Solving ordinary differential equations and Taylor expansion

ebrahimi

December 16, 2015

Solving ordinary differential equations

Solutions from the Maxima package can contain the three constants _C, _K1 , and _K2 where the underscore is used to distinguish them from symbolic variables that the user might have used. You can substitute values for them, and make them into accessible usable symbolic variables, for example withvar("_C").

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- desolve_rk4 - Solve numerically IVP for one first order equation, return list of points or plot.

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- desolve_rk4 - Solve numerically IVP for one first order equation, return list of points or plot.
- eulers_method - Approximate solution to a 1st order DE, presented as a table.

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- desolve_rk4 - Solve numerically IVP for one first order equation, return list of points or plot.
- eulers_method - Approximate solution to a 1st order DE, presented as a table.
- desolve_system - Solve any size system of 1st order odes using Maxima. Initial conditions are optional

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- desolve_rk4 - Solve numerically IVP for one first order equation, return list of points or plot.
- eulers_method - Approximate solution to a 1st order DE, presented as a table.
- desolve_system - Solve any size system of 1st order odes using Maxima. Initial conditions are optional
- desolve_system_rk4 - Solve numerically IVP for system of first order equations, return list of points.

Commands:

- desolve - Compute the general solution to a 1st or 2nd order ODE via Maxima
- desolve_laplace - Solve an ODE using Laplace transforms via Maxima. Initial conditions are optional
- desolve_rk4 - Solve numerically IVP for one first order equation, return list of points or plot.
- eulers_method - Approximate solution to a 1st order DE, presented as a table.
- desolve_system - Solve any size system of 1st order odes using Maxima. Initial conditions are optional
- desolve_system_rk4 - Solve numerically IVP for system of first order equations, return list of points.
- eulers_method_2x2 - Approximate solution to a 1st order system of DEs, presented as a table.

sage.calculus.desolvers.desolve(de, dvar, ics=None, ivar=None, show_method=False,contrib_ode=False)

Solves a 1st or 2nd order linear ODE via maxima. Including IVP and BVP.

INPUT:

- `de` - an expression or equation representing the ODE
- `dvar` - the dependent variable (hereafter called `y`)
- `ics` - (optional) the initial or boundary conditions
    - for a first-order equation, specify the initial `x` and `y`
    - for a second-order equation, specify the initial `x`, `y`, and `dy/dx`, i.e. write $[x_0, y(x_0), y'(x_0)]$
    - for a second-order boundary solution, specify initial and final `x` and `y` boundary conditions, i.e. write $[x_0, y(x_0), x_1, y(x_1)]$.
    - gives an error if the solution is not SymbolicEquation (as happens for example for a Clairaut equation)
- `ivar` - (optional) the independent variable (hereafter called x), which must be specified if there is more than one independent variable in the equation.

- `show_method` - (optional) if true, then Sage returns pair `[solution, method]`, where method is the string describing the metho
- d which has been used to get a solution (Maxima uses the following order for first order equations: linear, separable, exact (including exact with integrating factor), homogeneous, bernoulli, generalized homogeneous) - use carefully in class, see below for the example of the equation which is separable but this property is not recognized by Maxima and the equation is solved as exact.
- `contrib_ode` - (optional) if true, desolve allows to solve Clairaut, Lagrange, Riccati and some other equations. This may take a long time and is thus turned off by default. Initial conditions can be used only if the result is one SymbolicEquation (does not contain a singular solution, for example)

