References

- http://tutorial.math.lamar.edu/Classes/DE/SeriesSolutions.aspx
- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

2nd_power_series_regular

sympy.solvers.ode.ode.**ode_2nd_power_series_regular**(*eq, func, order, match*)

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

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

A point is said to regular singular at $x0$ if $x - x0\frac{Q(x)}{P(x)}$ and $(x - x0)^2\frac{R(x)}{P(x)}$ are analytic at $x0$. For simplicity $P(x)$, $Q(x)$ and $R(x)$ are assumed to be polynomials. The algorithm for finding the power series solutions is:

1. Try expressing $(x - x0)P(x)$ and $((x - x0)^2)Q(x)$ as power series solutions about x0. Find $p0$ and $q0$ which are the constants of the power series expansions.
2. Solve the indicial equation $f(m) = m(m - 1) + m * p0 + q0$, to obtain the roots $m1$ and $m2$ of the indicial equation.
3. If $m1 - m2$ is a non integer there exists two series solutions. If $m1 = m2$, there exists only one solution. If $m1 - m2$ is an integer, then the existence of one solution is confirmed. The other solution may or may not exist.

The power series solution is of the form $x^m \sum_{n=0}^{\infty} a_n x^n$. The coefficients are determined by the following recurrence relation. $a_n = -\frac{\sum_{k=0}^{n-1} q_{n-k} + (m+k)p_{n-k}}{f(m+n)}$. For the case in which $m1 - m2$ is an integer, it can be seen from the recurrence relation that for the lower root m, when n equals the difference of both the roots, the denominator becomes zero. So if the numerator is not equal to zero, a second series solution exists.

Examples

```
>>> from sympy import dsolve, Function, pprint
>>> from sympy.abc import x
>>> f = Function("f")
>>> eq = x*(f(x).diff(x, 2)) + 2*(f(x).diff(x)) + x*f(x)
>>> pprint(dsolve(eq, hint='2nd_power_series_regular'))
                              /    6    4    2    \
                              |   x    x    x     |
          /  4    2    \   C1*|- --- + -- - -- + 1|
          | x    x     |      \  720   24   2     /    / 6\
f(x) = C2*|--- - -- + 1| + ------------------------ + O\x /
          \120   6     /              x
```

References

- George E. Simmons, "Differential Equations with Applications and Historical Notes", p.p 176 - 184

Lie heuristics

These functions are intended for internal use of the Lie Group Solver. Nonetheless, they contain useful information in their docstrings on the algorithms implemented for the various heuristics.

abaco1_simple

`sympy.solvers.ode.lie_group.`**`lie_heuristic_abaco1_simple`**`(`*match*, *comp=False*`)`

The first heuristic uses the following four sets of assumptions on ξ and η

$$\xi = 0, \eta = f(x)$$

$$\xi = 0, \eta = f(y)$$

$$\xi = f(x), \eta = 0$$

$$\xi = f(y), \eta = 0$$

The success of this heuristic is determined by algebraic factorisation. For the first assumption $\xi = 0$ and η to be a function of x, the PDE

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

reduces to $f'(x) - f\frac{\partial h}{\partial y} = 0$ If $\frac{\partial h}{\partial y}$ is a function of x, then this can usually be integrated easily. A similar idea is applied to the other 3 assumptions as well.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A,C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco1_product

`sympy.solvers.ode.lie_group.`**`lie_heuristic_abaco1_product`**`(`*match*, *comp=False*`)`

The second heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) * g(y)$$

$$\eta = f(x) * g(y), \xi = 0$$

The first assumption of this heuristic holds good if $\frac{1}{h^2}\frac{\partial^2}{\partial x \partial y}\log(h)$ is separable in x and y, then the separated factors containing x is $f(x)$, and $g(y)$ is obtained by

$$e^{\int f \frac{\partial}{\partial x}\left(\frac{1}{f*h}\right) dy}$$

provided $f\frac{\partial}{\partial x}\left(\frac{1}{f*h}\right)$ is a function of y only.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y,x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) * g(y)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

bivariate

`sympy.solvers.ode.lie_group.`**`lie_heuristic_bivariate`**(*match*, *comp=False*)

The third heuristic assumes the infinitesimals ξ and η to be bi-variate polynomials in x and y. The assumption made here for the logic below is that h is a rational function in x and y though that may not be necessary for the infinitesimals to be bivariate polynomials. The coefficients of the infinitesimals are found out by substituting them in the PDE and grouping similar terms that are polynomials and since they form a linear system, solve and check for non trivial solutions. The degree of the assumed bivariates are increased till a certain maximum value.

References

- Lie Groups and Differential Equations pp. 327 - pp. 329

chi

`sympy.solvers.ode.lie_group.`**`lie_heuristic_chi`**(*match*, *comp=False*)

The aim of the fourth heuristic is to find the function $\chi(x, y)$ that satisfies the PDE $\frac{d\chi}{dx} + h\frac{d\chi}{dx} - \frac{\partial h}{\partial y}\chi = 0$.

This assumes χ to be a bivariate polynomial in x and y. By intuition, h should be a rational function in x and y. The method used here is to substitute a general binomial for χ up to a certain maximum degree is reached. The coefficients of the polynomials, are calculated by by collecting terms of the same order in x and y.

After finding χ, the next step is to use $\eta = \xi * h + \chi$, to determine ξ and η. This can be done by dividing χ by h which would give $-\xi$ as the quotient and η as the remainder.

References

- E.S Cheb-Terrab, L.G.S Duarte and L.A,C.P da Mota, Computer Algebra Solving of First Order ODEs Using Symmetry Methods, pp. 8

abaco2_similar

sympy.solvers.ode.lie_group.**lie_heuristic_abaco2_similar**(*match*, *comp=False*)

This heuristic uses the following two assumptions on ξ and η

$$\eta = g(x), \xi = f(x)$$

$$\eta = f(y), \xi = g(y)$$

For the first assumption,

1. First $\frac{\frac{\partial h}{\partial y}}{\frac{\partial^2 h}{\partial yy}}$ is calculated. Let us say this value is A
2. If this is constant, then h is matched to the form $A(x) + B(x)e^{\frac{y}{C}}$ then, $\frac{e^{\int \frac{A(x)}{C} \,dx}}{B(x)}$ gives $f(x)$ and $A(x) * f(x)$ gives $g(x)$
3. Otherwise $\frac{\frac{\partial A}{\partial X}}{\frac{\partial A}{\partial Y}} = \gamma$ is calculated. If

   a] γ is a function of x alone

   b] $\frac{\gamma \frac{\partial h}{\partial y} - \gamma'(x) - \frac{\partial h}{\partial x}}{h + \gamma} = G$ is a function of x alone. then, $e^{\int G \,dx}$ gives $f(x)$ and $-\gamma * f(x)$ gives $g(x)$

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y,x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(x)$, the coordinates are again interchanged, to get ξ as $f(x^*)$ and η as $g(y^*)$

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

function_sum

sympy.solvers.ode.lie_group.**lie_heuristic_function_sum**(*match*, *comp=False*)

This heuristic uses the following two assumptions on ξ and η

$$\eta = 0, \xi = f(x) + g(y)$$

$$\eta = f(x) + g(y), \xi = 0$$

The first assumption of this heuristic holds good if

$$\frac{\partial}{\partial y}[(h\frac{\partial^2}{\partial x^2}(h^{-1}))^{-1}]$$

is separable in x and y,

1. The separated factors containing y is $\frac{\partial g}{\partial y}$. From this $g(y)$ can be determined.
2. The separated factors containing x is $f''(x)$.
3. $h\frac{\partial^2}{\partial x^2}(h^{-1})$ equals $\frac{f''(x)}{f(x)+g(y)}$. From this $f(x)$ can be determined.

The second assumption holds good if $\frac{dy}{dx} = h(x, y)$ is rewritten as $\frac{dy}{dx} = \frac{1}{h(y,x)}$ and the same properties of the first assumption satisfies. After obtaining $f(x)$ and $g(y)$, the coordinates are again interchanged, to get η as $f(x) + g(y)$.

For both assumptions, the constant factors are separated among $g(y)$ and $f''(x)$, such that $f''(x)$ obtained from 3] is the same as that obtained from 2]. If not possible, then this heuristic fails.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 7 - pp. 8

abaco2_unique_unknown

sympy.solvers.ode.lie_group.**lie_heuristic_abaco2_unique_unknown**(*match*, *comp=False*)

This heuristic assumes the presence of unknown functions or known functions with non-integer powers.

