ODESOLVERS

- A Unified Python Interface to a Variety of Software for Solving Oridinary Differential Equations
- Available at https://github.com/hplgit/odesolvers

Part 1

Background

Ordinary Differential Equation

- We focus only on solving IVP for first order ODE systems.
- $u' = du/dt = f(u, t)$, scalar or vector.
- Numerical methods:

  explict vs implicit, one-step vs multi-step
- Existing ODE software:

  1. Traditional ODE solvers developed in Fortran or C:

     Well-tested, stable, efficient, but cubersome to use.

  2. ODE software developed in modern languages:

     Clean syntax, user-friendly, but lack of generic interface.

Problem Area

- Difficult to choose an appropriate ODE solver, especially for novice users who are new to ODE.
  - Diversity of numerical methods and software.
  - Accuracy, stability and efficiency.
  - User's understanding about solvers and the specific ODE problem.
- Troublesome to switch among existing ODE software.

Switch-over Example
$u'' = 3\,(1-u_2)\,u' + u,\ u_0 = (2.0, 1.0)$

- *Dlsode* in *ODEPACK*

```
      program main
      external f
      integer i, iopt, iout, istate, itask, itol, iwork,
     1        lrw, liw, mf, neq, nout
      double precision atol, t, tout, rtol, rwork, u, urr
      dimension u(2), rwork(52), iwork(20), urr(5,2)
      neq = 2
      mf = 10
      itol = 1
      rtol = 0.0d0
      atol = 1.0d-6
      lrw = 52
      liw = 20
      nout = 5
      t = 0.0d0
      tout = 0.2d0
      u(1)= 2.0d0
      u(2) = 0.0d0
      itask = 1
      istate = 1
      do 100 iout = 1, nout
        call dlsode(f,neq,u,t,tout,itol,rtol,atol,itask,
     1    istate,iopt,rwork,lrw,iwork,liw,jac,mf)
        urr(iout,1) = u(1)
        urr(iout,2) = u(2)
100     tout = tout + 2d-1
      end
      subroutine f(neq, t, u, udot)
      integer neq
      double precision t, u, udot
      dimension u(2), udot(2)
      udot(1) = u(2)
      udot(2) = 3.0d0*(1.0d0 - u(1)*u(1))*u(2) - u(1)
      return
      end
```

- *ode45* in *Matlab*

funsys.m:

```
function F = funsys(t, u)
F(1, 1) = u(2)
F(2, 1) = 3*(1 – u(1)*u(1))*u(2) – u(1)
```

In *Matlab* command window:

```
[t_s, u_s] = ode45('funsys', [0 1], [2;0])
```

- *ode* in *scipy*

```
from scipy.integrate import ode
u0, t0 = [2.0, 0.0], 0.0
def f(t, u):
    return [u[1],
            3.*(1. - u[0]*u[0])*u[1] - u[0]]
r = ode(f).set_integrator('dopri5',
            with_Jacobian=False)
r.set_initial_value(u0, t0)
while r.successful() and r.t <= 1.0:
    r.integrate(r.t + 0.2)
```

The Preferred Solution: A Unified Interface for ODE Solvers

- Unified interface → Easy-to-switch
- Clean syntax → Easy-to-study for users without much programming experience
- User-friendly interface → Easy-to-use for novices

Part 2

Usage

Typical usage

- Write the ODE problem in generic form $u' = f(u, t)$
- Implement the right-hand side function $f(u, t)$
- Create a method object with desired solver

  method = SomeSolver(f, prm1=..., prm2=..., ...)
- Set initial status, $u(0) = u0$

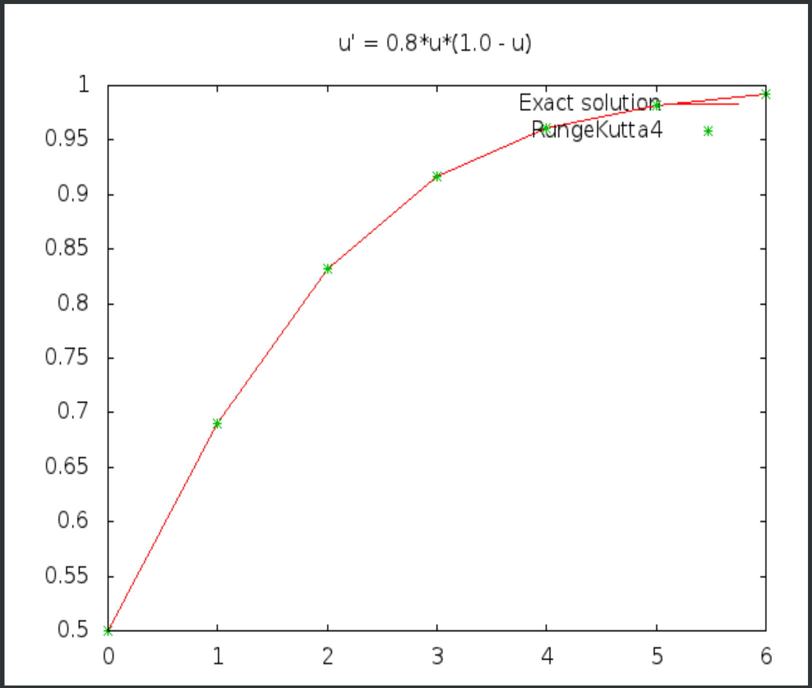
  method.set_initial_condition(u0)
- Solve the ODE problem within desired domain of time points

  u, t = method.solve(time_points, terminate=...)

