e   - 1/, f(x) = asin\C1*e   - 1/]
```

See also:

sympy.solvers.ode.single.FirstLinear (page 776)

References

- Joel Moses, "Symbolic Integration - The Stormy Decade", Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

linear_coefficients

class sympy.solvers.ode.single.**LinearCoefficients**(*ode_problem*)

Solves a differential equation with linear coefficients.

The general form of a differential equation with linear coefficients is

$$y' + F\left(\frac{a_1 x + b_1 y + c_1}{a_2 x + b_2 y + c_2}\right) = 0,$$

where a_1, b_1, c_1, a_2, b_2, c_2 are constants and $a_1 b_2 - a_2 b_1 \neq 0$.

This can be solved by substituting:

$$x = x' + \frac{b_2 c_1 - b_1 c_2}{a_2 b_1 - a_1 b_2}$$
$$y = y' + \frac{a_1 c_2 - a_2 c_1}{a_2 b_1 - a_1 b_2}.$$

This substitution reduces the equation to a homogeneous differential equation.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> df = f(x).diff(x)
>>> eq = (x + f(x) + 1)*df + (f(x) - 6*x + 1)
>>> dsolve(eq, hint='linear_coefficients')
[Eq(f(x), -x - sqrt(C1 + 7*x**2) - 1), Eq(f(x), -x + sqrt(C1 + 7*x**2) -␣
↪1)]
>>> pprint(dsolve(eq, hint='linear_coefficients'))
                ____________                    ____________
               /        2                      /        2
[f(x) = -x - \/  C1 + 7*x   - 1, f(x) = -x + \/  C1 + 7*x   - 1]
```

See also:

sympy.solvers.ode.single.HomogeneousCoeffBest (page 772), *sympy.solvers.ode.single.HomogeneousCoeffSubsIndepDivDep* (page 774), *sympy.solvers.ode.single.HomogeneousCoeffSubsDepDivIndep* (page 773)

References

- Joel Moses, "Symbolic Integration - The Stormy Decade", Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

separable_reduced

class sympy.solvers.ode.single.**SeparableReduced**(*ode_problem*)

Solves a differential equation that can be reduced to the separable form.

The general form of this equation is

$$y' + (y/x)H(x^n y) = 0.$$

This can be solved by substituting $u(y) = x^n y$. The equation then reduces to the separable form $\frac{u'}{u(\text{power} - H(u))} - \frac{1}{x} = 0$.

The general solution is:

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x, n
>>> f, g = map(Function, ['f', 'g'])
>>> genform = f(x).diff(x) + (f(x)/x)*g(x**n*f(x))
>>> pprint(genform)
                  / n     \
d          f(x)*g\x *f(x)/
--(f(x)) + ---------------
dx                 x
>>> pprint(dsolve(genform, hint='separable_reduced'))
 n
x *f(x)
```

(continues on next page)

(continued from previous page)

```
 /
|
|         1
|    ------------ dy = C1 + log(x)
|    y*(n - g(y))
|
/
```

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> d = f(x).diff(x)
>>> eq = (x - x**2*f(x))*d - f(x)
>>> dsolve(eq, hint='separable_reduced')
[Eq(f(x), (1 - sqrt(C1*x**2 + 1))/x), Eq(f(x), (sqrt(C1*x**2 + 1) + 1)/
↪x)]
>>> pprint(dsolve(eq, hint='separable_reduced'))
                ____________              ____________
              /     2                   /     2
        1 - \/  C1*x  + 1           \/  C1*x  + 1  + 1
[f(x) = ------------------, f(x) = -------------------]
                x                           x
```

See also:

sympy.solvers.ode.single.Separable (page 789)

References

- Joel Moses, “Symbolic Integration - The Stormy Decade”, Communications of the ACM, Volume 14, Number 8, August 1971, pp. 558

lie_group

class sympy.solvers.ode.single.**LieGroup**(*ode_problem*)

This hint implements the Lie group method of solving first order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE can be easily solved by quadrature. It makes use of the *sympy.solvers.ode.infinitesimals()* (page 764) function which returns the infinitesimals of the transformation.

The coordinates r and s can be found by solving the following Partial Differential Equations.

$$\xi\frac{\partial r}{\partial x} + \eta\frac{\partial r}{\partial y} = 0$$

$$\xi\frac{\partial s}{\partial x} + \eta\frac{\partial s}{\partial y} = 1$$

The differential equation becomes separable in the new coordinate system

$$\frac{ds}{dr} = \frac{\frac{\partial s}{\partial x} + h(x,y)\frac{\partial s}{\partial y}}{\frac{\partial r}{\partial x} + h(x,y)\frac{\partial r}{\partial y}}$$

After finding the solution by integration, it is then converted back to the original coordinate system by substituting r and s in terms of x and y again.

