```
sage: maxima = sage.calculus.calculus.maxima
sage: maxima.set('domain', 'real') # set domain to real
sage: f.simplify()
0
sage: maxima.set('domain', 'complex') # set domain back to its default value
sage: forget()

sage: # (YES) Transform equations, (x==2)/2+(1==1)=>x/2+1==2.
sage: eq1 = x == 2
sage: eq2 = SR(1) == SR(1)
sage: eq1/2 + eq2
1/2*x + 1 == 2

sage: # (SOMEWHAT) Solve Exp(x)=1 and get all solutions.
sage: # to_poly_solve in Maxima can do this.
sage: solve(exp(x) == 1, x)
[x == 0]

sage: # (SOMEWHAT) Solve Tan(x)=1 and get all solutions.
sage: # to_poly_solve in Maxima can do this.
sage: solve(tan(x) == 1, x)
[x == 1/4*pi]

sage: # (YES) Solve a degenerate 3x3 linear system.
sage: # x+y+z==6,2*x+y+2*z==10,x+3*y+z==10
sage: # First symbolically:
sage: solve([x+y+z==6, 2*x+y+2*z==10, x+3*y+z==10], x,y,z)
[[x == -r1 + 4, y == 2, z == r1]]

sage: # (YES) Invert a 2x2 symbolic matrix.
sage: # [[a,b],[1,a*b]]
sage: # Using multivariate poly ring -- much nicer
sage: R.<a,b> = QQ[]
sage: m = matrix(2,2,[a,b,  1, a*b])
sage: zz = m^(-1)
sage: print zz
[     a/(a^2 - 1)    (-1)/(a^2 - 1)]
[(-1)/(a^2*b - b)     a/(a^2*b - b)]

sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b, c, d.
sage: var('a,b,c,d')
(a, b, c, d)
sage: m = matrix(SR, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: print m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: d.factor()
(a - b)*(a - c)*(a - d)*(b - c)*(b - d)*(c - d)

sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b, c, d.
sage: # Do it instead in a multivariate ring
sage: R.<a,b,c,d> = QQ[]
sage: m = matrix(R, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
```

```
sage: print m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: print d
a^3*b^2*c - a^2*b^3*c - a^3*b*c^2 + a*b^3*c^2 + a^2*b*c^3 - a*b^2*c^3 - a^3*b^2*d + a^2*b^3*d + a^3*c
sage: print d.factor()
(-1) * (c - d) * (-b + c) * (b - d) * (-a + c) * (-a + b) * (a - d)
```

```
sage: # (YES) Find the eigenvalues of a 3x3 integer matrix.
sage: m = matrix(QQ, 3, [5,-3,-7, -2,1,2, 2,-3,-4])
sage: m.eigenspaces_left()
[
(3, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1  0 -1]),
(1, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1  1 -1]),
(-2, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[0 1 1])
]
```

```
sage: # (YES) Verify some standard limits found by L'Hopital's rule:
sage: #   Verify(Limit(x,Infinity) (1+1/x)^x, Exp(1));
sage: #   Verify(Limit(x,0) (1-Cos(x))/x^2, 1/2);
sage: limit( (1+1/x)^x, x = oo)
e
sage: limit( (1-cos(x))/(x^2), x = 1/2)
-4*cos(1/2) + 4
```

```
sage: # (OK-ish) D(x)Abs(x)
sage: #    Verify(D(x) Abs(x), Sign(x));
sage: diff(abs(x))
x/abs(x)
```

```
sage: # (YES) (Integrate(x)Abs(x))=Abs(x)*x/2
sage: integral(abs(x), x)
1/2*x*abs(x)
```

```
sage: #  (YES) Compute derivative of Abs(x), piecewise defined.
sage: #     Verify(D(x)if(x<0) (-x) else x,
sage: #        Simplify(if(x<0) -1 else 1))
Piecewise defined function with 2 parts, [[(-10, 0), -1], [(0, 10), 1]]
sage: #  (NOT really) Integrate Abs(x), piecewise defined.
sage: #      Verify(Simplify(Integrate(x)
sage: #        if(x<0) (-x) else x),
sage: #        Simplify(if(x<0) (-x^2/2) else x^2/2));
sage: f = piecewise([ [[-10,0], -x], [[0,10], x]])
sage: f.integral(definite=True)
100
```

```
sage: # (YES) Taylor series of 1/Sqrt(1-v^2/c^2) at v=0.
sage: var('v,c')
(v, c)