Example 1:
Logistic model with Population Growth Problem

- u'(t) = a(u,t) (1 - u(t)) /C

  where a = 0.8, C = 1.0



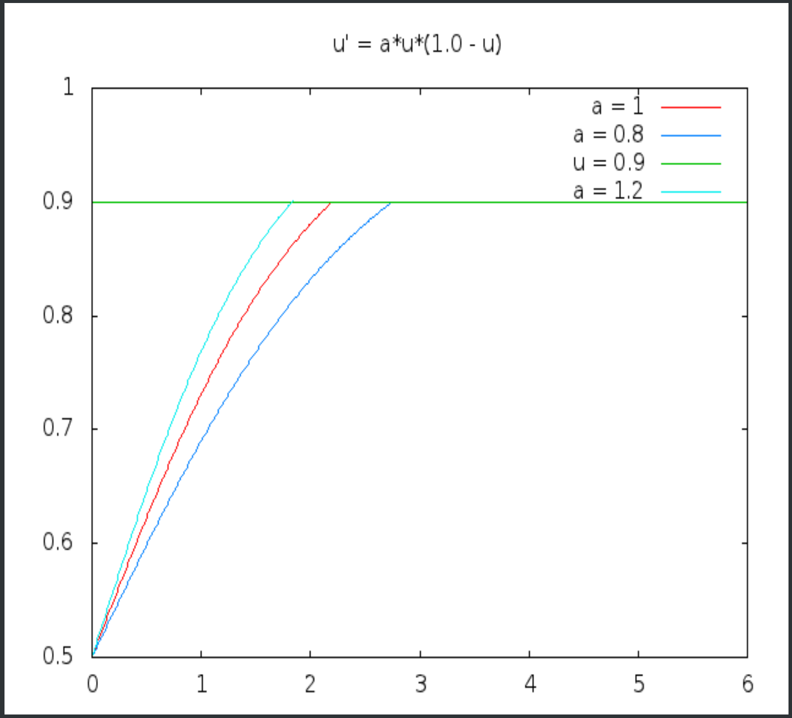
- *def f(u, t):   # a = 0.8, C = 1.0*

  *return 0.8*u*(1 – u)*

- *method = odesolvers.RungeKutta4(f)*
- *method.set_initial_condition(0.5)   # u0 = 0.5*
- *u, t = method.solve(numpy.arange(0.0, 7.0, 1.0))*

How Will the Population be Affected with Different Growth Rates? When Will the Population Exceeds 0.9 Million?

```
def f(u, t, a):           # f with extra parameter
    return a*u*(1.0 - u)

time_points = numpy.arange(0.0, 7.0, 0.01)
u0 = 0.5

def _terminate(u, t, step_number):
    # Stop when population exceeds 0.9 million
    tol = 1e-6
    return (u[step_number] - 0.9) < tol

u_solutions = {}
for a in [0.8, 1.0, 1.2]:      # 3 different values of a
    key = 'a = %g' % a
    method = odesolvers.RungeKutta4(f, f_args=(a,))
    u_solutions[key], t = method.solve(time_points,
                                       terminate=_terminate)
```



- Extra argument for f: a
- Loop with 3 different values for a
- Stop events are defined

Example 2: Van der Pol Oscillator Problem
Try three solvers in a single loop.

- $y'' = 3(1 - y^2)y + y$

  Introduce a new vector $u = (y, y') = (u_0, u_1)$,

  $u' = (u_1, 3.0*(1.0 - u_0*u_0)*u_1 + u_0)$

- ```
  def f(u,t):   # Define ODE system with vector variables
          u_0, u_1 = u
          return [u_1, 3.0*(1.0 - u_0*u_0)*u_1 + u_0]
  ```

- ```
  from odesolvers import *
   solver_list = [Lsode, DormandPrince, Vode]
   solutions = {}
  for solver in solver_list:
      method = solver(f)
      method.set_initial_conditions([2.0, 1.0])
      solutions[method.__name__], t = method.solve([0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
  ```

Ode45 in Matlab

Scipy.integrate.ode

Dlsode in ODEPACK

Switch-over Example
$u'' = 3\,(1-u_2)\,u' + u,\ u_0 = (2.0, 1.0)$

- *Dlsode* in *ODEPACK*

```
 program main
 external f
 integer i, iopt, iout, istate, itask, itol, iwork,
1          lrw, liw, mf, neq, nout
 double precision atol, t, tout, rtol, rwork, u, urr
 dimension u(2), rwork(52), iwork(20), urr(5,2)
 neq = 2
 mf = 10
 itol = 1
 rtol = 0.0d0
 atol = 1.0d-6
 lrw = 52
 liw = 20
 nout = 5
 t = 0.0d0
 tout = 0.2d0
 u(1)= 2.0d0
 u(2) = 0.0d0
 itask = 1
 istate = 1
 do 100 iout = 1, nout
    call dlsode(f,neq,u,t,tout,itol,rtol,atol,itask,
1     istate,iopt,rwork,lrw,iwork,liw,jac,mf)
    urr(iout,1) = u(1)
    urr(iout,2) = u(2)
100    tout = tout + 2d-1
 end
 subroutine f(neq, t, u, udot)
 integer neq
 double precision t, u, udot
 dimension u(2), udot(2)
 udot(1) = u(2)
 udot(2) = 3.0d0*(1.0d0 - u(1)*u(1))*u(2) - u(1)
 return
 end
```