1. A list of all functions and non-integer powers containing x and y
2. Loop over each element f in the list, find $\frac{\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial x}} = R$

   If it is separable in x and y, let X be the factors containing x. Then

   a] Check if $\xi = X$ and $\eta = -\frac{X}{R}$ satisfy the PDE. If yes, then return
   ξ and η

   b] Check if $\xi = \frac{-R}{X}$ and $\eta = -\frac{1}{X}$ satisfy the PDE.
   If yes, then return ξ and η

   If not, then check if

   a] $\xi = -R, \eta = 1$

   b] $\xi = 1, \eta = -\frac{1}{R}$

   are solutions.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

abaco2_unique_general

sympy.solvers.ode.lie_group.**lie_heuristic_abaco2_unique_general**(*match*, *comp=False*)

This heuristic finds if infinitesimals of the form $\eta = f(x)$, $\xi = g(y)$ without making any assumptions on h.

The complete sequence of steps is given in the paper mentioned below.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

linear

sympy.solvers.ode.lie_group.**lie_heuristic_linear**(*match*, *comp=False*)

This heuristic assumes

1. $\xi = ax + by + c$ and
2. $\eta = fx + gy + h$

After substituting the following assumptions in the determining PDE, it reduces to

$$f + (g - a)h - bh^2 - (ax + by + c)\frac{\partial h}{\partial x} - (fx + gy + c)\frac{\partial h}{\partial y}$$

Solving the reduced PDE obtained, using the method of characteristics, becomes impractical. The method followed is grouping similar terms and solving the system of linear equations obtained. The difference between the bivariate heuristic is that h need not be a rational function in this case.

References

- E.S. Cheb-Terrab, A.D. Roche, Symmetries and First Order ODE Patterns, pp. 10 - pp. 12

Rational Riccati Solver

These functions are intended for internal use to solve a first order Riccati differential equation with atleast one rational particular solution.

riccati_normal

sympy.solvers.ode.riccati.**riccati_normal**(*w*, *x*, *b1*, *b2*)

Given a solution $w(x)$ to the equation

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

and rational function coefficients $b_1(x)$ and $b_2(x)$, this function transforms the solution to give a solution $y(x)$ for its corresponding normal Riccati ODE

$$y'(x) + y(x)^2 = a(x)$$

using the transformation

$$y(x) = -b_2(x) * w(x) - b_2'(x)/(2 * b_2(x)) - b_1(x)/2$$

riccati_inverse_normal

sympy.solvers.ode.riccati.**riccati_inverse_normal**(*y*, *x*, *b1*, *b2*, *bp=None*)

Inverse transforming the solution to the normal Riccati ODE to get the solution to the Riccati ODE.

riccati_reduced

sympy.solvers.ode.riccati.**riccati_reduced**(*eq*, *f*, *x*)

Convert a Riccati ODE into its corresponding normal Riccati ODE.

construct_c

sympy.solvers.ode.riccati.**construct_c**(*num*, *den*, *x*, *poles*, *muls*)

Helper function to calculate the coefficients in the c-vector for each pole.

construct_d

sympy.solvers.ode.riccati.**construct_d**(*num*, *den*, *x*, *val_inf*)

Helper function to calculate the coefficients in the d-vector based on the valuation of the function at oo.

rational_laurent_series

sympy.solvers.ode.riccati.**rational_laurent_series**(*num*, *den*, *x*, *r*, *m*, *n*)

The function computes the Laurent series coefficients of a rational function.

Parameters

num: A Poly object that is the numerator of `f(x)`.

den: A Poly object that is the denominator of `f(x)`.

x: The variable of expansion of the series.

r: The point of expansion of the series.

m: Multiplicity of r if r is a pole of `f(x)`. Should

be zero otherwise.

n: Order of the term upto which the series is expanded.

Returns

series: A dictionary that has power of the term as key

and coefficient of that term as value.

Below is a basic outline of how the Laurent series of a

rational function $f(x)$ about x_0 is being calculated -

1. Substitute $x + x_0$ in place of x. If x_0

is a pole of $f(x)$, multiply the expression by x^m

where m is the multiplicity of x_0. Denote the

the resulting expression as g(x). We do this substitution

so that we can now find the Laurent series of g(x) about

$x = 0$.

2. We can then assume that the Laurent series of $g(x)$

takes the following form -

$$g(x) = \frac{num(x)}{den(x)} = \sum_{m=0}^{\infty} a_m x^m$$

where a_m denotes the Laurent series coefficients.

3. Multiply the denominator to the RHS of the equation

and form a recurrence relation for the coefficients a_m.

compute_m_ybar

`sympy.solvers.ode.riccati.compute_m_ybar(x, poles, choice, N)`

Helper function to calculate -

1. m - The degree bound for the polynomial solution that must be found for the auxiliary differential equation.
2. ybar - Part of the solution which can be computed using the poles, c and d vectors.

solve_aux_eq

`sympy.solvers.ode.riccati.solve_aux_eq(numa, dena, numy, deny, x, m)`

Helper function to find a polynomial solution of degree m for the auxiliary differential equation.

remove_redundant_sols

`sympy.solvers.ode.riccati.remove_redundant_sols(sol1, sol2, x)`

Helper function to remove redundant solutions to the differential equation.

get_gen_sol_from_part_sol

`sympy.solvers.ode.riccati.get_gen_sol_from_part_sol(part_sols, a, x)`

” Helper function which computes the general solution for a Riccati ODE from its particular solutions.

There are 3 cases to find the general solution from the particular solutions for a Riccati ODE depending on the number of particular solution(s) we have - 1, 2 or 3.

For more information, see Section 6 of “Methods of Solution of the Riccati Differential Equation” by D. R. Haaheim and F. M. Stein

solve_riccati

`sympy.solvers.ode.riccati.solve_riccati(fx, x, b0, b1, b2, gensol=False)`

The main function that gives particular/general solutions to Riccati ODEs that have atleast 1 rational particular solution.

System of ODEs

These functions are intended for internal use by `dsolve()` (page 755) for system of differential equations.

Linear, 2 equations, Order 1, Type 6

sympy.solvers.ode.ode.**_linear_2eq_order1_type6**(*x*, *y*, *t*, *r*, *eq*)

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = a[f(t) + ah(t)]x + a[g(t) - h(t)]y$$

This is solved by first multiplying the first equation by $-a$ and adding it to the second equation to obtain

$$y' - ax' = -ah(t)(y - ax)$$

Setting $U = y - ax$ and integrating the equation we arrive at

$$y - ax = C_1 e^{-a \int h(t) \, dt}$$

and on substituting the value of y in first equation give rise to first order ODEs. After solving for x, we can obtain y by substituting the value of x in second equation.

Linear, 2 equations, Order 1, Type 7

sympy.solvers.ode.ode.**_linear_2eq_order1_type7**(*x*, *y*, *t*, *r*, *eq*)

The equations of this type of ode are .

$$x' = f(t)x + g(t)y$$

$$y' = h(t)x + p(t)y$$

Differentiating the first equation and substituting the value of y from second equation will give a second-order linear equation

$$gx'' - (fg + gp + g')x' + (fgp - g^2h + fg' - f'g)x = 0$$

This above equation can be easily integrated if following conditions are satisfied.

1. $fgp - g^2h + fg' - f'g = 0$
2. $fgp - g^2h + fg' - f'g = ag, fg + gp + g' = bg$

If first condition is satisfied then it is solved by current dsolve solver and in second case it becomes a constant coefficient differential equation which is also solved by current solver.

Otherwise if the above condition fails then, a particular solution is assumed as $x = x_0(t)$ and $y = y_0(t)$ Then the general solution is expressed as

$$x = C_1 x_0(t) + C_2 x_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} \, dt$$

$$y = C_1 y_0(t) + C_2 [\frac{F(t)P(t)}{x_0(t)} + y_0(t) \int \frac{g(t)F(t)P(t)}{x_0^2(t)} \, dt]$$

where C1 and C2 are arbitrary constants and

$$F(t) = e^{\int f(t) \, dt}, P(t) = e^{\int p(t) \, dt}$$

Linear ODE to matrix

`sympy.solvers.ode.systems.`**`linear_ode_to_matrix`**(*eqs*, *funcs*, *t*, *order*)

Convert a linear system of ODEs to matrix form

Parameters

eqs : list of SymPy expressions or equalities

The equations as expressions (assumed equal to zero).

funcs : list of applied functions

The dependent variables of the system of ODEs.

t : symbol

The independent variable.

order : int

The order of the system of ODEs.

Returns

The tuple `(As, b)` where `As` is a tuple of matrices and `b` is the

the matrix representing the rhs of the matrix equation.

