UNIVERSITY OF OSLO
Department of Informatics

A Unified Python Interface to a Variety of Software for Solving Ordinary Differential Equations

Master thesis

Liwei Wang

6th March 2012

Contents

Preface

Nowadays there is a jungle of practical software and numerical methods to solve the ordinary differential equations (ODE) related problems; thus making the right choice of which method and software to use has become a headache decision problem for many users. This thesis aims to introduce an unified Python interface that collects a variety of such software and methods. Ideally, this interface should provide a simple user interface for those novice users so that they can apply selected numerical methods to solve their simple ODE problems; also it could enable them to easily switch between these solvers with a minimal effort. Meanwhile, for those professional users, this interface can be extended to an advanced one, which offers the expected functionality of advanced methods with great flexibility.

The interface is named as *odesolvers*, and mainly written in Python, a popular scripting language with great development flexibility in scientific computing and a clean and easy-to-use syntax. We try to present users this interface in the clean and easy-to-use Python syntax, and steer underlying numerical software with more functionalities efficiently.

Context

Chapter 1 gives a brief view for the motivation and background of *odesolvers*. Then we move forward to why we choose Python to develop this interface for its rich modularization features and good support for numerical computing.

A detailed introduction for the features and usage of *odesolvers* are provided in chapter 2, together with a set of examples. As one of the main and great feature of *odesolvers*, it is quite extensible for future development. This benefits much from its unified parameter-dictionary and well-designed methods with specified functionalities. A normal routine is suggested for future developers to integrate a new solver into *odesolvers*.

Chapter 3 describes how to wrap the important Fortran package ODEPACK such that it can be called from Python, with a user interface that is significantly simpler than calling the routines directly from a Fortran 'driver' program.

In the following chapter 4, a class *RungeKutta* is presented in detail with purpose to make a collection for all the explicit Runge-Kutta methods, including both unadaptive Runge-Kutta methods and adaptive Runge-Kutta methods, like Dormand&Prince Runge-Kutta method which is applied in the very popular ode45 routine in Matlab. Comparing with using the commercial licensed Matlab, users of *odesolvers* can make access to many more explicit RK methods, and for free.

Efficiency issues is the subject of chapter 5. As a well-known disadvantage of scripting language, there is often considered for unavoidable loss of efficiency for Python works with slow number crunching and callback functions. In this chapter, a set of experiments are taken to compare the efficiency cost of *odesolvers* with several other ODE solvers

Appendix A contains a brief overview of all the existing solvers in *odesolvers* at the time of this writing. Appendix B contains complete code for some examples in this thesis.

Background requirement

This thesis is written based on the assumption that the readers have some basic knowledge about Python programming language and have a general understanding of ordinary differential equations problems.

Acknowledgment

First of all, I would like to thank my supervisor Professor Hans Petter Langtangen for his invaluable help with this project. From the start of this project, I did not have much experience with mathematical modelling and scientific computing. It was Hans Petter's endless encouragement and continuous insights that guided me through the way of my exploration into the field. I would also like to thank my family – my husband and my lovely daughters. They are always the spring of my energy. Without their support and encouragement, this thesis would never be completed.

Chapter 1

Introduction

1.1 Background knowledge

1.1.1 Ordinary differential equation

The Ordinary Differential Equation (or ODE) is a type of problems in Mathematics. A formal definition of ODE can be found from the Wikipedia site [19] as follows: "In Mathematics, an ODE is a relation that contains functions of only one independent variable, and one or more of their derivatives with respect to that variable"

In this project, we focus only on the initial value problems (or IVP) in the field of differential equations. A specified value, called the initial condition, is known at a given time point in the domain of solution. In scientific computing, mathematical modelling frequently amounts to initial value problems.

High order differential equations can be trivially transformed to be a system of first order differential equations. Hence in this project, we focus only on solving first order ODE systems either in explicit form,

$$u'(t) = f(u, t)$$

or in linearly implicit form.

$$A(u, t)u'(t) = g(u, t)$$

.

1.1.2 Numerical methods for solving ODE

In science, discrete variable methods (or the difference methods) are often used with approximation the independent variables at discrete points, usually equally-spaced in the desired domain. In ODE solvers with a purpose for general cases, the step size inbetween these discrete points are often varied for better efficiency and error-control.

The commonly used discrete variable methods are Euler's Method, Trapezoidal Method, Midpoint Method, Modified Midpoint Method (Gragg's Method), Runge-Kutta Methods, Predictor-Corrector Methods, and certain adaptive techniques such as the embedded Runge-Kutta methods and the Gragg-Bulirsch-Stoer method. Most of these classical methods are intended for non-stiff equations only.