- *ode45* in *Matlab*

funsys.m:

```
   function F = funsys(t, u)
   F(1, 1) = u(2)
   F(2, 1) = 3*(1 – u(1)*u(1))*u(2) – u(1)
```

In *Matlab* command window:

```
   [t_s, u_s] = ode45('funsys', [0 1], [2;0])
```

- *ode* in *scipy*

```
from scipy.integrate import ode
u0, t0 = [2.0, 0.0], 0.0
def f(t, u):
      return [u[1],
                  3.*(1. - u[0]*u[0])*u[1] - u[0]]
r = ode(f).set_integrator('dopri5',
                  with_Jacobian=False)
r.set_initial_value(u0, t0)
while r.successful() and r.t <= 1.0:
      r.integrate(r.t + 0.2)
```

Further usage:
Supply *f* as a Fortran Subroutine

```
f_str = """
          subroutine f_f77(neq, t, u, udot)
Cf2py intent(hide) neq
Cf2py intent(out) udot
         integer neq
         double precision t, u, udot,i,j,k
         dimension u(neq), udot(neq)
         udot(1) = u(2)
         udot(2) = 3d0*(1d0 - u(1)**2)*u(2) - u(1)
         return
         end
"""

m = odesolvers.Lsode(None, f_f77=f_str)
m.set_initial_condition(u0)
u,t = m.solve(time_points)
```

f_f77 can either be supplied as multi-line string in Fortran, or as a F2py-compiled object.

```
from numpy import f2py
f2py.compile(f_str, modulename='callback',
                    verbose=False)
import callback
f_f77 = callback.f_f77
m = odesolvers.Lsode(None, f_f77=f_f77)
```

f_f77 is not a legal parameter for Fehlberg, which means that a Python function *f* is required as a mandatory parameter for class Fehlberg.
Then, if we want to try the same problem with Fehlberg method, must we re-implement *f* in Python?   No!

```
# Switch to Fehlberg
m_new = m.switch_to('Fehlberg')
u_new, t = m_new.solve(time_points)
```

In Lsode:
self._parameters['f_f77'] = dict(name_wrapped='f', paralist_old='t,u', paralist_new='u,t')
self.f = lambda u,t: self.f_f77(t,u)   → Automatically done in function func_wrapper()

Example 3:
A Simple DAE Problem: Chemical Kinetics

- Linearly implicit systems of first order ODEs,
  A(u,t) * u' = g(u,t) , where A is a square matrix.
- Solvers: *Lsodi, Lsoibt, Lsodis* in *Odepack*

- u(1)' = -0.04*u(1) + 1e4*u(2)*u(3)
  u(2)' = 0.04*y(1) - 1e4*u(2)*u(3) + 3e7*u(2)**2
  0 = u(1) + u(2) + u(3) - 1
- Where A =
  1 0 0
  0 1 0
  0 0 0

