ODES scikit

By Richard Terrile

What key problems are solved?

Package provides root-finding, preconditioning and error control to differential equation solvers in Python.

This software package is for Python, but requires a C compiler, Fortran compiler, and some Python libraries

Who are the stakeholders?

- Who are the developers? Who pays?
  - A variety of open source collaborators contribute to the software
- Who uses it?
  - Developed as a research tool. Currently being used in a 1D centrifuge simulation
- What are they looking for?
  - A better interface and more functionality in SciPy's differential equation solvers
- How do they communicate/collaborate with each other?
  - Through Github
- Who is impacted negatively?
  - No one, the software simply extends and enhances SciPy's capabilities
- Who is impacted positively?
  - Anyone who needs more control and a better interface with differential equation solvers in Python

Metrics and Features

- How do concepts like accuracy, conditioning, stability, and cost appear?
    - Accuracy, conditioning, and stability can be determined by the packages error control and conditioning functions. Cost is not represented in the project
- Is it fast?
    - Packages is generally as fast as normal SciPy solvers, which is can be defined as fast
- Is it accurate?
    - Testing error vs. time for different functions provides insight into the accuracy of each
- Cost?
    - Cost is not a consideration in this open source package
- Are there modeling decisions made in the interest of good-conditioning? Are there algorithmic choices for stability?
    - Conditioning of functions is up to the user, though by default it tries to increase stability

Example from ‘Simple Oscillator Example’

Simple Oscillator Example

This example shows the most simple way of using a solver. We solve free vibration of a simple oscillator:

$$m\ddot{u} + ku = 0, \quad u(0) = u_0, \quad \dot{u}(0) = \dot{u}_0$$

using the CVODE solver. An analytical solution exists, given by

$$u(t) = u_0 \cos\left(\sqrt{\frac{k}{m}}t\right) + \frac{\dot{u}_0}{\sqrt{\frac{k}{m}}} \sin\left(\sqrt{\frac{k}{m}}t\right)$$

```
#data of the oscillator
k = 4.0
m = 1.0
#initial position and speed data on t=0, x[0] = u, x[1] = \dot{u}, xp = \dot{x}
initx = [1, 0.1]
```

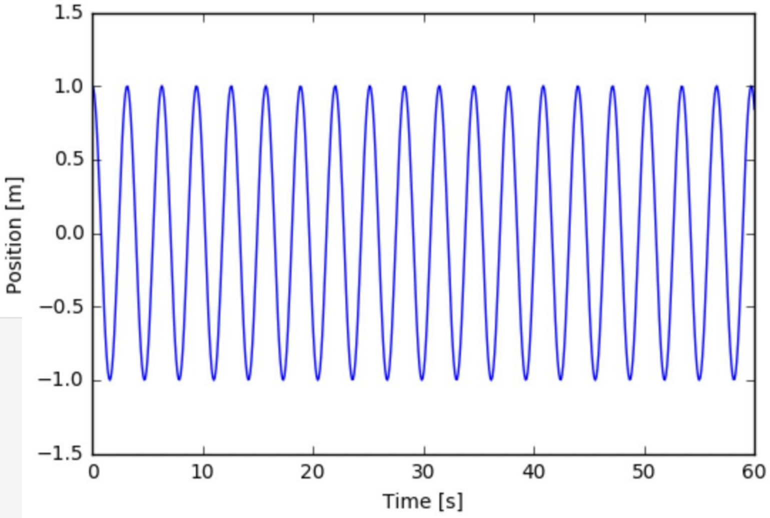
```
solver = ode('cvode', rhseqn, old_api=False, max_steps=5000)
solution = solver.solve(times, solution.values.y[-1])
if solution.errors.t:
    print ('Error: ', solution.message, 'Error at time', solution.errors.t)
print ('Computed Solutions:')
print('\n   t         Solution          Exact')
print('------------------------------------')
for t, u in zip(solution.values.t, solution.values.y):
    print('{0:>4.0f} {1:15.6g} {2:15.6g}'.format(t, u[0],
          initx[0]*np.cos(np.sqrt(k/m)*t)+initx[1]*np.sin(np.sqrt(k/m)*t)/np.sqrt(k/m)))
```

Computed Solutions:

t	Solution	Exact
2	-0.691508	-0.691484
60	0.843074	0.843212
120	0.372884	0.373054
180	-0.235749	-0.235745
240	-0.756553	-0.756932
300	-0.996027	-0.996814
360	-0.865262	-0.866242
420	-0.412897	-0.413742
480	0.192583	0.192521
540	0.726263	0.727236
600	0.989879	0.991682
660	0.885441	0.887581
3120	0.72638	0.734626
3180	0.199071	0.203121
3240	-0.401994	-0.403871
3300	-0.853534	-0.860769
3360	-0.987807	-0.997773
3420	-0.75483	-0.763966
3480	-0.241495	-0.246241
3540	0.361435	0.362997
3600	0.82981	0.837332

Can also graph function and refine tolerances

```
#plot of the oscilator
solver = ode('cvode', rhseqn, old_api=False)
times = np.linspace(0,60,600)
solution = solver.solve(times, initx)
plt.plot(solution.values.t,[x[0] for x in solution.values.y])
plt.xlabel('Time [s]')
plt.ylabel('Position [m]')
plt.show()
```




```
options1= {'rtol': 1e-6, 'atol': 1e-12, 'max_steps': 50000}      # default rtol and atol
options2= {'rtol': 1e-15, 'atol': 1e-25, 'max_steps': 50000}
solver1 = ode('cvode', rhseqn, old_api=False, **options1)
solver2 = ode('cvode', rhseqn, old_api=False, **options2)
solution1 = solver1.solve([0., 1., 60], initx)
solution2 = solver2.solve([0., 1., 60], initx)

print('\n   t        Solution1        Solution2            Exact')
print('-----------------------------------------------------')
for t, u1, u2 in zip(solution1.values.t, solution1.values.y, solution2.values.y):
    print('{0:>4.0f} {1:15.8g} {2:15.8g} {3:15.8g}'.format(t, u1[0], u2[0],
           initx[0]*np.cos(np.sqrt(k/m)*t)+initx[1]*np.sin(np.sqrt(k/m)*t)/np.sqrt(k/m)))
```

t	Solution1	Solution2	Exact
0	1	1	1
1	-0.37069371	-0.37068197	-0.37068197
60	0.8430298	0.84321153	0.84321153

Questions and Experiments

My question is whether this is faster and/or more accurate than other Python libraries. Also wondering if there are cases that the solvers cannot handle.

Experiment: I want to try the functions on more complicated equations and compare its speed and accuracy to other Python libraries. Utilize plots of error vs. time and use conditioning functions for each package to assess the differences.

Link to Github repository

https://github.com/bmcage/odes