Solvers for stiff equations usually require to evaluate the Jacobian, which can be very expensive in terms of efficiency cost. There are several well-known representatives intend for stiff equations: Rosenbrock Methods, Bulirsch-Stoer-Bader-Deuflhard semi-implicit methods, Implicit Runge-Kutta methods and widely used Backward Differentiation Formulae (BDF or Gear methods).

1.1.3 Existing ODE software

Due to the period in which traditional ODE solvers were developed, most of these traditional solvers were developed with Fortran (or C) language in old programming style; however, they are proved to be well-tested, stable and efficient in use.

ODEPACK, one of the most widely used ODE package that was developed in 1983 and last updated in 2003, is a collection of Fortran solvers suitable for both stiff and non-stiff systems in either explicit form or linearly implicit form.

On the website of Netlib[23], codes of many traditional free ODE solvers are collected and distributed. Table 1.1 is a short list for main solvers in Netlib repository.

In the past decade, there has been many ODE software developed in modern languages. Many of these software are just wrappers for traditional ODE solvers, and apply them as black boxes behind sophisticate interfaces.

Specially in Python, as a well-known language with great features for the purpose of scientific computing, there are also some widely-used ODE packages.

For example, in the important scientific package *scipy*, which contains numerous modules with scientific computing in Python, a few ODE solvers are included: odeint for Lsoda() in ODEPACK, vode for vode.f, dopri5 for Dormand&Prince method order 5, dop853 for Dormand&Prince method order 8(5,3).

1.2 An unified interface for ODE solvers is desired

1.2.1 Difficult to choose an appropriate ODE solver

Nowadays, there are so many numerical methods and software available for solving the ODE problems; then it has become a decision problem for the users to make their choices among the the diversity of available "ODE solvers".

For example, if you search for "ode" in the Netlib repository [23], 182 matches can be found. When confronted with many ODE software products, the users usually do not really know which one will be the right

Solver	Type of problem	Applied methods
composition	ODE	composition methods, mostly palindromic schemes size
cvode	large IVP	combines earlier vode and vodpk
dresol	differential Riccati equations	Adamsformula and BDSs
dverk	ODE with global error control	Verner's 5th and 6th order Runge-Kutta pair
epsode	stiff ODE	BDFs (variable coefficient formulae)
mebdfdae	stiff ODE, linearly and implicit DAE IVPs	extended BDFs
mebdfso	large sparse stiff ODE IVPs	extended BDFs
ode	ODE	Adamsmethods
odeToJav	ODEs	explicit Runge-Kutta, linearly implicit-explicit IMEX
parsodes	large stiff ODEs	multi-implicit Runge-Kutta with 'across the method' parallellization in MPI
rkc	parabolic PDEs	2nd-order explicit Runge-Kutta-Chebyshev formulae
rkf45	ODEs	Runge-Kutta Fehlberg 4th-5th order
rksuite	ODEs	a suite of codes, choice of RK methods, including an error assessment facility and a sophisticated, stiffness checker
sderoot	ODE with root stopping	Adamsmethods
sode	ODE	Adamsmethods
srkf45	ODE	Runge-Kutta Fehlberg
svode	ODEs	BDFs (variable coefficient formulae)
svodpk	large ODEs	BDFs (variable coefficient formulae) with GMRES with user-supplied preconditioner
vode	ODEs	BDFs (variable coefficient formulae)
vodpk	large ODEs	BDFs(variable coefficient formulae) with GMRES with user-supplied preconditioner

Table 1.1: Solvers in Netlib repository.

one to choose, and how this software can be optimally applied in the specific ODE problem. This usually tends to a rather difficult task to many users. Another difficult problem is also the selection of an adequate method from the various method choices of a specific software product.

Although solving ODE is an old topic and software for solving ODES developed significantly during the past decades, there seems still in lack of common measures to evaluate these methods and software [1]. The fact is, each solver has applied a set of formulas and techniques to make the code efficient and reliable, hence it suits best for some specific type of ODE problems from its intention of design. On the other hand, some solvers have been proved to be efficient and stable to many ODE problems, but none of them can solve all kinds of problems. That is also why there exist so many software for solving ODE.

Accuracy, stability/robustness and computational efficiency are the three most important features. In practice the problems are related to blow-up of results (instability) or too long execution times (too advanced solver perhaps).

In the case of an initial value problem for ODEs, a correct selection for appropriate method or software can only be done by very professional users. This is not only caused by the variety and diverse properties of softwares, but also depend much on the user's knowledge about ODE solvers and his or her understanding about properties of the specific ODE problem.