Raises

ODEOrderError

When the system of ODEs have an order greater than what was specified

ODENonlinearError

When the system of ODEs is nonlinear

Explanation

Express a system of linear ordinary differential equations as a single matrix differential equation [1]. For example the system $x' = x + y + 1$ and $y' = x - y$ can be represented as

$$A_1 X' = A_0 X + b$$

where A_1 and A_0 are 2×2 matrices and b, X and X' are 2×1 matrices with $X = [x, y]^T$.

Higher-order systems are represented with additional matrices e.g. a second-order system would look like

$$A_2 X'' = A_1 X' + A_0 X + b$$

Examples

```
>>> from sympy import Function, Symbol, Matrix, Eq
>>> from sympy.solvers.ode.systems import linear_ode_to_matrix
>>> t = Symbol('t')
>>> x = Function('x')
>>> y = Function('y')
```

We can create a system of linear ODEs like

```
>>> eqs = [
...     Eq(x(t).diff(t), x(t) + y(t) + 1),
...     Eq(y(t).diff(t), x(t) - y(t)),
... ]
>>> funcs = [x(t), y(t)]
>>> order = 1 # 1st order system
```

Now `linear_ode_to_matrix` can represent this as a matrix differential equation.

```
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, t, order)
>>> A1
Matrix([
[1, 0],
[0, 1]])
>>> A0
Matrix([
[1, 1],
[1,  -1]])
>>> b
Matrix([
[1],
[0]])
```

The original equations can be recovered from these matrices:

```
>>> eqs_mat = Matrix([eq.lhs - eq.rhs for eq in eqs])
>>> X = Matrix(funcs)
>>> A1 * X.diff(t) - A0 * X - b == eqs_mat
True
```

If the system of equations has a maximum order greater than the order of the system specified, a ODEOrderError exception is raised.

```
>>> eqs = [Eq(x(t).diff(t, 2), x(t).diff(t) + x(t)), Eq(y(t).diff(t),␣
↪y(t) + x(t))]
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
ODEOrderError: Cannot represent system in 1-order form
```

If the system of equations is nonlinear, then ODENonlinearError is raised.

```
>>> eqs = [Eq(x(t).diff(t), x(t) + y(t)), Eq(y(t).diff(t), y(t)**2 +␣
↪x(t))]
```

(continues on next page)

(continued from previous page)

```
>>> linear_ode_to_matrix(eqs, funcs, t, 1)
Traceback (most recent call last):
...
ODENonlinearError: The system of ODEs is nonlinear.
```

See also:

linear_eq_to_matrix **(page 870)**
for systems of linear algebraic equations.

References

[R791]

Canonical Equations Converter

sympy.solvers.ode.systems.**canonical_odes**(*eqs, funcs, t*)
Function that solves for highest order derivatives in a system

Parameters
eqs : List
List of the ODEs

funcs : List
List of dependent variables

t : Symbol
Independent variable

Returns
List

Explanation

This function inputs a system of ODEs and based on the system, the dependent variables and their highest order, returns the system in the following form:

$$X'(t) = A(t)X(t) + b(t)$$

Here, $X(t)$ is the vector of dependent variables of lower order, $A(t)$ is the coefficient matrix, $b(t)$ is the non-homogeneous term and $X'(t)$ is the vector of dependent variables in their respective highest order. We use the term canonical form to imply the system of ODEs which is of the above form.

If the system passed has a non-linear term with multiple solutions, then a list of systems is returned in its canonical form.

Examples

```
>>> from sympy import symbols, Function, Eq, Derivative
>>> from sympy.solvers.ode.systems import canonical_odes
>>> f, g = symbols("f g", cls=Function)
>>> x, y = symbols("x y")
>>> funcs = [f(x), g(x)]
>>> eqs = [Eq(f(x).diff(x) - 7*f(x), 12*g(x)), Eq(g(x).diff(x) + g(x),␣
↪20*f(x))]
```

```
>>> canonical_eqs = canonical_odes(eqs, funcs, x)
>>> canonical_eqs
[[Eq(Derivative(f(x), x), 7*f(x) + 12*g(x)), Eq(Derivative(g(x), x),␣
↪20*f(x) - g(x))]]
```

```
>>> system = [Eq(Derivative(f(x), x)**2 - 2*Derivative(f(x), x) + 1, 4),␣
↪Eq(-y*f(x) + Derivative(g(x), x), 0)]
```

```
>>> canonical_system = canonical_odes(system, funcs, x)
>>> canonical_system
[[Eq(Derivative(f(x), x), -1), Eq(Derivative(g(x), x), y*f(x))],␣
↪[Eq(Derivative(f(x), x), 3), Eq(Derivative(g(x), x), y*f(x))]]
```

LinODESolve Systems Information

sympy.solvers.ode.systems.**linodesolve_type**(*A*, *t*, *b=None*)

Helper function that determines the type of the system of ODEs for solving with *sympy.solvers.ode.systems.linodesolve()* (page 814)

Parameters

A : Matrix

Coefficient matrix of the system of ODEs

b : Matrix or None

Non-homogeneous term of the system. The default value is None. If this argument is None, then the system is assumed to be homogeneous.

Returns

Dict

Raises

NotImplementedError

When the coefficient matrix does not have a commutative antiderivative

Explanation

This function takes in the coefficient matrix and/or the non-homogeneous term and returns the type of the equation that can be solved by *sympy.solvers.ode.systems.linodesolve()* (page 814).

If the system is constant coefficient homogeneous, then "type1" is returned

If the system is constant coefficient non-homogeneous, then "type2" is returned

If the system is non-constant coefficient homogeneous, then "type3" is returned

If the system is non-constant coefficient non-homogeneous, then "type4" is returned

If the system has a non-constant coefficient matrix which can be factorized into constant coefficient matrix, then "type5" or "type6" is returned for when the system is homogeneous or non-homogeneous respectively.

Note that, if the system of ODEs is of "type3" or "type4", then along with the type, the commutative antiderivative of the coefficient matrix is also returned.

If the system cannot be solved by *sympy.solvers.ode.systems.linodesolve()* (page 814), then NotImplementedError is raised.

Examples

```
>>> from sympy import symbols, Matrix
>>> from sympy.solvers.ode.systems import linodesolve_type
>>> t = symbols("t")
>>> A = Matrix([[1, 1], [2, 3]])
>>> b = Matrix([t, 1])
```

```
>>> linodesolve_type(A, t)
{'antiderivative': None, 'type_of_equation': 'type1'}
```

```
>>> linodesolve_type(A, t, b=b)
{'antiderivative': None, 'type_of_equation': 'type2'}
```

```
>>> A_t = Matrix([[1, t], [-t, 1]])
```

```
>>> linodesolve_type(A_t, t)
{'antiderivative': Matrix([
[      t, t**2/2],
[-t**2/2,      t]]), 'type_of_equation': 'type3'}
```

```
>>> linodesolve_type(A_t, t, b=b)
{'antiderivative': Matrix([
[      t, t**2/2],
[-t**2/2,      t]]), 'type_of_equation': 'type4'}
```

```
>>> A_non_commutative = Matrix([[1, t], [t, -1]])
>>> linodesolve_type(A_non_commutative, t)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError:
The system does not have a commutative antiderivative, it cannot be
solved by linodesolve.
```

See also:

linodesolve **(page 814)**
Function for which linodesolve_type gets the information

Matrix Exponential Jordan Form

sympy.solvers.ode.systems.**matrix_exp_jordan_form**(A, t)

Matrix exponential $\exp(A*t)$ for the matrix A and scalar t.

Parameters

A : Matrix

The matrix A in the expression $\exp(A*t)$

t : Symbol

The independent variable

Explanation

Returns the Jordan form of the $\exp(A*t)$ along with the matrix P such that:

$$\exp(A*t) = P*expJ*P^{-1}$$

Examples

```
>>> from sympy import Matrix, Symbol
>>> from sympy.solvers.ode.systems import matrix_exp, matrix_exp_jordan_
↪form
>>> t = Symbol('t')
```

We will consider a 2x2 defective matrix. This shows that our method works even for defective matrices.

```
>>> A = Matrix([[1, 1], [0, 1]])
```

It can be observed that this function gives us the Jordan normal form and the required invertible matrix P.

```
>>> P, expJ = matrix_exp_jordan_form(A, t)
```

Here, it is shown that P and expJ returned by this function is correct as they satisfy the formula: P * expJ * P_inverse = exp(A*t).

```
>>> P * expJ * P.inv() == matrix_exp(A, t)
True
```

References

[R792], [R793], [R794]

Matrix Exponential

sympy.solvers.ode.systems.**matrix_exp**(*A*, *t*)

Matrix exponential $\exp(A*t)$ for the matrix A and scalar t.