Examples

```
>>> from sympy import Function, dsolve, exp, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> pprint(dsolve(f(x).diff(x) + 2*x*f(x) - x*exp(-x**2), f(x),
... hint='lie_group'))
       /      2\    2
       |     x |  -x
f(x) = |C1 + --|*e
       \     2 /
```

References

- Solving differential equations by Symmetry Groups, John Starrett, pp. 1 - pp. 14

2nd_hypergeometric

class sympy.solvers.ode.single.**SecondHypergeometric**(*ode_problem*)

Solves 2nd order linear differential equations.

It computes special function solutions which can be expressed using the 2F1, 1F1 or 0F1 hypergeometric functions.

$$y'' + A(x)y' + B(x)y = 0,$$

where A and B are rational functions.

These kinds of differential equations have solution of non-Liouvillian form.

Given linear ODE can be obtained from 2F1 given by

$$(x^2 - x)y'' + ((a + b + 1)x - c)y' + bay = 0,$$

where {a, b, c} are arbitrary constants.

Notes

The algorithm should find any solution of the form

$$y = P(x)_p F_q(..;..;\frac{\alpha x^k + \beta}{\gamma x^k + \delta}),$$

where pFq is any of 2F1, 1F1 or 0F1 and P is an "arbitrary function". Currently only the 2F1 case is implemented in SymPy but the other cases are described in the paper and could be implemented in future (contributions welcome!).

Examples

```
>>> from sympy import Function, dsolve, pprint
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (x*x - x)*f(x).diff(x,2) + (5*x - 1)*f(x).diff(x) + 4*f(x)
>>> pprint(dsolve(eq, f(x), '2nd_hypergeometric'))
                                                _
       /        /           4  \\  |_  /-1, -1 |  \
       |C1 + C2*|log(x) + -----||* |   |       | x|
       \        \         x + 1// 2  1 \  1    |  /
f(x) = ------------------------------------------
                             3
                      (x - 1)
```

References

- "Non-Liouvillian solutions for second order linear ODEs" by L. Chan, E.S. Cheb-Terrab

1st_power_series

sympy.solvers.ode.ode.**ode_1st_power_series**(*eq*, *func*, *order*, *match*)

The power series solution is a method which gives the Taylor series expansion to the solution of a differential equation.

For a first order differential equation $\frac{dy}{dx} = h(x, y)$, a power series solution exists at a point $x = x_0$ if $h(x, y)$ is analytic at x_0. The solution is given by

$$y(x) = y(x_0) + \sum_{n=1}^{\infty} \frac{F_n(x_0, b)(x - x_0)^n}{n!},$$

where $y(x_0) = b$ is the value of y at the initial value of x_0. To compute the values of the $F_n(x_0, b)$ the following algorithm is followed, until the required number of terms are generated.

1. $F_1 = h(x_0, b)$
2. $F_{n+1} = \frac{\partial F_n}{\partial x} + \frac{\partial F_n}{\partial y} F_1$

Examples

```
>>> from sympy import Function, pprint, exp, dsolve
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = exp(x)*(f(x).diff(x)) - f(x)
>>> pprint(dsolve(eq, hint='1st_power_series'))
                         3       4       5
                    C1*x    C1*x    C1*x     / 6\
f(x) = C1 + C1*x - ----- + ----- + ----- + O\x /
                     6       24      60
```

References

- Travis W. Walker, Analytic power series technique for solving first-order differential equations, p.p 17, 18

2nd_power_series_ordinary

sympy.solvers.ode.ode.**ode_2nd_power_series_ordinary**(*eq*, *func*, *order*, *match*)

Gives a power series solution to a second order homogeneous differential equation with polynomial coefficients at an ordinary point. A homogeneous differential equation is of the form

$$P(x)\frac{d^2y}{dx^2} + Q(x)\frac{dy}{dx} + R(x)y(x) = 0$$

For simplicity it is assumed that $P(x)$, $Q(x)$ and $R(x)$ are polynomials, it is sufficient that $\frac{Q(x)}{P(x)}$ and $\frac{R(x)}{P(x)}$ exists at x_0. A recurrence relation is obtained by substituting y as $\sum_{n=0}^{\infty} a_n x^n$, in the differential equation, and equating the nth term. Using this relation various terms can be generated.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = f(x).diff(x, 2) + f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_ordinary'))
          / 4    2    \        /     2\
          |x    x     |        |    x |    / 6\
f(x) = C2*|-- - -- + 1| + C1*x*|1 - --| + O\x /
          \24   2     /        \    6 /
```