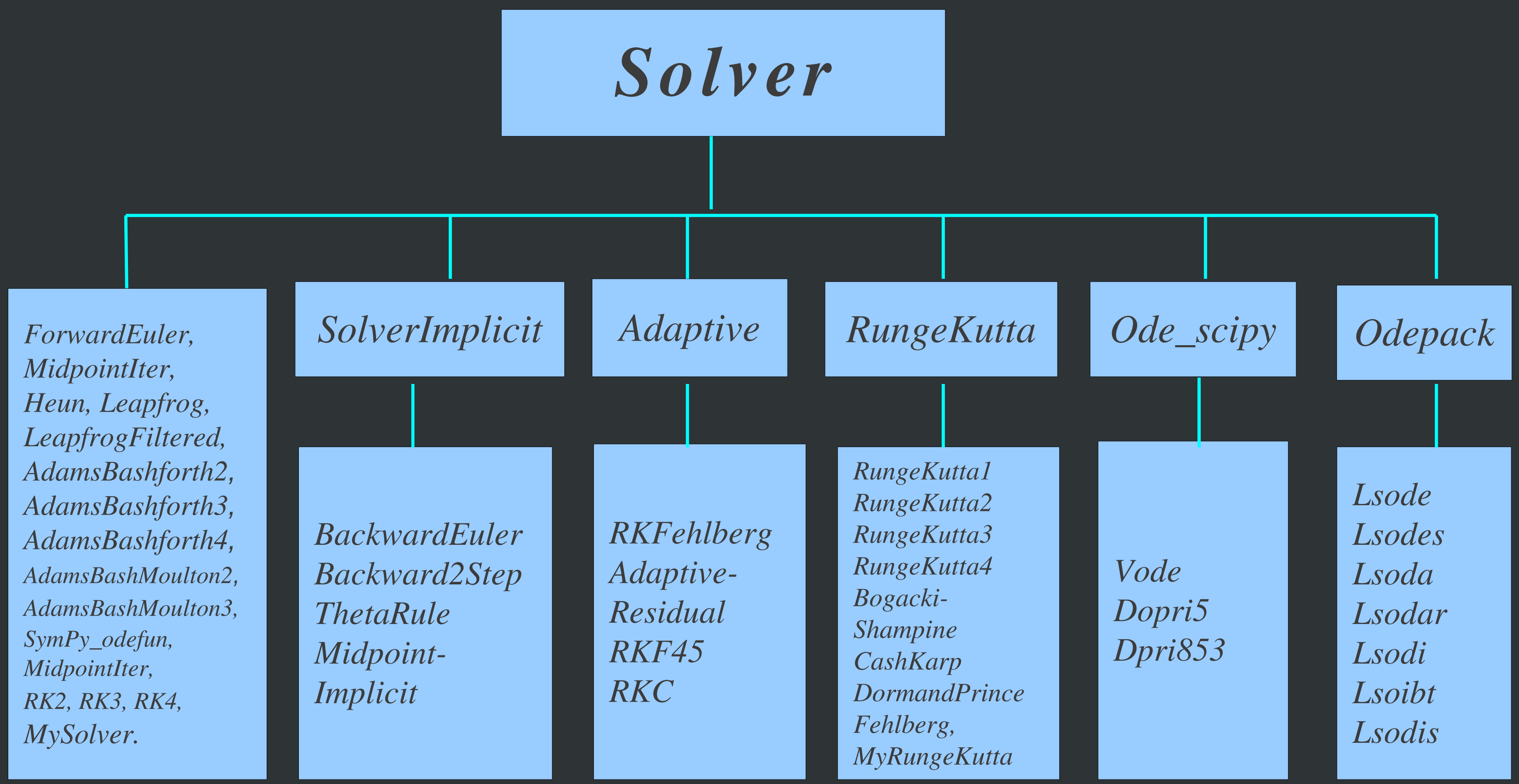
- Supply a function *res* to compute the residual:
  *r = g(u,t) – A(u,t)*s*
  s is an internally generated approxiamation for *u'*.
- Supply a function *adda* to add *A* to another matrix *P*.
- Jacobian matrix *dr/du* is an optional input.
- Provide initial value of *u* and *u'*.
- Solve this DAE problem with the same steps as ODE problems.

```
def res(u, t, s, ires):
    r = numpy.zeros(3,float)
    r[0] = -.04*u[0] + 1e4*u[1]*u[2] - s[0]
    r[1] = .04*u[0] - 1e4*u[1]*u[2] - 3e7*u[1]*u[1] -
        s[1]
    r[2] = sum(u) - 1
    return r,ires
def adda(u, t, p):
    p[0][0] += 1.
    p[1][1] += 1.
    return p

u0 = [1.,0.,0.]
time_points = np.linspace(0., 4., 5)
ydoti = [-.04,.04,0.]      # initial value of du/dt

m = odesolvers.Lsodi(res=res, ydoti=ydoti,
      adda_lsodi=adda)
m.set_initial_condition(u0)
u,t = m.solve(time_points)
```

Part 3

Design Issues

Class Hierarchy



- 43 subclasses
- Each solver in this package is implemented as a class in a class hierarchy with a common super class ***Solver***.
- Common functionality and attributes is inherited from super classes.
- Specific settings and fomula is implemented in the actual subclass.