Parameters

A : Matrix

The matrix A in the expression $\exp(A*t)$

t : Symbol

The independent variable

Explanation

This functions returns the $\exp(A*t)$ by doing a simple matrix multiplication:

$$\exp(A*t) = P*expJ*P^{-1}$$

where $expJ$ is $\exp(J*t)$. J is the Jordan normal form of A and P is matrix such that:

$$A = P*J*P^{-1}$$

The matrix exponential $\exp(A*t)$ appears in the solution of linear differential equations. For example if x is a vector and A is a matrix then the initial value problem

$$\frac{dx(t)}{dt} = A \times x(t), x(0) = x0$$

has the unique solution

$$x(t) = \exp(At)x0$$

Examples

```
>>> from sympy import Symbol, Matrix, pprint
>>> from sympy.solvers.ode.systems import matrix_exp
>>> t = Symbol('t')
```

We will consider a 2x2 matrix for comupting the exponential

```
>>> A = Matrix([[2, -5], [2, -4]])
>>> pprint(A)
[2  -5]
[     ]
[2  -4]
```

Now, exp(A*t) is given as follows:

```
>>> pprint(matrix_exp(A, t))
[   -t            -t                     -t              ]
[3*e  *sin(t) + e  *cos(t)         -5*e  *sin(t)         ]
[                                                         ]
[         -t                      -t            -t       ]
[      2*e  *sin(t)         - 3*e  *sin(t) + e  *cos(t)]
```

See also:

matrix_exp_jordan_form **(page 812)**
For exponential of Jordan normal form

References

[R795], [R796]

Linear, n equations, Order 1 Solver

sympy.solvers.ode.systems.**linodesolve**(*A*, *t*, *b=None*, *B=None*, *type='auto'*, *doit=False*, *tau=None*)

System of n equations linear first-order differential equations

Parameters

A : Matrix

Coefficient matrix of the system of linear first order ODEs.

t : Symbol

Independent variable in the system of ODEs.

b : Matrix or None

Non-homogeneous term in the system of ODEs. If None is passed, a homogeneous system of ODEs is assumed.

B : Matrix or None

Antiderivative of the coefficient matrix. If the antiderivative is not passed and the solution requires the term, then the solver would compute it internally.

type : String

Type of the system of ODEs passed. Depending on the type, the solution is evaluated. The type values allowed and the corresponding system it solves are: "type1" for constant coefficient homogeneous "type2" for constant coefficient non-homogeneous, "type3" for non-constant coefficient homogeneous, "type4" for non-constant coefficient non-homogeneous, "type5" and "type6" for non-constant coefficient homogeneous and non-homogeneous systems respectively where the coefficient matrix can be factorized to a constant coefficient matrix. The default value is "auto" which will let the solver decide the correct type of the system passed.

doit : Boolean

Evaluate the solution if True, default value is False

tau: Expression

Used to substitute for the value of t after we get the solution of the system.

Returns

List

Raises

ValueError

This error is raised when the coefficient matrix, non-homogeneous term or the antiderivative, if passed, are not a matrix or do not have correct dimensions

NonSquareMatrixError

When the coefficient matrix or its antiderivative, if passed is not a square matrix

NotImplementedError

If the coefficient matrix does not have a commutative antiderivative

Explanation

This solver solves the system of ODEs of the follwing form:

$$X'(t) = A(t)X(t) + b(t)$$

Here, $A(t)$ is the coefficient matrix, $X(t)$ is the vector of n independent variables, $b(t)$ is the non-homogeneous term and $X'(t)$ is the derivative of $X(t)$

Depending on the properties of $A(t)$ and $b(t)$, this solver evaluates the solution differently.

When $A(t)$ is constant coefficient matrix and $b(t)$ is zero vector i.e. system is homogeneous, the system is "type1". The solution is:

$$X(t) = \exp(At)C$$

Here, C is a vector of constants and A is the constant coefficient matrix.

When $A(t)$ is constant coefficient matrix and $b(t)$ is non-zero i.e. system is non-homogeneous, the system is "type2". The solution is:

$$X(t) = e^{At}\left(\int e^{-At} b \, dt + C\right)$$

When $A(t)$ is coefficient matrix such that its commutative with its antiderivative $B(t)$ and $b(t)$ is a zero vector i.e. system is homogeneous, the system is “type3”. The solution is:

$$X(t) = \exp(B(t))C$$

When $A(t)$ is commutative with its antiderivative $B(t)$ and $b(t)$ is non-zero i.e. system is non-homogeneous, the system is “type4”. The solution is:

$$X(t) = e^{B(t)}(\int e^{-B(t)}b(t)\,dt + C)$$

When $A(t)$ is a coefficient matrix such that it can be factorized into a scalar and a constant coefficient matrix:

$$A(t) = f(t) * A$$

Where $f(t)$ is a scalar expression in the independent variable t and A is a constant matrix, then we can do the following substitutions:

$$tau = \int f(t)dt, X(t) = Y(tau), b(t) = b(f^{-1}(tau))$$

Here, the substitution for the non-homogeneous term is done only when its non-zero. Using these substitutions, our original system becomes:

$$Y'(tau) = A * Y(tau) + b(tau)/f(tau)$$

The above system can be easily solved using the solution for “type1” or “type2” depending on the homogeneity of the system. After we get the solution for $Y(tau)$, we substitute the solution for tau as t to get back $X(t)$

$$X(t) = Y(tau)$$

Systems of “type5” and “type6” have a commutative antiderivative but we use this solution because its faster to compute.

The final solution is the general solution for all the four equations since a constant coefficient matrix is always commutative with its antidervative.

An additional feature of this function is, if someone wants to substitute for value of the independent variable, they can pass the substitution tau and the solution will have the independent variable substituted with the passed expression(tau).

Examples

To solve the system of ODEs using this function directly, several things must be done in the right order. Wrong inputs to the function will lead to incorrect results.

```
>>> from sympy import symbols, Function, Eq
>>> from sympy.solvers.ode.systems import canonical_odes, linear_ode_to_
↪matrix, linodesolve, linodesolve_type
>>> from sympy.solvers.ode.subscheck import checkodesol
>>> f, g = symbols("f, g", cls=Function)
>>> x, a = symbols("x, a")
>>> funcs = [f(x), g(x)]
>>> eqs = [Eq(f(x).diff(x) - f(x), a*g(x) + 1), Eq(g(x).diff(x) + g(x),␣
↪a*f(x))]
```

Here, it is important to note that before we derive the coefficient matrix, it is important to get the system of ODEs into the desired form. For that we will use *sympy.solvers.ode.systems.canonical_odes()* (page 809).

```
>>> eqs = canonical_odes(eqs, funcs, x)
>>> eqs
[[Eq(Derivative(f(x), x), a*g(x) + f(x) + 1), Eq(Derivative(g(x), x),␣
↪a*f(x) - g(x))]]
```

Now, we will use *sympy.solvers.ode.systems.linear_ode_to_matrix()* (page 807) to get the coefficient matrix and the non-homogeneous term if it is there.

```
>>> eqs = eqs[0]
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)
>>> A = A0
```

We have the coefficient matrices and the non-homogeneous term ready. Now, we can use *sympy.solvers.ode.systems.linodesolve_type()* (page 810) to get the information for the system of ODEs to finally pass it to the solver.

```
>>> system_info = linodesolve_type(A, x, b=b)
>>> sol_vector = linodesolve(A, x, b=b, B=system_info['antiderivative'],␣
↪type=system_info['type_of_equation'])
```

Now, we can prove if the solution is correct or not by using *sympy.solvers.ode.checkodesol()* (page 762)

```
>>> sol = [Eq(f, s) for f, s in zip(funcs, sol_vector)]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

We can also use the doit method to evaluate the solutions passed by the function.

```
>>> sol_vector_evaluated = linodesolve(A, x, b=b, type="type2",␣
↪doit=True)
```

Now, we will look at a system of ODEs which is non-constant.

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

The system defined above is already in the desired form, so we do not have to convert it.

```
>>> (A1, A0), b = linear_ode_to_matrix(eqs, funcs, x, 1)
>>> A = A0
```

A user can also pass the commutative antiderivative required for type3 and type4 system of ODEs. Passing an incorrect one will lead to incorrect results. If the coefficient matrix is not commutative with its antiderivative, then *sympy.solvers.ode.systems.linodesolve_type()* (page 810) raises a NotImplementedError. If it does have a commutative antiderivative, then the function just returns the information about the system.

```
>>> system_info = linodesolve_type(A, x, b=b)
```

Now, we can pass the antiderivative as an argument to get the solution. If the system information is not passed, then the solver will compute the required arguments internally.

```
>>> sol_vector = linodesolve(A, x, b=b)
```

Once again, we can verify the solution obtained.