sage: taylor(1/sqrt(1-v^2/c^2), v, 0, 7)
1/2*v^2/c^2 + 3/8*v^4/c^4 + 5/16*v^6/c^6 + 1

sage: # (OK-ish) (Taylor expansion of Sin(x))/(Taylor expansion of Cos(x)) = (Taylor expansion of Ta
sage: #      TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
sage: #        (Taylor(x,0,5)Cos(x)), Taylor(x,0,5)Tan(x));
sage: f = taylor(sin(x), x, 0, 8)
sage: g = taylor(cos(x), x, 0, 8)
sage: h = taylor(tan(x), x, 0, 8)
sage: f = f.power_series(QQ)
sage: g = g.power_series(QQ)
sage: h = h.power_series(QQ)
sage: f - g*h
O(x^8)

sage: # (YES) Taylor expansion of Ln(x)^a*Exp(-b*x) at x=1.
sage: a,b = var('a,b')
sage: taylor(log(x)^a*exp(-b*x), x, 1, 3)
-1/48*(a^3*(x - 1)^a + a^2*(6*b + 5)*(x - 1)^a + 8*b^3*(x - 1)^a + 2*(6*b^2 + 5*b + 3)*a*(x - 1)^a)*

sage: # (YES) Taylor expansion of Ln(Sin(x)/x) at x=0.
sage: taylor(log(sin(x)/x), x, 0, 10)
-1/467775*x^10 - 1/37800*x^8 - 1/2835*x^6 - 1/180*x^4 - 1/6*x^2

sage: # (NO) Compute n-th term of the Taylor series of Ln(Sin(x)/x) at x=0.
sage: # need formal functions

sage: # (NO) Compute n-th term of the Taylor series of Exp(-x)*Sin(x) at x=0.
sage: # (Sort of, with some work)
sage: # Solve x=Sin(y)+Cos(y) for y as Taylor series in x at x=1.
sage: #      TestYacas(InverseTaylor(y,0,4) Sin(y)+Cos(y),
sage: #        (y-1)+(y-1)^2/2+2*(y-1)^3/3+(y-1)^4);
sage: #       Note that InverseTaylor does not give the series in terms of x but in terms of y which
sage: # wrong. But other CAS do the same.
sage: f = sin(y) + cos(y)
sage: g = f.taylor(y, 0, 10)
sage: h = g.power_series(QQ)
sage: k = (h - 1).reversion()
sage: print k
y + 1/2*y^2 + 2/3*y^3 + y^4 + 17/10*y^5 + 37/12*y^6 + 41/7*y^7 + 23/2*y^8 + 1667/72*y^9 + 3803/80*y^

sage: # (OK) Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/(2^n*n!) *(Deriv
sage: #      P(n,x) := Simplify( 1/(2*n)!! *
sage: #        Deriv(x,n) (x^2-1)^n );
sage: #      TestYacas(P(4,x), (35*x^4)/8+(-15*x^2)/4+3/8);
sage: P = lambda n, x: simplify(diff((x^2-1)^n,x,n) / (2^n * factorial(n)))
sage: P(4,x).expand()
35/8*x^4 - 15/4*x^2 + 3/8

sage: # (YES) Define the polynomial p=Sum(i,1,5,a[i]*x^i).
sage: # symbolically
sage: ps = sum(var('a%s'%i)*x^i for i in range(1,6)); ps
a5*x^5 + a4*x^4 + a3*x^3 + a2*x^2 + a1*x
```

```
sage: ps.parent()
Symbolic Ring
sage: # algebraically
sage: R = PolynomialRing(QQ,5,names='a')
sage: S.<x> = PolynomialRing(R)
sage: p = S(list(R.gens()))*x; p
a4*x^5 + a3*x^4 + a2*x^3 + a1*x^2 + a0*x
sage: p.parent()
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a0, a1, a2, a3, a4 over Rationa
```

```
sage: # (YES) Convert the above to Horner's form.
sage: #      Verify(Horner(p, x), ((((a[5]*x+a[4])*x
sage: #        +a[3])*x+a[2])*x+a[1])*x);
sage: # We use the trick of evaluating the algebraic poly at a symbolic variable:
sage: restore('x')
sage: p(x)
((((a4*x + a3)*x + a2)*x + a1)*x + a0)*x
```

```
sage: # (NO) Convert the result of problem 127 to Fortran syntax.
sage: #      CForm(Horner(p, x));
```

```
sage: # (YES) Verify that True And False=False.
sage: (True and False) == False
True
```

```
sage: # (YES) Prove x Or Not x.
sage: for x in [True, False]:
...     print x or (not x)
True
True
```

```
sage: # (YES) Prove x Or y Or x And y=>x Or y.
sage: for x in [True, False]:
...   for y in [True, False]:
...       if x or y or x and y:
...           if not (x or y):
...              print "failed!"
```