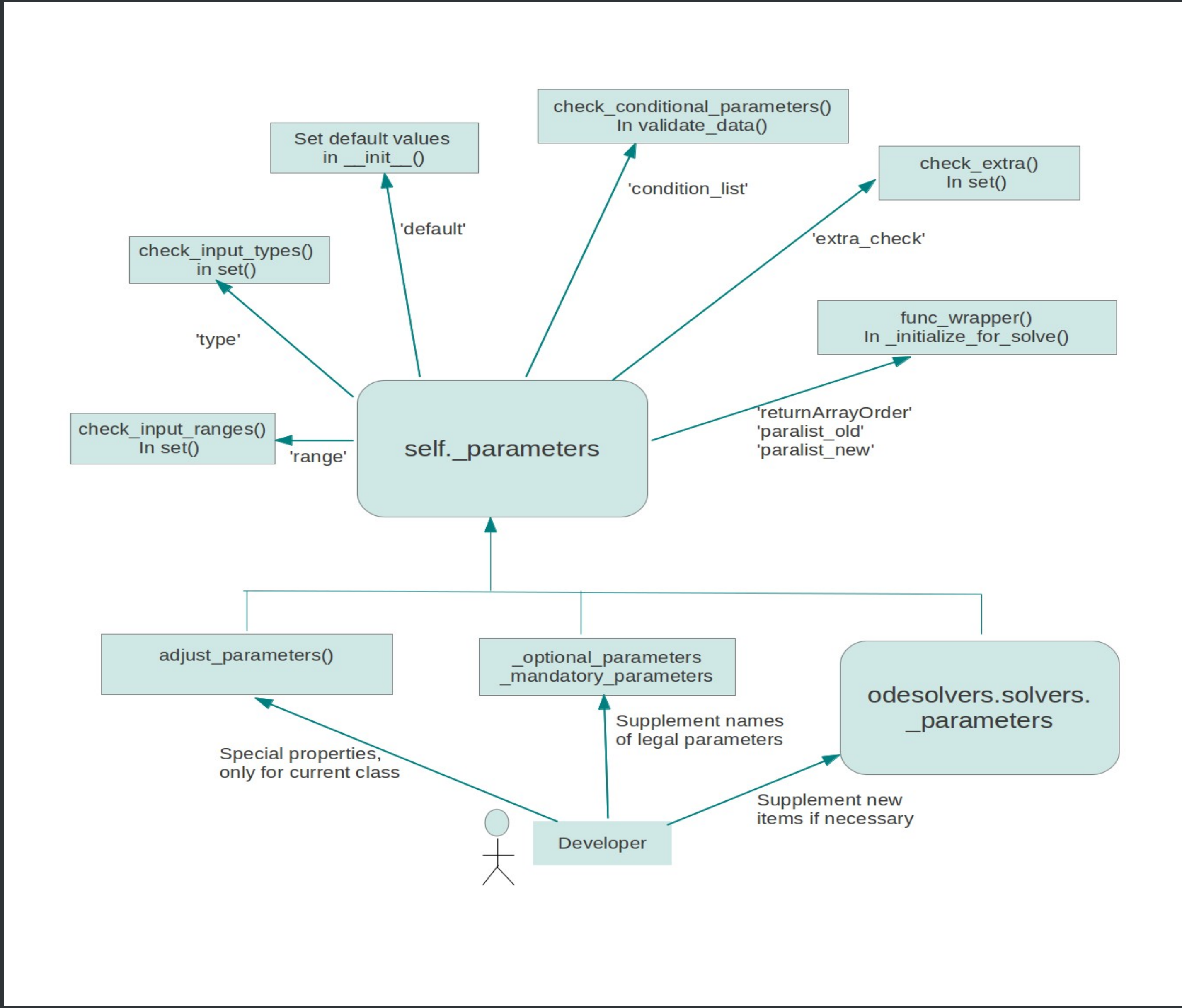
solvers._parameters & self._parameters

```
solvers._parameters:
_parameters = dict(
      f = dict(
            help = 'Right-hand side f(u,t) defining ODE',
            type = callable),
      …
      atol = dict(
            help='absolute tolerance for solution',
            type=(float,list,tuple,numpy.ndarray),
            default=1e-8),
      beta = dict(…),
      …
      )

>>> print len(solver._parameters.keys())
79
```

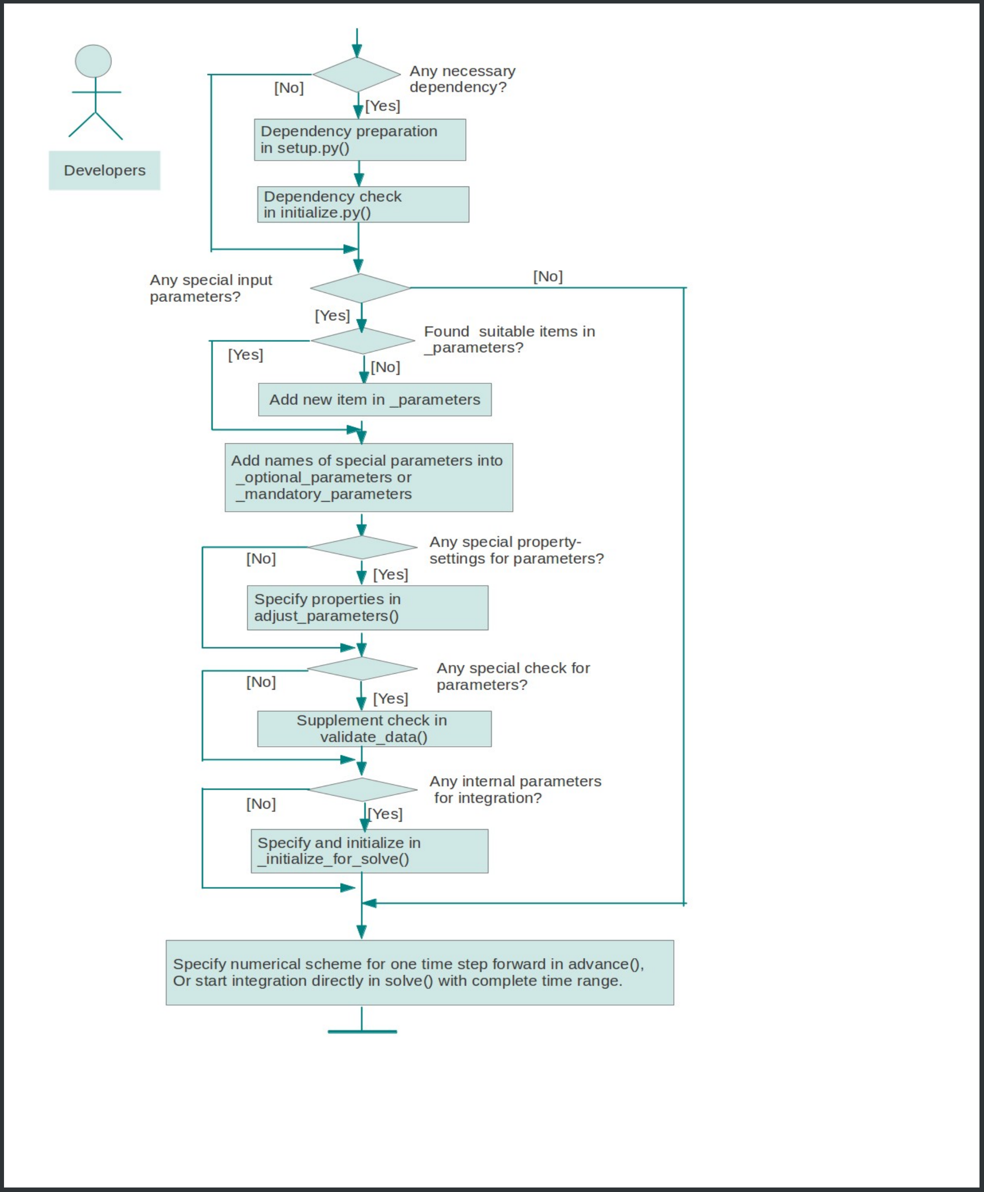
```
SomeMethod._required_parameters = ['f', ]
SomeMethod._optional_parameters = ['atol', ]

>>> m = SomeMethod(None)
>>> import pprint
>>> print pprint.pformat(m._parameters)
{'atol': {'default': 1e-08,
            'help': 'absolute tolerance for solution',
            'type': (<type 'float'>,
                        <type 'list'>,
                        <type 'tuple'>,
                        <type 'numpy.ndarray'>)},
'f': {'help': 'right-hand side f(u,t) defining the ODE',
      'type': <built-in function callable>}}
>>> print len(m._parameters.keys())
2
```