```
>>> sol = [Eq(f, s) for f, s in zip(funcs, sol_vector)]
>>> checkodesol(eqs, sol)
(True, [0, 0])
```

See also:

linear_ode_to_matrix **(page 807)**
Coefficient matrix computation function

canonical_odes **(page 809)**
System of ODEs representation change

linodesolve_type **(page 810)**
Getting information about systems of ODEs to pass in this solver

Nonlinear, 2 equations, Order 1, Type 1

sympy.solvers.ode.ode.**_nonlinear_2eq_order1_type1**(*x*, *y*, *t*, *eq*)

Equations:

$$x' = x^n F(x, y)$$

$$y' = g(y)F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)} \,dy = t + C_2$$

where

if $n \neq 1$

$$\varphi = [C_1 + (1 - n) \int \frac{1}{g(y)} \,dy]^{\frac{1}{1-n}}$$

if $n = 1$

$$\varphi = C_1 e^{\int \frac{1}{g(y)} \,dy}$$

where C_1 and C_2 are arbitrary constants.

Nonlinear, 2 equations, Order 1, Type 2

sympy.solvers.ode.ode.**_nonlinear_2eq_order1_type2**(*x*, *y*, *t*, *eq*)

Equations:

$$x' = e^{\lambda x} F(x, y)$$

$$y' = g(y)F(x, y)$$

Solution:

$$x = \varphi(y), \int \frac{1}{g(y)F(\varphi(y), y)}\,dy = t + C_2$$

where

if $\lambda \neq 0$

$$\varphi = -\frac{1}{\lambda} log(C_1 - \lambda \int \frac{1}{g(y)}\,dy)$$

if $\lambda = 0$

$$\varphi = C_1 + \int \frac{1}{g(y)}\,dy$$

where C_1 and C_2 are arbitrary constants.

Nonlinear, 2 equations, Order 1, Type 3

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type3`(*x*, *y*, *t*, *eq*)

Autonomous system of general form

$$x' = F(x, y)$$

$$y' = G(x, y)$$

Assuming $y = y(x, C_1)$ where C_1 is an arbitrary constant is the general solution of the first-order equation

$$F(x, y)y'_x = G(x, y)$$

Then the general solution of the original system of equations has the form

$$\int \frac{1}{F(x, y(x, C_1))}\,dx = t + C_1$$

Nonlinear, 2 equations, Order 1, Type 4

`sympy.solvers.ode.ode._nonlinear_2eq_order1_type4`(*x*, *y*, *t*, *eq*)

Equation:

$$x' = f_1(x)g_1(y)\phi(x, y, t)$$

$$y' = f_2(x)g_2(y)\phi(x, y, t)$$

First integral:

$$\int \frac{f_2(x)}{f_1(x)}\,dx - \int \frac{g_1(y)}{g_2(y)}\,dy = C$$

where C is an arbitrary constant.

On solving the first integral for x (resp., y) and on substituting the resulting expression into either equation of the original solution, one arrives at a first-order equation for determining y (resp., x).

Nonlinear, 2 equations, Order 1, Type 5

sympy.solvers.ode.ode.**_nonlinear_2eq_order1_type5**(*func*, *t*, *eq*)

Clairaut system of ODEs

$$x = tx' + F(x', y')$$

$$y = ty' + G(x', y')$$

The following are solutions of the system

(i) straight lines:

$$x = C_1 t + F(C_1, C_2), y = C_2 t + G(C_1, C_2)$$

where C_1 and C_2 are arbitrary constants;

(ii) envelopes of the above lines;

(iii) continuously differentiable lines made up from segments of the lines (i) and (ii).

Nonlinear, 3 equations, Order 1, Type 1

sympy.solvers.ode.ode.**_nonlinear_3eq_order1_type1**(*x*, *y*, *z*, *t*, *eq*)

Equations:

$$ax' = (b - c)yz, \;\; by' = (c - a)zx, \;\; cz' = (a - b)xy$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrives at a separable first-order equation on x. Similarly doing that for other two equations, we will arrive at first order equation on y and z too.

References

-http://eqworld.ipmnet.ru/en/solutions/sysode/sode0401.pdf

Nonlinear, 3 equations, Order 1, Type 2

sympy.solvers.ode.ode.**_nonlinear_3eq_order1_type2**(*x*, *y*, *z*, *t*, *eq*)

Equations:

$$ax' = (b - c)yzf(x, y, z, t)$$

$$by' = (c - a)zxf(x, y, z, t)$$

$$cz' = (a - b)xyf(x, y, z, t)$$

First Integrals:

$$ax^2 + by^2 + cz^2 = C_1$$

$$a^2x^2 + b^2y^2 + c^2z^2 = C_2$$

where C_1 and C_2 are arbitrary constants. On solving the integrals for y and z and on substituting the resulting expressions into the first equation of the system, we arrives at a first-order differential equations on x. Similarly doing that for other two equations we will arrive at first order equation on y and z.

References

-http://eqworld.ipmnet.ru/en/solutions/sysode/sode0402.pdf

Nonlinear, 3 equations, Order 1, Type 3

sympy.solvers.ode.ode.**_nonlinear_3eq_order1_type3**(*x, y, z, t, eq*)

Equations:

$$x' = cF_2 - bF_3, \quad y' = aF_3 - cF_1, \quad z' = bF_1 - aF_2$$

where $F_n = F_n(x, y, z, t)$.

1. First Integral:

$$ax + by + cz = C_1,$$

where C is an arbitrary constant.

2. If we assume function F_n to be independent of t,i.e, $F_n = F_n(x, y, z)$ Then, on eliminating t and z from the first two equation of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{aF_3(x, y, z) - cF_1(x, y, z)}{cF_2(x, y, z) - bF_3(x, y, z)}$$

where $z = \frac{1}{c}(C_1 - ax - by)$

References

-http://eqworld.ipmnet.ru/en/solutions/sysode/sode0404.pdf

Nonlinear, 3 equations, Order 1, Type 4

sympy.solvers.ode.ode.**_nonlinear_3eq_order1_type4**(*x, y, z, t, eq*)

Equations:

$$x' = czF_2 - byF_3, \quad y' = axF_3 - czF_1, \quad z' = byF_1 - axF_2$$

where $F_n = F_n(x, y, z, t)$

1. First integral:

$$ax^2 + by^2 + cz^2 = C_1$$

where C is an arbitrary constant.

2. Assuming the function F_n is independent of t: $F_n = F_n(x, y, z)$. Then on eliminating t and z from the first two equations of the system, one arrives at the first-order equation

$$\frac{dy}{dx} = \frac{axF_3(x, y, z) - czF_1(x, y, z)}{czF_2(x, y, z) - byF_3(x, y, z)}$$

where $z = \pm\sqrt{\frac{1}{c}(C_1 - ax^2 - by^2)}$

References

-http://eqworld.ipmnet.ru/en/solutions/sysode/sode0405.pdf

Nonlinear, 3 equations, Order 1, Type 5

sympy.solvers.ode.ode.**_nonlinear_3eq_order1_type5**(*x*, *y*, *z*, *t*, *eq*)

$$x' = x(cF_2 - bF_3), \quad y' = y(aF_3 - cF_1), \quad z' = z(bF_1 - aF_2)$$

where $F_n = F_n(x, y, z, t)$ and are arbitrary functions.

First Integral:

$$|x|^a |y|^b |z|^c = C_1$$

where C is an arbitrary constant. If the function F_n is independent of t, then, by eliminating t and z from the first two equations of the system, one arrives at a first-order equation.

References

-http://eqworld.ipmnet.ru/en/solutions/sysode/sode0406.pdf

Information on the ode module

This module contains `dsolve()` (page 755) and different helper functions that it uses.

`dsolve()` (page 755) solves ordinary differential equations. See the docstring on the various functions for their uses. Note that partial differential equations support is in `pde.py`. Note that hint functions have docstrings describing their various methods, but they are intended for internal use. Use `dsolve(ode, func, hint=hint)` to solve an ODE using a specific hint. See also the docstring on `dsolve()` (page 755).

Functions in this module

These are the user functions in this module:

- `dsolve()` (page 755) - Solves ODEs.
- `classify_ode()` (page 760) - Classifies ODEs into possible hints for `dsolve()` (page 755).

- *checkodesol()* (page 762) - Checks if an equation is the solution to an ODE.
- *homogeneous_order()* (page 763) - Returns the homogeneous order of an expression.
- *infinitesimals()* (page 764) - Returns the infinitesimals of the Lie group of point transformations of an ODE, such that it is invariant.
- *checkinfsol()* (page 765) - Checks if the given infinitesimals are the actual infinitesimals of a first order ODE.

These are the non-solver helper functions that are for internal use. The user should use the various options to *dsolve()* (page 755) to obtain the functionality provided by these functions:

- *odesimp()* (page 767) - Does all forms of ODE simplification.
- *ode_sol_simplicity()* (page 769) - A key function for comparing solutions by simplicity.
- *constantsimp()* (page 765) - Simplifies arbitrary constants.
- *constant_renumber()* (page 768) - Renumber arbitrary constants.
- *_handle_Integral()* (page 826) - Evaluate unevaluated Integrals.

See also the docstrings of these functions.

Currently implemented solver methods

The following methods are implemented for solving ordinary differential equations. See the docstrings of the various hint functions for more information on each (run `help(ode)`):

- 1st order separable differential equations.
- 1st order differential equations whose coefficients or dx and dy are functions homogeneous of the same order.
- 1st order exact differential equations.
- 1st order linear differential equations.
- 1st order Bernoulli differential equations.
- Power series solutions for first order differential equations.
- Lie Group method of solving first order differential equations.
- 2nd order Liouville differential equations.
- Power series solutions for second order differential equations at ordinary and regular singular points.
- nth order differential equation that can be solved with algebraic rearrangement and integration.
- nth order linear homogeneous differential equation with constant coefficients.
- nth order linear inhomogeneous differential equation with constant coefficients using the method of undetermined coefficients.
- nth order linear inhomogeneous differential equation with constant coefficients using the method of variation of parameters.

Philosophy behind this module

This module is designed to make it easy to add new ODE solving methods without having to mess with the solving code for other methods. The idea is that there is a *classify_ode()* (page 760) function, which takes in an ODE and tells you what hints, if any, will solve the ODE. It does this without attempting to solve the ODE, so it is fast. Each solving method is a hint, and it has its own function, named `ode_<hint>`. That function takes in the ODE and any match expression gathered by *classify_ode()* (page 760) and returns a solved result. If this result has any integrals in it, the hint function will return an unevaluated *Integral* (page 601) class. *dsolve()* (page 755), which is the user wrapper function around all of this, will then call *odesimp()* (page 767) on the result, which, among other things, will attempt to solve the equation for the dependent variable (the function we are solving for), simplify the arbitrary constants in the expression, and evaluate any integrals, if the hint allows it.

