ODES

What is ODES

ODES - Ordinary Differential Equations

DAES - Differential Algebraic Equations

Uses

- Calculate movement or flow of electricity, motion of an object to and fro like a pendulum, explain thermodynamics concepts
- Root Finding, finding min and max of functions

About ODES

- Started in November 2008
- Originated from by bmcage
- Spicy extension
- Zenodo

Different Solvers

VCODE

- ODE solver with BDF linear multistep method for stiff problems and Adams-MMoulton linear multistep method for nonstiff problems. Supports modern features such as: root (event) finding, error control, and (Krylov-)preconditioning.

IDA

- ODE solver with BDF linear multistep method for stiff problems and Adams-MMoulton linear multistep method for nonstiff problems. Supports modern features such as: root (event) finding, error control, and (Krylov-)preconditioning.

Example

Planar Pendulum Example

This example shows how to solve the planar pendulum in full coordinate space. This results in a dae system with one algebraic equation.

The problem is easily stated: a pendulum must move on a circle with radius l, it has a mass m, and gravitational accelleration is g.

The Lagragian is $L = 1/2m(u^2 + v^2) - mgy$, with constraint: $x^2 + y^2 = l$. u is the speed $\dot{x}$ and v is $\dot{y}$.

Adding a Lagrange multiplier λ, we arrive at the Euler Lagrange differential equations for the problem:

$$\begin{aligned}
\dot{x} &= u \\
\dot{y} &= v \\
\dot{u} &= \lambda \tfrac{x}{m} \\
\dot{v} &= \lambda \tfrac{y}{m} - g
\end{aligned}$$

and λ must be such that the constraint is satisfied: $x^2 + y^2 = l$.

We next derive a different constraint that contains more of the unknowns, as well as λ. Derivation to time of the constraint gives a new constraint: $xu + yv = 0$.

Derivating a second time to time gives us:

$$u^2 + v^2 + x\dot{u} + y\dot{v} = 0$$

which can be written with the known form of $\dot{u}$, $\dot{v}$ as

$$u^2 + v^2 + \lambda \frac{l^2}{m} - gy = 0.$$

This last expression will be used to find the solution to the planar pendulum problem.

The algorithm first needs to find initial conditions for the derivatives, then it solves the problme at hand. We take $g = 1$, $m = 1$, $l = 1$.

Example

```python
from __future__ import print_function, division
import matplotlib.pyplot as plt
import numpy as np
from scikits.odes import dae
```

```python
#data of the pendulum
l = 1.0
m = 1.0
g = 1.0
#initial condition
theta0= np.pi/3 #starting angle
x0=np.sin(theta0)
y0=-(l-x0**2)**.5
lambdaval = 0.1
z0  = [x0, y0, 0., 0., lambdaval]
zp0 = [0., 0., lambdaval*x0/m, lambdaval*y0/m-g, -g]
```

We need a first order system cast into residual equations, so we convert the problem as such. This consists of 4 differential equations and one algebraic equation:

$$
\begin{aligned}
0 &= u - \dot{x} \\
0 &= v - \dot{y} \\
0 &= -\dot{u} + \lambda \frac{x}{m} \\
0 &= -\dot{v} + \lambda \frac{y}{m} - g \\
0 &= u^2 + v^2 + \lambda \frac{l^2}{m} - gy
\end{aligned}
$$

Example

You need to define a function that computes the right hand side of above equation:

```
def residual(t, x, xdot, result):
    """ we create the residual equations for the problem"""
    result[0] = x[2]-xdot[0]
    result[1] = x[3]-xdot[1]
    result[2] = -xdot[2]+x[4]*x[0]/m
    result[3] = -xdot[3]+x[4]*x[1]/m-g
    result[4] = x[2]**2 + x[3]**2 \
                    + (x[0]**2 + x[1]**2)/m*x[4] - x[1] * g
```

To solve the DAE you define a dae object, specify the solver to use, here ida, and pass the residual function. You request the solution at specific timepoints by passing an array of times to the solve member.

```
solver = dae('ida', residual,
             compute_initcond='yp0',
             first_step_size=1e-18,
             atol=1e-6,
             rtol=1e-6,
             algebraic_vars_idx=[4],
             compute_initcond_t0 = 60,
             old_api=False)
solution = solver.solve([0., 1., 2.], z0, zp0)
```

```
print('\n   t        Solution')
print('----------------------')
for t, u in zip(solution.values.t, solution.values.y):
    print('{0:>4.0f} {1:15.6g} '.format(t, u[0]))
```

```
   t        Solution
----------------------
   0        0.866025
   1        0.592663
   2       -0.304225
```

You can continue the solver by passing further times. Calling the solve routine reinits the solver, so you can restart at whatever time. To continue from the last computed solution, pass the last obtained time and solution.