sage: $x = var('x')$

sage: $y = function('y', x)$

```
sage: f = desolve(diff(y,x) + y - 1, y, ics=[10,2]); f
(e^10 + e^x)*e^(-x)
```

```
sage: de = diff(y,x,2) - y == x
sage: desolve(de, y)
_K2*e^(-x) + _K1*e^x - x
sage: f = desolve(de, y, [10,2,1]); f
-x + 7*e^(x - 10) + 5*e^(-x + 10)
```

```
sage: desolve(diff(y,x)^2+x*diff(y,x)-y==0,y,contrib_ode=True,show_method=True)
 [[y(x) == _C^2 + _C*x, y(x) == -1/4*x^2], 'clairault']
```

```
sage: de = diff(y,x,2) + y == 0
sage: desolve(de, y)
_K2*cos(x) + _K1*sin(x)
sage: desolve(de, y, [0,1,pi/2,4])
cos(x) + 4*sin(x)
sage: desolve(y*diff(y,x)+sin(x)==0,y)
-1/2*y(x)^2 == _C - cos(x)
```

```
sage: desolve(diff(y,x)*sin(y) == cos(x),y)
-cos(y(x)) == _C + sin(x)
sage: desolve(diff(y,x)*sin(y) == cos(x),y,show_method=True)
[-cos(y(x)) == _C + sin(x), 'separable']
sage: desolve(diff(y,x)*sin(y) == cos(x),y,[pi/2,1])
-cos(y(x)) == -cos(1) + sin(x) - 1
```

```
sage: a,b,c,n=var('a b c n')
sage:
desolve(x^2*diff(y,x)==a+b*x^n+c*x^2*y^2,y,ivar=x,contrib_ode=True)
[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]]
sage:
desolve(x^2*diff(y,x)==a+b*x^n+c*x^2*y^2,y,ivar=x,contrib_ode=True,show
_method=True)
[[[y(x) == 0, (b*x^(n - 2) + a/x^2)*c^2*u == 0]], 'riccati']
```

```
sage: desolve(diff(y,x)+(y) == cos(x),y)
1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x)
sage: desolve(diff(y,x)+(y) == cos(x),y,show_method=True)
[1/2*((cos(x) + sin(x))*e^x + 2*_C)*e^(-x), 'linear']
sage: desolve(diff(y,x)+(y) == cos(x),y,[0,1])
1/2*(cos(x)*e^x + e^x*sin(x) + 1)*e^(-x)
```

```
sage: desolve(x^2*diff(y,x,x)+x*diff(y,x)+(x^2-4)*y==0,y)
_K1*bessel_J(2, x) + _K2*bessel_Y(2, x)
```

`sage.calculus.desolvers.`**`desolve_laplace`**(*de*, *dvar*, *ics=None*, *ivar=None*)

Solve an ODE using Laplace transforms. Initial conditions are optional.

INPUT:

- `de` - a lambda expression representing the ODE (eg, de = diff(y,x,2) == diff(y,x)+sin(x))
- `dvar` - the dependent variable (eg y)
- `ivar` - (optional) the independent variable (hereafter called x), which must be specified if there is more than one independent variable in the equation.
- `ics` - a list of numbers representing initial conditions, (eg, f(0)=1, f'(0)=2 is ics = [0,1,2])

OUTPUT:

Solution of the ODE as symbolic expression

```
sage: u=function('u',x)
sage: eq = diff(u,x) - exp(-x) - u == 0
sage: desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*e^(-x)
```

```
sage: desolve_laplace(eq,u,ics=[0,3])
-1/2*e^(-x) + 7/2*e^x
```

```
sage: f=function('f', x)
sage: eq = diff(f,x) + f == 0
sage: desolve_laplace(eq,f,[0,1])
e^(-x)
sage: x = var('x')
sage: f = function('f', x)
sage: de = diff(f,x,x) - 2*diff(f,x) + f
sage: desolve_laplace(de,f)
-x*e^x*f(0) + x*e^x*D[0](f)(0) + e^x*f(0)
sage: desolve_laplace(de,f,ics=[0,1,2])
x*e^x + e^x
```

sage.calculus.desolvers.**desolve_rk4**(*de*, *dvar*, *ics=None*, *ivar=None*, *end_points=None*, *step=0.1*,*output='list'*, ***kwds*)

Solve numerically one first-order ordinary differential equation. See also `ode_solver`.

INPUT:

input is similar to `desolve` command. The differential equation can be written in a form close to the plot_slope_field or desolve command

- Variant 1 (function in two variables)
  - `de` - right hand side, i.e. the function $f(x,y)$ from ODE $y'=f(x,y)$
  - `dvar` - dependent variable (symbolic variable declared by var)
- Variant 2 (symbolic equation)
  - `de` - equation, including term with `diff(y,x)`
  - `dvar` - dependent variable (declared as function of independent variable)

- Other parameters
  - `ivar` - should be specified, if there are more variables or if the equation is autonomous
  - `ics` - initial conditions in the form [x0,y0]
  - `end_points` - the end points of the interval
    - if end_points is a or [a], we integrate on between min(ics[0],a) and max(ics[0],a)
    - if end_points is None, we use end_points=ics[0]+10
    - if end_points is [a,b] we integrate on between min(ics[0],a) and max(ics[0],b)
  - `step` - (optional, default:0.1) the length of the step (positive number)
  - `output` - (optional, default: 'list') one of 'list', 'plot', 'slope_field' (graph of the solution with slope field)