The property-information of legal parameters of a particular solver is loaded into *self._parameters* in the constructor function *__init__()*.

Automatic Check & Process of Parameters



Routine to Integrate a New Solver




```
class RungeKutta3(Solver):
def advance(self):
    u, f, n, t = self.u, self.f, self.n, self.t
    dt = t[n+1] - t[n]
    k1 = dt*f(u[n], t[n])
    k2 = dt*f(u[n] + 0.5*k1, t[n] + dt/2.0)
    k3 = dt*f(u[n] - k1 + 2*k2, t[n] + dt)
    unew = u[n] + (1/6.0)*(k1 + 4*k2 + k3)
return unew
```

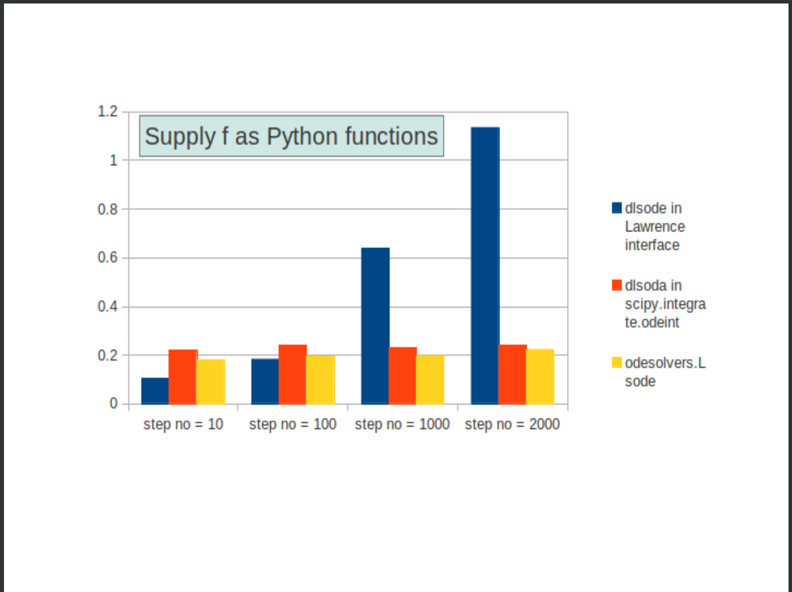
Part 4

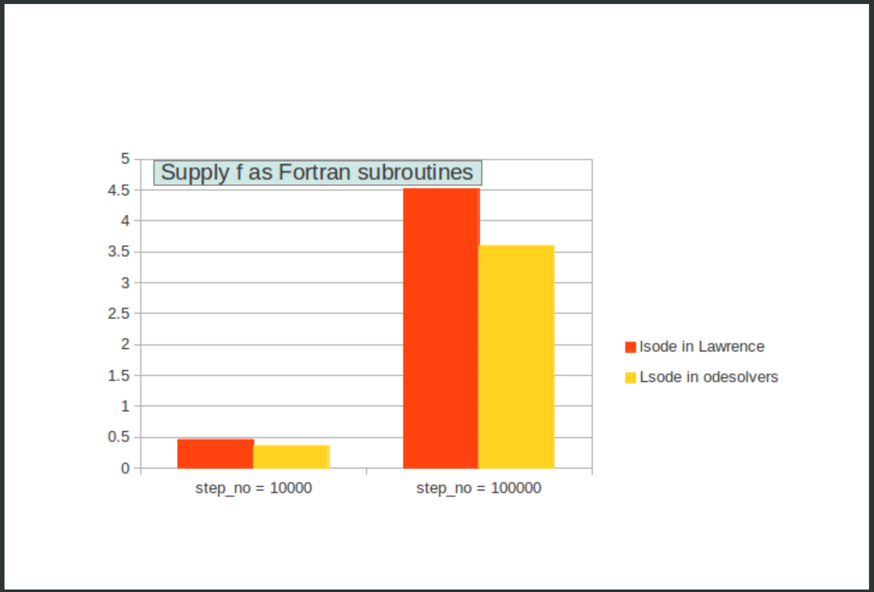
Odepack

Comparison with Fortran Package ODEPACK

- Fewer parameters:
  dlsode in ODEPACK: 17 mandatory parameters + 10 optional ones
  Lsode in *odesolvers*: 1 mandatory parameter + 15 optional ones
- Shorter code: 346 lines in Fortran → 50 lines in Python.
  Precise syntax of Python
  simplified user interface of odesolvers
  shortened parameter list
- Better error-check, clearer messages:
  Suppose a user forget to supply jacobian matrix when prameter *iter_method* is set to 1,
  dlsode in ODEPACK → *Segmentation fault.*
  Lsode in *odesolvers* → *ValueError: Error! Insufficient input!*
  jac must be set when iter_method is 1!
- Fewer interrupts with automatic error-handeling
  For instance, when error-flag is returned as -1, → *Excessive amount of work.*
  dlsode in ODEPACK → *Interrupt.*
  Lsode in *odesolvers* → max_step = max_step + 200, and restart iteration.
  (Increase the allowed step numbers without interrupt.)

Comparison With Other Python Wrappers of ODEPACK





More efficient in comparison with the *ODEPACK* wrappers in *scipy* and Lawrence lab

1. In both figures, the running time of *odesolvers.Lsode* is much shorter than ode.*lsode* in Lawrence interface.
2. In the first example, *odeint* in *scipy* shows comparable efficiency as Odepack. However, in *odein*t, *f* can only be supplied as a Python function. This feature makes its efficiency in a limited level.

Part 5

PyDSTOOL

PyDSTool

```
xnames = ['x1', 'x2']
timeData = array([0, 11, 20])
x1data = array([10.2, -1.4, 4.1])
x2data = array([0.1, 0.01, 0.4])
xData = makeDataDict(xnames, [x1data, x2data])
DSargs = args(tdata=timeData, ics=xData, name='interp')
interptable = Generator.InterpolateTable(DSargs)
itabletraj = interptable.compute('interp')
wfn_str = '100 - (1 + a)*w*heav(0.2-sin(t+1))-2*itable -
      k*auxval1(t, w)*auxval2(y)'
yfn_str = '50 - (w/100)*y'
```

```
auxfndict = {'auxval1': (['t', 'x'],
                'if(x>100,2*cos(10*t/pi), 0.5)'),
                'auxval2': (['x'], 'x/2')}
DSargs = args(name='ODEtest')
DSargs.tdomain = [0,10]
DSargs.pars = {'k':1, 'a':2}
DSargs.inputs = {'itable' : itabletraj.variables['x1']}
DSargs.varspecs = {'w': wfn_str, 'y': yfn_str}
DSargs.fnspecs = auxfndict
DSargs.algparams = {'init_step' :0.02, 'strictopt':False}
DSargs.ics = {'w':30.0, 'y': 80}
```

- PyDSTool is claimed to be 'a sophisticated & integrated simulation and analysis environment for dynamical systems models of physical systems (ODEs, DAEs, maps, and hybrid systems)'.
- With this target, its structure and user interface is very sophisticated with a lot of options, related classes and very special syntax.
- Core classes in PyDSTool: *Point, Pointset, Variable, Trajectory, str, QuantSpec, Quantity, Var, Par, Fun, Input, Generator, Model, Event, PyCont, and Toolbox classes*.
- The above code is its 'a simple ODE' example in 'Get started' part from its webpage. Is it really simple?

My Attempt in PyDSTool.py

- The first design principle of PyDSTool is described as index-free data structures. That is, the numerical data are stored mainly through Python dictionaries with string keys in PyDSTool.
- For instance, regarding an ODE system with 5 equations (neq=5),

  u → *Var('u0'), Var('u1'), Var('u2'), Var('u3'), Var('u4')*

  t → *Var('t')*

  f → *Fun('u0'), Fun('u1'), Fun('u2'), Fun('u3'), Fun('u4')*
- My wrappers for u and t:

  name_list = ['u%d' % i for i in range(neq)] # ['u0', 'u1', 'u2', 'u3', 'u4']

  u, t = [PyDSTool.Var(name) for name in name_list], PyDSTool.Var('t')
- Error occurs with the wrapping of function *f*.

Error with wrapping f to be:
Fun('udot0'), Fun('udot1'), Fun('udot2'), Fun('udot3'), Fun('udot4')

- f is wrapped from f(u,t) to f_wrap(u0,u1,u2,u3,u4,t) first.