How to add new solution methods

If you have an ODE that you want *dsolve()* (page 755) to be able to solve, try to avoid adding special case code here. Instead, try finding a general method that will solve your ODE, as well as others. This way, the *ode* (page 755) module will become more robust, and unhindered by special case hacks. WolphramAlpha and Maple's DETools[odeadvisor] function are two resources you can use to classify a specific ODE. It is also better for a method to work with an nth order ODE instead of only with specific orders, if possible.

To add a new method, there are a few things that you need to do. First, you need a hint name for your method. Try to name your hint so that it is unambiguous with all other methods, including ones that may not be implemented yet. If your method uses integrals, also include a `hint_Integral` hint. If there is more than one way to solve ODEs with your method, include a hint for each one, as well as a `<hint>_best` hint. Your `ode_<hint>_best()` function should choose the best using min with `ode_sol_simplicity` as the key argument. See *HomogeneousCoeffBest* (page 772), for example. The function that uses your method will be called `ode_<hint>()`, so the hint must only use characters that are allowed in a Python function name (alphanumeric characters and the underscore '_' character). Include a function for every hint, except for `_Integral` hints (*dsolve()* (page 755) takes care of those automatically). Hint names should be all lowercase, unless a word is commonly capitalized (such as Integral or Bernoulli). If you have a hint that you do not want to run with `all_Integral` that does not have an `_Integral` counterpart (such as a best hint that would defeat the purpose of `all_Integral`), you will need to remove it manually in the *dsolve()* (page 755) code. See also the *classify_ode()* (page 760) docstring for guidelines on writing a hint name.

Determine *in general* how the solutions returned by your method compare with other methods that can potentially solve the same ODEs. Then, put your hints in the *allhints* (page 767) tuple in the order that they should be called. The ordering of this tuple determines which hints are default. Note that exceptions are ok, because it is easy for the user to choose individual hints with *dsolve()* (page 755). In general, `_Integral` variants should go at the end of the list, and `_best` variants should go before the various hints they apply to. For example, the `undetermined_coefficients` hint comes before the `variation_of_parameters` hint because, even though variation of parameters is more general than undetermined coefficients, undetermined coefficients generally returns cleaner results for the ODEs that it can solve than variation of parameters does, and it does not require integration, so it is much faster.

Next, you need to have a match expression or a function that matches the type of the ODE, which you should put in *classify_ode()* (page 760) (if the match function is more than just a few lines. It should match the ODE without solving for it as much as possible, so that *classify_ode()* (page 760) remains fast and is not hindered by bugs in solving code. Be sure to consider corner cases. For example, if your solution method involves dividing by something, make sure you exclude the case where that division will be 0.

In most cases, the matching of the ODE will also give you the various parts that you need to solve it. You should put that in a dictionary (`.match()` will do this for you), and add that as `matching_hints['hint'] = matchdict` in the relevant part of *classify_ode()* (page 760). *classify_ode()* (page 760) will then send this to *dsolve()* (page 755), which will send it to your function as the match argument. Your function should be named `ode_<hint>(eq, func, order, match)`. If you need to send more information, put it in the ``match` dictionary. For example, if you had to substitute in a dummy variable in *classify_ode()* (page 760) to match the ODE, you will need to pass it to your function using the $match$ dict to access it. You can access the independent variable using `func.args[0]`, and the dependent variable (the function you are trying to solve for) as `func.func`. If, while trying to solve the ODE, you find that you cannot, raise `NotImplementedError`. *dsolve()* (page 755) will catch this error with the `all` meta-hint, rather than causing the whole routine to fail.

Add a docstring to your function that describes the method employed. Like with anything else in SymPy, you will need to add a doctest to the docstring, in addition to real tests in `test_ode.py`. Try to maintain consistency with the other hint functions' docstrings. Add your method to the list at the top of this docstring. Also, add your method to `ode.rst` in the `docs/src` directory, so that the Sphinx docs will pull its docstring into the main SymPy documentation. Be sure to make the Sphinx documentation by running `make html` from within the doc directory to verify that the docstring formats correctly.

If your solution method involves integrating, use *Integral* (page 601) instead of *integrate()* (page 964). This allows the user to bypass hard/slow integration by using the `_Integral` variant of your hint. In most cases, calling *sympy.core.basic.Basic.doit()* (page 932) will integrate your solution. If this is not the case, you will need to write special code in *_handle_Integral()* (page 826). Arbitrary constants should be symbols named `C1`, `C2`, and so on. All solution methods should return an equality instance. If you need an arbitrary number of arbitrary constants, you can use `constants = numbered_symbols(prefix='C', cls=Symbol, start=1)`. If it is possible to solve for the dependent function in a general way, do so. Otherwise, do as best as you can, but do not call solve in your `ode_<hint>()` function. *odesimp()* (page 767) will attempt to solve the solution for you, so you do not need to do that. Lastly, if your ODE has a common simplification that can be applied to your solutions, you can add a special case in *odesimp()* (page 767) for it. For example, solutions returned from the `1st_homogeneous_coeff` hints often have many *log* (page 413) terms, so *odesimp()* (page 767) calls *logcombine()* (page 669) on them (it also helps to write the arbitrary constant as `log(C1)` instead of `C1` in this case). Also consider common ways that you can rearrange your solution to have *constantsimp()* (page 765) take better advantage of it. It is better to put simplification in *odesimp()* (page 767) than in your method, because it can then be turned off with the simplify flag in *dsolve()* (page 755). If you have any extraneous simplification in your function, be sure to only run it using `if match.get('simplify', True):`, especially if it can be slow or if it can reduce the domain of the solution.

Finally, as with every contribution to SymPy, your method will need to be tested. Add a test for each method in `test_ode.py`. Follow the conventions there, i.e., test the solver using `dsolve(eq, f(x), hint=your_hint)`, and also test the solution using *checkodesol()* (page 762) (you can put these in a separate tests and skip/XFAIL if it runs too slow/does not work). Be sure to call your hint specifically in *dsolve()* (page 755), that way the test will not be broken simply by the introduction of another matching hint. If your method works for higher order (>1) ODEs, you will need to run `sol = constant_renumber(sol, 'C', 1, order)` for each solution, where `order` is the order of the ODE. This is because `constant_renumber` renumbers the arbitrary constants by printing order, which is platform dependent. Try to test every corner case of your solver, including a range of orders if it is a nth order solver, but if your solver is slow, such as if it involves hard integration, try to keep the test run time down.

Feel free to refactor existing hints to avoid duplicating code or creating inconsistencies. If you can show that your method exactly duplicates an existing method, including in the simplicity and speed of obtaining the solutions, then you can remove the old, less general method. The existing code is tested extensively in `test_ode.py`, so if anything is broken, one of those tests will surely fail.

Internal functions

These functions are not intended for end-user use.

`sympy.solvers.ode.ode.`**`_handle_Integral`**(*expr*, *func*, *hint*)

Converts a solution with Integrals in it into an actual solution.

For most hints, this simply runs `expr.doit()`.

PDE

User Functions

These are functions that are imported into the global namespace with `from sympy import *`. They are intended for user use.

pde_separate

`sympy.solvers.pde.`**`pde_separate`**(*eq*, *fun*, *sep*, *strategy='mul'*)

Separate variables in partial differential equation either by additive or multiplicative separation approach. It tries to rewrite an equation so that one of the specified variables occurs on a different side of the equation than the others.

Parameters

- **eq** – Partial differential equation
- **fun** – Original function F(x, y, z)
- **sep** – List of separated functions [X(x), u(y, z)]
- **strategy** – Separation strategy. You can choose between additive separation ('add') and multiplicative separation ('mul') which is default.

Examples