CHAPTER

SIXTEEN

SOLVING ORDINARY DIFFERENTIAL EQUATIONS

This file contains functions useful for solving differential equations which occur commonly in a 1st semester differential equations course. For another numerical solver see the `ode_solver()` function and the optional package Octave.

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 with `var("_C")`.

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.
- `desolve_system_rk4` - Solve numerically IVP for system of first order equations, return list of points.
- `desolve_odeint` - Solve numerically a system of first-order ordinary differential equations using `odeint` from scipy.integrate module.
- `desolve_system` - Solve any size system of 1st order odes using Maxima. Initial conditions are optional.
- `eulers_method` - Approximate solution to a 1st order DE, presented as a table.
- `eulers_method_2x2` - Approximate solution to a 1st order system of DEs, presented as a table.
- `eulers_method_2x2_plot` - Plot the sequence of points obtained from Euler's method.

AUTHORS:

- David Joyner (3-2006) - Initial version of functions
- Marshall Hampton (7-2007) - Creation of Python module and testing
- Robert Bradshaw (10-2008) - Some interface cleanup.
- Robert Marik (10-2009) - Some bugfixes and enhancements

`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.

Use `desolve?    <tab>` *if the output in truncated in notebook.*

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 method 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)

OUTPUT:

In most cases return a SymbolicEquation which defines the solution implicitly. If the result is in the form y(x)=... (happens for linear eqs.), return the right-hand side only. The possible constant solutions of separable ODE's are omitted.

EXAMPLES:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve(diff(y,x) + y - 1, y)
(_C + e^x)*e^(-x)

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

sage: plot(f)
```

We can also solve second-order differential equations.:

```
sage: x = var('x')
sage: y = function('y', 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: f(x=10)
2

sage: diff(f,x)(x=10)
1
```

```
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)
```

Clairaut equation: general and singular solutions:

```
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']
```

For equations involving more variables we specify an independent variable:

```
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']
```

Higher order equations, not involving independent variable:

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y).expand()
1/6*y(x)^3 + _K1*y(x) == _K2 + x

sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,1,3]).expand()
1/6*y(x)^3 - 5/3*y(x) == x - 3/2

sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,1,3],show_method=True)
[1/6*y(x)^3 - 5/3*y(x) == x - 3/2, 'freeofx']
```

Separable equations - Sage returns solution in implicit form:

```
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
```

Linear equation - Sage returns the expression on the right hand side only:

```
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)
```

This ODE with separated variables is solved as exact. Explanation - factor does not split e^{x-y} in Maxima into $e^x e^y$:

```
sage: desolve(diff(y,x)==exp(x-y),y,show_method=True)
[-e^x + e^y(x) == _C, 'exact']
```