OUTPUT:

Return a list of points, or plot produced by list_plot, optionally with slope field.

```
sage: x,y=var('x y')
sage: desolve_rk4(x*y*(2-y),y,ics=[0,1],end_points=1,step=0.5)
[[0, 1], [0.5, 1.12419127424558], [1.0, 1.46159016228825]]
```

```
sage: y=function('y',x)
sage:    desolve_rk4(diff(y,x)+y*(y-1)    ==    x-2,y,ics=[1,1],step=0.5,
end_points=0)
[[0.0, 8.904257108962112], [0.5, 1.909327945361535], [1, 1]]
```

`sage.calculus.desolvers.`**`desolve_rk4_determine_bounds`**(*ics*, *end_points=None*)

Used to determine bounds for numerical integration.

- If end_points is None, the interval for integration is from ics[0] to ics[0]+10
- If end_points is a or [a], the interval for integration is from min(ics[0],a) to max(ics[0],a)

- If end_points is [a,b], the interval for integration is from min(ics[0],a) to max(ics[0],b)

```
sage: from sage.calculus.desolvers import desolve_rk4_determine_bounds
sage: desolve_rk4_determine_bounds([0,2],1)
(0, 1)
sage: desolve_rk4_determine_bounds([0,2])
(0, 10)
sage: desolve_rk4_determine_bounds([0,2],[-2])
(-2, 0)
sage: desolve_rk4_determine_bounds([0,2],[-2,4])
(-2, 4)
```

`sage.calculus.desolvers.`**`eulers_method`**(*f*, *x0*, *y0*, *h*, *x1*, *algorithm='table'*)

This implements Euler's method for finding numerically the solution of the 1st order ODE `y' = f(x,y)`,`y(a)=c`. The "x" column of the table increments from `x0` to `x1` by `h` (so `(x1-x0)/h` must be an integer). In the "y" column, the new y-value equals the old y-value plus the corresponding entry in the last column.

For pedagogical purposes only.

```
sage: x,y = PolynomialRing(QQ,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1)
     x                    y                  h*f(x,y)
     0                    1                     -2
   1/2                   -1                   -7/4
     1                -11/4                  -11/8
sage: x,y = PolynomialRing(QQ,2,"xy").gens()
sage: eulers_method(5*x+y-5,0,1,1/2,1,algorithm="none")
[[0, 1], [1/2, -1], [1, -11/4], [3/2, -33/8]]
```

`sage.calculus.desolvers.`**`desolve_system`**(*des*, *vars*, *ics=None*, *ivar=None*)

Solve any size system of 1st order ODE's. Initial conditions are optional.

Onedimensional systems are passed to **`desolve_laplace()`**.

INPUT:

- `des` - list of ODEs
- `vars` - list of dependent variables
- `ics` - (optional) list of initial values for ivar and vars. If ics is defined, it should provide initial conditions for each variable, otherwise an exception would be raised.
- `ivar` - (optional) the independent variable, which must be specified if there is more than one independent variable in the equation.

```
sage: t = var('t')
sage: x = function('x', t)
sage: y = function('y', t)
sage: de1 = diff(x,t) + y - 1 == 0
sage: de2 = diff(y,t) - x + 1 == 0
sage: desolve_system([de1, de2], [x,y])
[x(t) == (x(0) - 1)*cos(t) - (y(0) - 1)*sin(t) + 1,
 y(t) == (y(0) - 1)*cos(t) + (x(0) - 1)*sin(t) + 1]
```

```
sage: sol = desolve_system([de1, de2], [x,y], ics=[0,1,2]); sol
[x(t) == -sin(t) + 1, y(t) == cos(t) + 1]
```

```
sage: t = var('t')
sage: x = function('x', t)
sage: y = function('y', t)
sage: de1 = diff(x,t) + y - 1 == 0
sage: de2 = diff(y,t) - x + 1 == 0
sage: des = [de1,de2]
sage: ics = [0,1,-1]
sage: vars = [x,y]
sage: sol = desolve_system(des, vars, ics); sol
[x(t) == 2*sin(t) + 1, y(t) == -2*cos(t) + 1]
```

sage.calculus.desolvers.**desolve_system_rk4**(*des*, *vars*, *ics=None*, *ivar=None*, *end_points=None*,*step=0.1*)

Solve numerically a system of first-order ordinary diffetrential equations using the 4th order Runge-Kutta method. Wrapper for Maxima command `rk`. See also `ode_solver`.