  *f_wrap = eval('lambda *args: f(args[:-1], args[-1])')*

- Then we should seperate f_wrap to be 5 separate function objects.

  string2 = ','.join(['u[%d]' % i for i in range(neq)])   #   'u[0], u[1], u[2], u[3], u[4]'

  *udot = [eval('PyDSTool.Fun(f_wrap(%s,t)[%d],[%s],\"udot%d\")' *

  % (string2, i,string2,i)) for i in range(neq)]

- In other words,

  udot[0] = PyDSTool.Fun(f_wrap(u[0],u[1],u[2],...,t)[0], [u[0],u[1],u[2],...], "udot0")

  udot[1] = PyDSTool.Fun(f_wrap(u[0],u[1],u[2],...,t)[1], [u[0],u[1],u[2],...], "udot1")

  …

- This step is exactly where error happens.

  IndexError: 0d-array cannot be indexed.

  Reasonable because the return value of user-supplied *f* cannot be predicted in compiling process.

Part 6

Conclusion

Why odesolvers?

- Completeness

  43 solvers have been integrated.

  Both classic numerical methods and complicated ODE solvers are available for different users.
- Simple syntax & friendly user interface

  Easy-to-learn, easy-to-switch, save programming time.
- Free-of-charge and Easy-to-install
- Great potential for future development

  Planned to integrate the following solvers in the following month:

  *odelab, sundial*s, a couple of F77 routines codes by Hairer and Wanner.
- Easy-to-join with other Python tools
- A suitable facility for university courses

References

- PyDSTool homepage, available at http://www2.gsu.edu/~matrhc/PyDSTool.htm
- Test set for IVP solvers, available at http://www.dm.uniba.it/~testset/testivpsolvers
- Netlib Repository, available at http://www.netlib.org
- WikiPedia page for Ordinary Differential Equations, available at http://en.wikipedia.org/wiki/Ordinary_differential_equation
- Hans Petter Langtangen, Python Scripting for Computational Science, Second edition, 2005
- Clewley RH, Sherwood WE, LaMar MD, Guckenheimer JM (2007) PyDSTool, a software environment for dynamical systems modeling. Available at http://pydstool.sourceforge.net

The End

Ordinary Differential Equation

- A relation that contains functions of only one independent variable, and one or more of their derivatives with respect to that variable.
- Initial value problems (IVP).
- High order ODE can be transformed to be a system of first order differential equations.
- Hence, we focus only on solving IVP for first order ODE systems.
- $u' = du/dt = f(u, t)$, scalar or vector.

Numeric Methods for Solving ODEs

- Explicit Vs Implicit:

  $u' = f(u,t) \rightarrow (u_{n+1} - u_n)/\Delta t.$

  $u_{n+1} = u_n + \Delta t\, f(u, t)$

  Explicit (ForwardEuler) : $u_{n+1} = u_n + \Delta t\, f(u_n, t_n)$

  Only conditionaly stable, but simpler to implement.

  Implicit (BackwardEuler): $u_{n+1} = u_n + \Delta t\, f(u_{n+1}, t_{n+1})$

  Often unconditionaly stable, but harder to solve.

- Stiff ODEs often require an implicit solver, which is more expensive for each time step.

- One-step & Multi-step

Existing ODE Software

- Traditional ODE solvers developed in Fortran or C:

  ODEPACK, *rkc*, *rkf45*, *vode*, *epsode*, etc..

  Well-tested, stable, efficient,

  but cubersome to use.

- ODE software developed in modern languages:

  Scipy, *Sympy*, *Sundials*, *Matlab*, etc..

  Clean syntax, friendly user interface,

  but still in lack of generic interface.

Why Python?

- Popular in the field of scientific computing.
- Clean and friendly syntax
- Rich modularization
- Great features for scientific computing, like slicing of numeric arrays, vectorization of functions.
- Strong support with various powerful libraries, e.g. GUI programming, CGI programming, etc..
- Mixed-language programming style

User interface

- __init__(self, f, **kwargs)
- set_initial_condition(self, u0)
- solve(self, time_points, terminate=None)
- switch_to(self, solver_target, print_info=False, **kwargs)
- set(self, strict=False, **kwargs)
- get_parameter_info(self, print_info=False)
- __repr__(self)
- __str__(self)   // parameters with non-default values
- list_all_solvers()
- list_available_solvers()

_parameters:

Global dictionary holding properties of all parameters.

'default'	(int, float)
'type'	1e-6
'help_info'	'Relative error tolerance'
'range'	(1, 2, 4, 5) or (0.0, 1.0)
'condition_list'	{'1': ('jac',), '5':('ml','mu'), …}
'paralist_old'	'u, t'
'paralist_new'	't, u'
'returnArrayOrder'	'C' or 'Fortran'
'name_wrapped'	'f'
'extra_check'	lambda float_seq: numpy.asarray(\ map(lambda x: isinstance(x, float), float_seq)).all()

func_wrapper()

- Incompatible order in parameter lists
- Incompatible array-index between Python arrays and Fortran arrays
- Store return value of user-supplied functions in column-order

  Compilation with *-DF2PY_REPORT_ON_ARRAY_COPY = 1*.

return Array Order	paralist_old	paralist_new	name_wrapped	
None	'u,t,col_index-1'	't,u,col_index'	'jac_column_f77'	self.jac_column_f77 = lambda t,u,n: jac_column(u,t,n-1)
C	'u,t'	't,u'	'jac_f77'	self.jac_f77 = lambda t,u: numpy.asarray(jac(u,t))
Fortran	'u,t'	None	'jac'	self.jac = lambda u,t: numpy.asarray(jac(u,t), order='Fortran')