You can solve Bessel equations, also using initial conditions, but you cannot put (sometimes desired) the initial condition at x=0, since this point is a singular point of the equation. Anyway, if the solution should be bounded at x=0, then _K2=0.:

```
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)
```

Example of difficult ODE producing an error:

```
sage: desolve(sqrt(y)*diff(y,x)+e^(y)+cos(x)-sin(x+y)==0,y) # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option contrib_ode to
```

Another difficult ODE with error - moreover, it takes a long time

```
sage: desolve(sqrt(y)*diff(y,x)+e^(y)+cos(x)-sin(x+y)==0,y,contrib_ode=True) # not tested
```

Some more types of ODE's:

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

sage: desolve(diff(y,x)==(x+y)^2,y,contrib_ode=True,show_method=True)
[[[x == _C - arctan(sqrt(t)), y(x) == -x - sqrt(t)], [x == _C + arctan(sqrt(t)), y(x) == -x + sq
```

These two examples produce an error (as expected, Maxima 5.18 cannot solve equations from initial conditions). Maxima 5.18 returns false answer in this case!:

```
sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,2]).expand() # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option contrib_ode to

sage: desolve(diff(y,x,2)+y*(diff(y,x,1))^3==0,y,[0,1,2],show_method=True) # not tested
Traceback (click to the left for traceback)
...
NotImplementedError, "Maxima was unable to solve this ODE. Consider to set option contrib_ode to
```

Second order linear ODE:

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y)
(_K2*x + _K1)*e^(-x) + 1/2*sin(x)

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,show_method=True)
[(_K2*x + _K1)*e^(-x) + 1/2*sin(x), 'variationofparameters']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,1])
1/2*(7*x + 6)*e^(-x) + 1/2*sin(x)

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,1],show_method=True)
[1/2*(7*x + 6)*e^(-x) + 1/2*sin(x), 'variationofparameters']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,pi/2,2])
3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x)
```

```
sage: desolve(diff(y,x,2)+2*diff(y,x)+y == cos(x),y,[0,3,pi/2,2],show_method=True)
[3*(x*(e^(1/2*pi) - 2)/pi + 1)*e^(-x) + 1/2*sin(x), 'variationofparameters']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y)
(_K2*x + _K1)*e^(-x)

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,show_method=True)
[(_K2*x + _K1)*e^(-x), 'constcoeff']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,1])
(4*x + 3)*e^(-x)

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,1],show_method=True)
[(4*x + 3)*e^(-x), 'constcoeff']

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,pi/2,2])
(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x)

sage: desolve(diff(y,x,2)+2*diff(y,x)+y == 0,y,[0,3,pi/2,2],show_method=True)
[(2*x*(2*e^(1/2*pi) - 3)/pi + 3)*e^(-x), 'constcoeff']
```

TESTS:

Trac #9961 fixed (allow assumptions on the dependent variable in desolve):

```
sage: y=function('y',x); assume(x>0); assume(y>0)
sage: sage.calculus.calculus.maxima('domain:real')  # needed since Maxima 5.26.0 to get the answ
real
sage: desolve(x*diff(y,x)-x*sqrt(y^2+x^2)-y == 0, y, contrib_ode=True)
[x - arcsinh(y(x)/x) == _C]
```

Trac #10682 updated Maxima to 5.26, and it started to show a different solution in the complex domain for the ODE above:

```
sage: sage.calculus.calculus.maxima('domain:complex')  # back to the default complex domain
complex
sage: desolve(x*diff(y,x)-x*sqrt(y^2+x^2)-y == 0, y, contrib_ode=True)
[1/2*(2*x^2*sqrt(x^(-2)) - 2*x*sqrt(x^(-2))*arcsinh(y(x)/sqrt(x^2)) -
    2*x*sqrt(x^(-2))*arcsinh(y(x)^2/(x*sqrt(y(x)^2))) +
    log(4*(2*x^2*sqrt((x^2*y(x)^2 + y(x)^4)/x^2)*sqrt(x^(-2)) + x^2 +
    2*y(x)^2)/x^2))/(x*sqrt(x^(-2))) == _C]
```