INPUT:

input is similar to desolve_system and desolve_rk4 commands

- `des` - right hand sides of the system
- `vars` - dependent variables
- `ivar` - (optional) should be specified, if there are more variables or if the equation is autonomous and the independent variable is missing
- `ics` - initial conditions in the form [x0,y01,y02,y03,....]

- `end_points` - the end points of the interval
    - if end_points is a or [a], we integrate on between min(ics[0],a) and max(ics[0],a)
    - if end_points is None, we use end_points=ics[0]+10
    - if end_points is [a,b] we integrate on between min(ics[0],a) and max(ics[0],b)
- `step` – (optional, default: 0.1) the length of the step

OUTPUT:

Return a list of points.

```
t=var('t')
x=function('x',t)
y=function('y',t)
t,x,y = PolynomialRing (QQ, 3, "txy").gens()
desolve_system_rk4([x*( 1-y), -y*( x-1)], [x,y], ics=[0,0.5,2], ivar=t, end_points=10,step=0.5)
```

```
[[0, 0.500000000000000, 2],
 [0.5, 0.2592749737668782, 2.731931223766878],
 [1.0, 0.08381181869088677, 4.159831105800261],
 [1.5, 0.03036496486737937, 6.749428008461739],
 [2.0, 0.1140108761050079, 11.18996636203008],
 [2.5, 3.726275194583882, 21.9842955659135],
 [3.0, -38058.46123938399, -38028.36403392813],
 [3.5, -2.39313067749154e+64, -2.39313067749154e+64]]
```

`sage.calculus.desolvers.`**`eulers_method_2x2`**(*f*, *g*, *t0*, *x0*, *y0*, *h*, *t1*, *algorithm='table'*)

This implements Euler's method for finding numerically the solution of the 1st order system of two ODEs

```
x' = f(t, x, y), x(t0)=x0.
```

```
y' = g(t, x, y), y(t0)=y0.
```

The "t" column of the table increments from t_0 to t_1 by h (so fract$_1$−t$_0$h must be an integer). In the "x" column, the new x-value equals the old x-value plus the corresponding entry in the next (third) column. In the "y" column, the new y-value equals the old y-value plus the corresponding entry in the next (last) column.

For pedagogical purposes only.

```
sage: t, x, y = PolynomialRing(QQ,3,"txy").gens()
sage: f = x+y+t; g = x-y
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1,algorithm="none")
[[0, 0, 0], [1/3, 0, 0], [2/3, 1/9, 0], [1, 10/27, 1/27], [4/3, 68/81,
4/27]]
sage: eulers_method_2x2(f,g, 0, 0, 0, 1/3, 1)
     t                    x                h*f(t,x,y)                    y
h*g(t,x,y)
     0                    0                         0                    0
0
   1/3                    0                       1/9                    0
0
   2/3                  1/9                      7/27                    0
1/27
     1                10/27                     38/81                 1/27
1/9
```

برای بسط تیلور ومکلورن توابع از دستور زیر استفاده می کنیم:

```
taylor(f,x,a,n)
```

که در آن

f: تعیین کننده تابع مورد نظر

x: بیانگر متغییر مستقل تابع

a: بیانگر نقطه ای که می خواهیم تابع حول آن بسط دهیم

n: بیانگر حداکثر درجه بسط می باشد.

برای به دست آوردن بسط مک لورن تابع کافیست به a مقدار صفر دهیم.

```
show(taylor(e^x,x,2,5))
```

$$\frac{1}{120}\left(x-2\right)^{5}e^{2}+\frac{1}{24}\left(x-2\right)^{4}e^{2}+\frac{1}{6}\left(x-2\right)^{3}e^{2}+\frac{1}{2}\left(x-2\right)^{2}e^{2}+(x-2)e^{2}+e^{2}$$

```
show(taylor(sin(x),x,0,8))
```

$$-\frac{1}{5040}\,x^{7}+\frac{1}{120}\,x^{5}-\frac{1}{6}\,x^{3}+x$$