Note: The solver performes better if it can take into account history information, so avoid calling solve to continue computation!

In general, you must check for errors using the errors output of solve.

Example

```python
#Solve over the next hour by continuation
times = np.linspace(0, 3600, 61)
times[0] = solution.values.t[-1]
solution = solver.solve(times, solution.values.y[-1], solution.values.ydot[-1])
if solution.errors.t:
    print ('Error: ', solution.message, 'Error at time', solution.errors.t)
print ('Computed Solutions:')
print('\n   t         Solution ')
print('-----------------------')
for t, u in zip(solution.values.t, solution.values.y):
    print('{0:>4.0f} {1:15.6g}'.format(t, u[0]))
```

```
Error:  Could not reach endpoint Error at time 15.7411967287
Computed Solutions:

   t         Solution
-----------------------
   2       -0.304225
```

The solution fails at a time around 15 seconds. Errors can be due to many things. Here however the reason is simple: we try to make too large jumps in time output. Increasing the allowed steps the solver can take will fix this. This is the **max_steps** option of ida:

```python
solver = dae('ida', residual,
             compute_initcond='yp0',
             first_step_size=1e-18,
             atol=1e-6,
             rtol=1e-6,
             algebraic_vars_idx=[4],
             compute_initcond_t0 = 60,
             old_api=False,
             max_steps=5000)
solution = solver.solve(times, solution.values.y[-1], solution.values.ydot[-1])
if solution.errors.t:
    print ('Error: ', solution.message, 'Error at time', solution.errors.t)
print ('Computed Solutions:')
print('\n   t         Solution')
print('----------------------')
for t, u in zip(solution.values.t, solution.values.y):
    print('{0:>4.0f} {1:15.6g} '.format(t, u[0]))
```

Computed Solutions:

t	Solution
2	-0.304225
60	0.748758
120	0.304114
180	-0.371785
240	-0.791746
300	-0.859838
360	-0.628585
420	0.0639543
480	0.7327
540	0.859553
600	0.346761
660	-0.681437
720	-0.827876
780	0.245997
840	0.874605
900	-0.403099
960	-0.69528
1020	0.895303
1080	-0.696907
1140	0.417578
1200	-0.335093
1260	0.513395
1320	-0.812889
1380	0.926238
1440	-0.405328
1500	-0.789546
1560	0.649514
1620	0.903153
1680	0.197514
1740	-0.540406
1800	-0.824258
1860	-0.885005
1920	-0.808702
1980	-0.440341
2040	0.406477
2100	0.9957
2160	0.349396
2220	-0.997113
2280	0.252089
2340	0.620979
2400	-0.925843
2460	0.975331
2520	-0.873387
2580	0.373025
2640	0.686289
2700	-0.989182
2760	-0.574528
2820	0.66487
2880	1.0753
2940	1.04512
3000	1.00029
3060	1.06947
3120	1.08483
3180	0.569069
3240	-0.715037
3300	-0.987832
3360	0.786597
3420	0.430898
3480	-1.06173
3540	1.14725
3600	-1.10472

Example

```python
#plot of the oscilator
solver =  dae('ida', residual,
              compute_initcond='yp0',
              first_step_size=1e-18,
              atol=1e-6,
              rtol=1e-6,
              algebraic_vars_idx=[4],
              old_api=False,
              max_steps=5000)
times = np.linspace(0,60,600)
solution = solver.solve(times, z0, zp0)
f, axs = plt.subplots(2,2,figsize=(15,7))
plt.subplot(1, 2, 1)
plt.plot(solution.values.t,[x[0] for x in solution.values.y])
plt.xlabel('Time [s]')
plt.ylabel('Position x [m]')
plt.subplot(1, 2, 2)
plt.plot(solution.values.t,[x[1] for x in solution.values.y])
plt.xlabel('Time [s]')
plt.ylabel('Position y [m]')
plt.show()
# plot in space
plt.axis('equal')
plt.plot([x[0] for x in solution.values.y],[x[1] for x in solution.values.y],)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```