Trac #6479 fixed:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve( diff(y,x,x) == 0, y, [0,0,1])
x

sage: desolve( diff(y,x,x) == 0, y, [0,1,1])
x + 1
```

Trac #9835 fixed:

```
sage: x = var('x')
sage: y = function('y', x)
sage: desolve(diff(y,x,2)+y*(1-y^2)==0,y,[0,-1,1,1])
Traceback (most recent call last):
...
NotImplementedError: Unable to use initial condition for this equation (freeofx).
```

Trac #8931 fixed:

```
sage: x=var('x'); f=function('f',x); k=var('k'); assume(k>0)
sage: desolve(diff(f,x,2)/f==k,f,ivar=x)
_K1*e^(sqrt(k)*x) + _K2*e^(-sqrt(k)*x)
```

AUTHORS:

- David Joyner (1-2006)
- Robert Bradshaw (10-2008)
- Robert Marik (10-2009)

`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

EXAMPLES:

```
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)
```

We can use initial conditions:

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

The initial conditions do not persist in the system (as they persisted in previous versions):

```
sage: desolve_laplace(eq,u)
1/2*(2*u(0) + 1)*e^x - 1/2*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
```

TESTS:

Trac #4839 fixed:

```
sage: t=var('t')
sage: x=function('x', t)
sage: soln=desolve_laplace(diff(x,t)+x==1, x, ics=[0,2])
sage: soln
e^(-t) + 1

sage: soln(t=3)
e^(-3) + 1
```

AUTHORS:

- David Joyner (1-2006,8-2007)
- Robert Marik (10-2009)

`sage.calculus.desolvers.`**`desolve_odeint`**(*des*, *ics*, *times*, *dvars*, *ivar=None*, *compute_jac=False*, *args=()*, *rtol=None*, *atol=None*, *tcrit=None*, *h0=0.0*, *hmax=0.0*, *hmin=0.0*, *ixpr=0*, *mxstep=0*, *mxhnil=0*, *mxordn=12*, *mxords=5*, *printmessg=0*)

Solve numerically a system of first-order ordinary differential equations using `odeint` from scipy.integrate module.

INPUT:

- `des` – right hand sides of the system
- `ics` – initial conditions
- `times` – a sequence of time points in which the solution must be found
- `dvars` – dependent variables. ATTENTION: the order must be the same as in des, that means: d(dvars[i])/dt=des[i]
- `ivar` – independent variable, optional.
- `compute_jac` – boolean. If True, the Jacobian of des is computed and used during the integration of Stiff Systems. Default value is False.

Other Parameters (taken from the documentation of odeint function from scipy.integrate module)

- `rtol`, `atol` : float The input parameters rtol and atol determine the error control performed by the solver. The solver will control the vector, e, of estimated local errors in y, according to an inequality of the form:

  max-norm of (e / ewt) <= 1

  where ewt is a vector of positive error weights computed as:

  ewt = rtol * abs(y) + atol

  rtol and atol can be either vectors the same length as y or scalars.
- `tcrit` : array Vector of critical points (e.g. singularities) where integration care should be taken.
- `h0` : float, (0: solver-determined) The step size to be attempted on the first step.
- `hmax` : float, (0: solver-determined) The maximum absolute step size allowed.
- `hmin` : float, (0: solver-determined) The minimum absolute step size allowed.
- `ixpr` : boolean. Whether to generate extra printing at method switches.

- `mxstep` : integer, (0: solver-determined) Maximum number of (internally defined) steps allowed for each integration point in t.
- `mxhnil` : integer, (0: solver-determined) Maximum number of messages printed.
- `mxordn` : integer, (0: solver-determined) Maximum order to be allowed for the nonstiff (Adams) method.
- `mxords` : integer, (0: solver-determined) Maximum order to be allowed for the stiff (BDF) method.

OUTPUT:

Return a list with the solution of the system at each time in times.

EXAMPLES:

Lotka Volterra Equations:

```
sage: from sage.calculus.desolvers import desolve_odeint
sage: x,y=var('x,y')
sage: f=[x*(1-y),-y*(1-x)]
sage: sol=desolve_odeint(f,[0.5,2],srange(0,10,0.1),[x,y])
sage: p=line(zip(sol[:,0],sol[:,1]))
sage: p.show()
```

Lorenz Equations:

```
sage: x,y,z=var('x,y,z')
sage: # Next we define the parameters
sage: sigma=10
sage: rho=28
sage: beta=8/3
sage: # The Lorenz equations
sage: lorenz=[sigma*(y-x),x*(rho-z)-y,x*y-beta*z]
sage: # Time and initial conditions
sage: times=srange(0,50.05,0.05)
sage: ics=[0,1,1]
sage: sol=desolve_odeint(lorenz,ics,times,[x,y,z],rtol=1e-13,atol=1e-14)
```

One-dimensional Stiff system:

```
sage: y= var('y')
sage: epsilon=0.01
sage: f=y^2*(1-y)
sage: ic=epsilon
sage: t=srange(0,2/epsilon,1)
sage: sol=desolve_odeint(f,ic,t,y,rtol=1e-9,atol=1e-10,compute_jac=True)
sage: p=points(zip(t,sol))
sage: p.show()
```

Another Stiff system with some optional parameters with no default value:

```
sage: y1,y2,y3=var('y1,y2,y3')
sage: f1=77.27*(y2+y1*(1-8.375*1e-6*y1-y2))
sage: f2=1/77.27*(y3-(1+y1)*y2)
sage: f3=0.16*(y1-y3)
sage: f=[f1,f2,f3]
sage: ci=[0.2,0.4,0.7]
sage: t=srange(0,10,0.01)
sage: v=[y1,y2,y3]
sage: sol=desolve_odeint(f,ci,t,v,rtol=1e-3,atol=1e-4,h0=0.1,hmax=1,hmin=1e-4,mxstep=1000,mxords
```

AUTHOR:

•Oriol Castejon (05-2010)

`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 funciton 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.

EXAMPLES:

```
sage: from sage.calculus.desolvers import desolve_rk4
```

Variant 2 for input - more common in numerics:

```
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, 1.12419127425], [1.0, 1.46159016229]]
```

Variant 1 for input - we can pass ODE in the form used by desolve function In this example we integrate bakwards, since `end_points < ics[0]`:

```
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, 8.90425710896], [0.5, 1.90932794536], [1, 1]]
```

Here we show how to plot simple pictures. For more advanced aplications use list_plot instead. To see the resulting picture use `show(P)` in Sage notebook.

```
sage: x,y=var('x y')
sage: P=desolve_rk4(y*(2-y),y,ics=[0,.1],ivar=x,output='slope_field',end_points=[-4,6],thickness
```

ALGORITHM:

4th order Runge-Kutta method. Wrapper for command `rk` in Maxima's dynamics package. Perhaps could be faster by using fast_float instead.

AUTHORS:

- Robert Marik (10-2009)

`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)

EXAMPLES:

```
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.`**`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
- `ivar` - (optional) the independent variable, which must be specified if there is more than one independent variable in the equation.

EXAMPLES:

```
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]
```

Now we give some initial conditions:

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

sage: solnx, solny = sol[0].rhs(), sol[1].rhs()
sage: plot([solnx,solny],(0,1))  # not tested
sage: parametric_plot((solnx,solny),(0,1))  # not tested
```

TESTS:

Trac #9823 fixed:

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

AUTHORS:

- Robert Bradshaw (10-2008)

`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 differential 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.

EXAMPLES:

```
sage: from sage.calculus.desolvers import desolve_system_rk4
```

Lotka Volterra system:

```
sage: from sage.calculus.desolvers import desolve_system_rk4
sage: x,y,t=var('x y t')
sage: P=desolve_system_rk4([x*(1-y),-y*(1-x)],[x,y],ics=[0,0.5,2],ivar=t,end_points=20)
sage: Q=[ [i,j] for i,j,k in P]
sage: LP=list_plot(Q)
```