```
>>> from sympy import E, Eq, Function, pde_separate, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='add')
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

```
>>> eq = Eq(D(u(x, t), x, 2), D(u(x, t), t, 2))
>>> pde_separate(eq, u(x, t), [X(x), T(t)], strategy='mul')
[Derivative(X(x), (x, 2))/X(x), Derivative(T(t), (t, 2))/T(t)]
```

See also:

pde_separate_add (page 827), *pde_separate_mul* (page 827)

pde_separate_add

sympy.solvers.pde.**pde_separate_add**(*eq, fun, sep*)

Helper function for searching additive separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w, we look for the product of two functions depending on different arguments:

$w(x, y, z) = X(x) + y(y, z)$

Examples

```
>>> from sympy import E, Eq, Function, pde_separate_add, Derivative as D
>>> from sympy.abc import x, t
>>> u, X, T = map(Function, 'uXT')
```

```
>>> eq = Eq(D(u(x, t), x), E**(u(x, t))*D(u(x, t), t))
>>> pde_separate_add(eq, u(x, t), [X(x), T(t)])
[exp(-X(x))*Derivative(X(x), x), exp(T(t))*Derivative(T(t), t)]
```

pde_separate_mul

sympy.solvers.pde.**pde_separate_mul**(*eq, fun, sep*)

Helper function for searching multiplicative separable solutions.

Consider an equation of two independent variables x, y and a dependent variable w, we look for the product of two functions depending on different arguments:

$w(x, y, z) = X(x) * u(y, z)$

Examples

```
>>> from sympy import Function, Eq, pde_separate_mul, Derivative as D
>>> from sympy.abc import x, y
>>> u, X, Y = map(Function, 'uXY')
```

```
>>> eq = Eq(D(u(x, y), x, 2), D(u(x, y), y, 2))
>>> pde_separate_mul(eq, u(x, y), [X(x), Y(y)])
[Derivative(X(x), (x, 2))/X(x), Derivative(Y(y), (y, 2))/Y(y)]
```

pdsolve

sympy.solvers.pde.**pdsolve**(*eq*, *func=None*, *hint='default'*, *dict=False*, *solvefun=None*, ***kwargs*)

Solves any (supported) kind of partial differential equation.

Usage

pdsolve(eq, f(x,y), hint) -> Solve partial differential equation eq for function f(x,y), using method hint.

Details

eq can be any supported partial differential equation (see
the pde docstring for supported methods). This can either be an Equality, or an expression, which is assumed to be equal to 0.

f(x,y) is a function of two variables whose derivatives in that
variable make up the partial differential equation. In many cases it is not necessary to provide this; it will be autodetected (and an error raised if it could not be detected).

hint is the solving method that you want pdsolve to use. Use
classify_pde(eq, f(x,y)) to get all of the possible hints for a PDE. The default hint, 'default', will use whatever hint is returned first by classify_pde(). See Hints below for more options that you can use for hint.

solvefun is the convention used for arbitrary functions returned
by the PDE solver. If not set by the user, it is set by default to be F.

Hints

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

"default":
This uses whatever hint is returned first by classify_pde(). This is the default argument to pdsolve().

"all":
To make pdsolve apply all relevant classification hints, use pdsolve(PDE, func, hint="all"). This will return a dictionary of hint:solution terms. If a hint causes pdsolve to raise the NotImplementedError, value of that hint's key will be the exception object raised. The dictionary will also include some special keys:

- order: The order of the PDE. See also ode_order() in deutils.py
- default: The solution that would be returned by default. This is the one produced by the hint that appears first in the tuple returned by classify_pde().

"all_Integral":
This is the same as "all", except if a hint also has a corresponding "_Integral" hint, it only returns the "_Integral" hint. This is useful if "all" causes pdsolve() to hang because of a difficult or impossible integral. This meta-hint will also be much faster than "all", because integrate() is an expensive routine.

See also the classify_pde() docstring for more info on hints, and the pde docstring for a list of all supported hints.

Tips

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

```
>>> from sympy import Function, Derivative
>>> from sympy.abc import x, y # x and y are the independent␣
↪variables
>>> f = Function("f")(x, y) # f is a function of x and y
>>> # fx will be the partial derivative of f with respect to x
>>> fx = Derivative(f, x)
>>> # fy will be the partial derivative of f with respect to y
>>> fy = Derivative(f, y)
```

- See test_pde.py for many tests, which serves also as a set of examples for how to use pdsolve().
- pdsolve always returns an Equality class (except for the case when the hint is "all" or "all_Integral"). Note that it is not possible to get an explicit solution for f(x, y) as in the case of ODE's
- Do help(pde.pde_hintname) to get help more information on a specific hint

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> pdsolve(eq)
Eq(f(x, y), F(3*x - 2*y)*exp(-2*x/13 - 3*y/13))
```

classify_pde

`sympy.solvers.pde.`**`classify_pde`**(*eq, func=None, dict=False, *, prep=True, **kwargs*)

Returns a tuple of possible pdsolve() classifications for a PDE.

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

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

You can get help on different hints by doing help(pde.pde_hintname), where hintname is the name of the hint without "_Integral".

See sympy.pde.allhints or the sympy.pde docstring for a list of all supported hints that can be returned from classify_pde.

Examples

```
>>> from sympy.solvers.pde import classify_pde
>>> from sympy import Function, Eq
>>> from sympy.abc import x, y
>>> f = Function('f')
>>> u = f(x, y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> eq = Eq(1 + (2*(ux/u)) + (3*(uy/u)), 0)
>>> classify_pde(eq)
('1st_linear_constant_coeff_homogeneous',)
```

checkpdesol

sympy.solvers.pde.**checkpdesol**(*pde, sol, func=None, solve_for_func=True*)

Checks if the given solution satisfies the partial differential equation.

pde is the partial differential equation which can be given in the form of an equation or an expression. sol is the solution for which the pde is to be checked. This can also be given in an equation or an expression form. If the function is not provided, the helper function _preprocess from deutils is used to identify the function.

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

The following methods are currently being implemented to check if the solution satisfies the PDE:

1. Directly substitute the solution in the PDE and check. If the solution has not been solved for f, then it will solve for f provided solve_for_func has not been set to False.

If the solution satisfies the PDE, then a tuple (True, 0) is returned. Otherwise a tuple (False, expr) where expr is the value obtained after substituting the solution in the PDE. However if a known solution returns False, it may be due to the inability of doit() to simplify it to zero.

Examples

```
>>> from sympy import Function, symbols
>>> from sympy.solvers.pde import checkpdesol, pdsolve
>>> x, y = symbols('x y')
>>> f = Function('f')
>>> eq = 2*f(x,y) + 3*f(x,y).diff(x) + 4*f(x,y).diff(y)
>>> sol = pdsolve(eq)
>>> assert checkpdesol(eq, sol)[0]
>>> eq = x*f(x,y) + f(x,y).diff(x)
>>> checkpdesol(eq, sol)
(False, (x*F(4*x - 3*y) - 6*F(4*x - 3*y)/25 + 4*Subs(Derivative(F(_xi_1),
↪ _xi_1), _xi_1, 4*x - 3*y))*exp(-6*x/25 - 8*y/25))
```

Hint Methods

These functions are meant for internal use. However they contain useful information on the various solving methods.

pde_1st_linear_constant_coeff_homogeneous

sympy.solvers.pde.**pde_1st_linear_constant_coeff_homogeneous**(*eq, func, order, match, solvefun*)

Solves a first order linear homogeneous partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a\frac{\partial f(x,y)}{\partial x} + b\frac{\partial f(x,y)}{\partial y} + cf(x,y) = 0$$

where a, b and c are constants.

The general solution is of the form:

$$f(x,y) = F(-ay + bx)e^{-\frac{c(ax+by)}{a^2+b^2}}$$

and can be found in SymPy with pdsolve:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u
>>> pprint(genform)
  d               d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y)
  dx              dy

>>> pprint(pdsolve(genform))
                         -c*(a*x + b*y)
                         ---------------
                              2    2
                             a  + b
f(x, y) = F(-a*y + b*x)*e
```

Examples

```
>>> from sympy import pdsolve
>>> from sympy import Function, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y))
Eq(f(x, y), F(x - y)*exp(-x/2 - y/2))
>>> pprint(pdsolve(f(x,y) + f(x,y).diff(x) + f(x,y).diff(y)))
                      x   y
                    - - - -
                      2   2
f(x, y) = F(x - y)*e
```

References

- Viktor Grigoryan, "Partial Differential Equations" Math 124A - Fall 2010, pp.7

pde_1st_linear_constant_coeff

sympy.solvers.pde.**pde_1st_linear_constant_coeff**(*eq*, *func*, *order*, *match*, *solvefun*)

Solves a first order linear partial differential equation with constant coefficients.

The general form of this partial differential equation is

$$a\frac{\partial f(x,y)}{\partial x} + b\frac{\partial f(x,y)}{\partial y} + cf(x,y) = G(x,y)$$

where a, b and c are constants and $G(x,y)$ can be an arbitrary function in x and y.

The general solution of the PDE is:

$$f(x,y) = \left.\left[F(\eta) + \frac{1}{a^2+b^2}\int\limits^{ax+by} G\left(\frac{a\xi+b\eta}{a^2+b^2}, \frac{-a\eta+b\xi}{a^2+b^2}\right)e^{\frac{c\xi}{a^2+b^2}}\,d\xi\right]e^{-\frac{c\xi}{a^2+b^2}}\right|_{\substack{\eta=-ay+bx\\ \xi=ax+by}},$$

where $F(\eta)$ is an arbitrary single-valued function. The solution can be found in SymPy with pdsolve:

```
>>> from sympy.solvers import pdsolve
>>> from sympy.abc import x, y, a, b, c
>>> from sympy import Function, pprint
>>> f = Function('f')
>>> G = Function('G')
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a*ux + b*uy + c*u - G(x,y)
>>> pprint(genform)
  d               d
a*--(f(x, y)) + b*--(f(x, y)) + c*f(x, y) - G(x, y)
```

(continues on next page)

(continued from previous page)

```
  dx               dy
>>> pprint(pdsolve(genform, hint='1st_linear_constant_coeff_Integral'))
          //         a*x + b*y                                          ␣
 ↪   \
          ||              /                                             ␣
 ↪   |
          ||             |                                              ␣
 ↪   |
          ||             |                                       c*xi   ␣
 ↪   |
          ||             |                                     -------  ␣
 ↪   |
          ||             |                                      2    2  ␣
 ↪   |
          ||             |      /a*xi + b*eta  -a*eta + b*xi\  a  + b   ␣
 ↪   |
          ||             |     G|------------, -------------|*e       ␣
 ↪d(xi)|
          ||             |      |   2    2          2    2   |          ␣
 ↪   |
          ||             |      \  a  + b          a  + b    /          ␣
 ↪   |
          ||             |                                              ␣
 ↪   |
          ||            /                                               ␣
 ↪   |
          ||                                                            ␣
 ↪   |
f(x, y) = ||F(eta) + ----------------------------------------------------
 ↪---|*
          ||                              2    2                        ␣
 ↪   |
          \\                             a  + b                         ␣
 ↪   /

        \|
        ||
        ||
        ||
        ||
        ||
        ||
        ||
        ||
  -c*xi ||
 -------||
  2    2||
 a  + b ||
e       ||
        ||
        /|eta=-a*y + b*x, xi=a*x + b*y
```

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint, exp
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq = -2*f(x,y).diff(x) + 4*f(x,y).diff(y) + 5*f(x,y) - exp(x + 3*y)
>>> pdsolve(eq)
Eq(f(x, y), (F(4*x + 2*y)*exp(x/2) + exp(x + 4*y)/15)*exp(-y))
```

References

- Viktor Grigoryan, "Partial Differential Equations" Math 124A - Fall 2010, pp.7

pde_1st_linear_variable_coeff

sympy.solvers.pde.**pde_1st_linear_variable_coeff**(*eq*, *func*, *order*, *match*, *solvefun*)

Solves a first order linear partial differential equation with variable coefficients. The general form of this partial differential equation is

$$a(x,y)\frac{\partial f(x,y)}{\partial x} + b(x,y)\frac{\partial f(x,y)}{\partial y} + c(x,y)f(x,y) = G(x,y)$$

where $a(x,y)$, $b(x,y)$, $c(x,y)$ and $G(x,y)$ are arbitrary functions in x and y. This PDE is converted into an ODE by making the following transformation:

1. ξ as x
2. η as the constant in the solution to the differential equation $\frac{dy}{dx} = -\frac{b}{a}$

Making the previous substitutions reduces it to the linear ODE

$$a(\xi,\eta)\frac{du}{d\xi} + c(\xi,\eta)u - G(\xi,\eta) = 0$$

which can be solved using `dsolve`.

```
>>> from sympy.abc import x, y
>>> from sympy import Function, pprint
>>> a, b, c, G, f= [Function(i) for i in ['a', 'b', 'c', 'G', 'f']]
>>> u = f(x,y)
>>> ux = u.diff(x)
>>> uy = u.diff(y)
>>> genform = a(x, y)*u + b(x, y)*ux + c(x, y)*uy - G(x,y)
>>> pprint(genform)
                                             d                     d
-G(x, y) + a(x, y)*f(x, y) + b(x, y)*--(f(x, y)) + c(x, y)*--(f(x, y))
                                             dx                    dy
```

Examples

```
>>> from sympy.solvers.pde import pdsolve
>>> from sympy import Function, pprint
>>> from sympy.abc import x,y
>>> f = Function('f')
>>> eq =  x*(u.diff(x)) - y*(u.diff(y)) + y**2*u - y**2
>>> pdsolve(eq)
Eq(f(x, y), F(x*y)*exp(y**2/2) + 1)
```

References

- Viktor Grigoryan, "Partial Differential Equations" Math 124A - Fall 2010, pp.7

Information on the pde module

This module contains pdsolve() and different helper functions that it uses. It is heavily inspired by the ode module and hence the basic infrastructure remains the same.

Functions in this module

These are the user functions in this module:

- pdsolve() - Solves PDE's
- classify_pde() - Classifies PDEs into possible hints for dsolve().
- **pde_separate() - Separate variables in partial differential equation either by**
  additive or multiplicative separation approach.

These are the helper functions in this module:

- pde_separate_add() - Helper function for searching additive separable solutions.
- **pde_separate_mul() - Helper function for searching multiplicative**
  separable solutions.

Currently implemented solver methods

The following methods are implemented for solving partial differential equations. See the docstrings of the various pde_hint() functions for more information on each (run help(pde)):

- 1st order linear homogeneous partial differential equations with constant coefficients.
- 1st order linear general partial differential equations with constant coefficients.
- 1st order linear partial differential equations with variable coefficients.

Solvers

The *solvers* module in SymPy implements methods for solving equations.

Note: *solve()* (page 836) is an older more mature general function for solving many types of equations. *solve()* (page 836) has many options and uses different methods internally to determine what type of equations you pass it, so if you know what type of equation you are dealing with you may want to use the newer *solveset()* (page 858) which solves univariate equations, *linsolve()* (page 872) which solves system of linear equations, and *nonlinsolve()* (page 875) which solves systems of non linear equations.

Algebraic equations

Use *solve()* (page 836) to solve algebraic equations. We suppose all equations are equaled to 0, so solving x**2 == 1 translates into the following code:

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

The first argument for *solve()* (page 836) is an equation (equaled to zero) and the second argument is the symbol that we want to solve the equation for.

`sympy.solvers.solvers.`**`solve`**`(`*f, *symbols, **flags*`)`

Algebraically solves equations and systems of equations.

Parameters

f :

- a single Expr or Poly that must be zero
- an Equality
- a Relational expression
- a Boolean
- iterable of one or more of the above

symbols : (object(s) to solve for) specified as

- none given (other non-numeric objects will be used)
- single symbol
- denested list of symbols (e.g., `solve(f, x, y)`)
- ordered iterable of symbols (e.g., `solve(f, [x, y])`)

flags :

dict=True (default is False)
Return list (perhaps empty) of solution mappings.

set=True (default is False)
Return list of symbols and set of tuple(s) of solution(s).