For example, let us take a look at one of the simplest ODE problem

$$u' = -15u \tag{1.1}$$

$$u(0) = 1.0 \tag{1.2}$$

$$\text{time\_points} = [0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.0] \tag{1.3}$$

We tried to apply third order Adams-bashforth method, fourth order Runge-Kutta method and ForwardEuler method to solve this simple decay problem. Figure 1.1 illustrates the different behavior of these methods. ...

In terms of the stiffness of equations, the ODE problems can be classified in three main categories: stiff, non-stiff, or unknown. While for the stiff ODE problems, most of classical ODE solvers will become much slower and possibly unstable comparing with non-stiff ones. Unfortunately, a large set of ODEs are often stiff in real cases. It becomes even worse in some cases when these ODE equations may be stiff in some parts of a time interval, but non-stiff for other parts of the same time interval. This will become a great challenge for the user to select the appropriate ODE solver.

Because of the diversity of ODE solvers and complicated properties of ODE problems, many users have to keep trying with many different numerical methods or software before a satisfactory result is reached in reasonable CPU time. Therefore, these users may also need to switch over from one to another solver in order to get a better result; however, the switch-over process is often not as easy as the imagination.

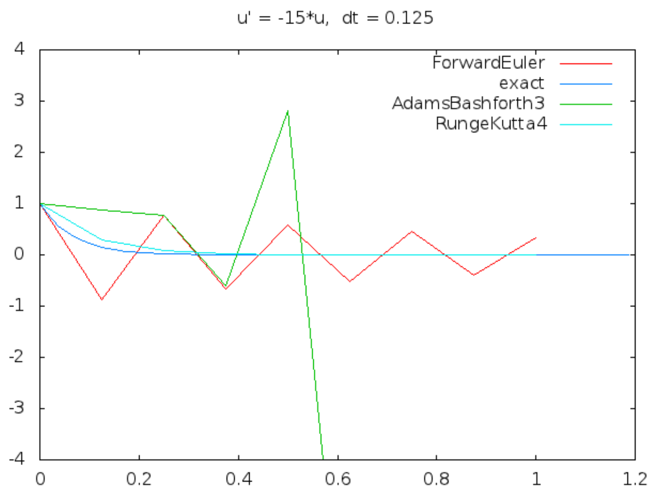


Figure 1.1: u' = -15*u, Time step = 0.125

1.2.2 Trouble to switch among ODE solvers

Due to the disadvantages of underlying low-level languages, traditional ODE solvers are often in lack of friendly user interface and thus make them difficult to use for unfamiliar users. For example, even for experienced programmers it can take several days to resolve a mistake caused by dynamic memory locating and management in the ODE applications.

Besides, comparing with other solvers in high-level languages, these packages are often complained to be cumbersome to use, due to numerous input and output parameters that need proper settings in the old programming style. For instance, considering the main solver dlsode in ODEPACK package, there are 17 mandatory input parameters required to be set up with proper values, plus 6 extra parameters as optional inputs. Furthermore, there are 10 method modes to choose in this solver. This makes its usage even more confusing for novice users.

On the other side, ODE softwares that are composed in high-level programming languages, have their own and usually different user interfaces.

For example, if we consider about the error tolerance parameters (which seems to be quite simple and clear with an uniformed mathematical definition), we can find many different names in different packages, like 'reltol', 'rtol', 'tol', 'abstol', 'atol', 'errtol', etc.. Some of these error tolerance parameters are demanded to be scalar, while the others need to be defined as sequences. These inconsistencies lead to incompatible problems among ODE solvers, thus make it difficult to switch among them.

Let us go forward with the following example, Van der Pol oscillator problem, and try to switch this simple ODE problem among several different ODE packages.

$$u'' = 3(1-u^2)u' + u \tag{1.4}$$

$$u(0) = (2.0, 1.0) \tag{1.5}$$
$$\text{time\_points} = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0] \tag{1.6}$$

Firstly, we try to solve this problem with the basic solver Lsode 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
```

As we discussed in the previous section, it is cubersome to apply ODEPACK due to its old programming style. The parameter list is very long even for this simple ODE problem. Furthermore, before starting integration, users need to select among the 10 possible method modes in basic solver Lsode, and specify input parameter *mf* with a proper value.

It seems much easier to switch to the popular solver ode45 in Matlab:

First, we need to create a Matlab file which evaluates the right-hand side of the ODE system f(t,u) for any given t and u(1), u(2), and name this file as *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)
```

Now we are ready to get the solution in Matlab command window with a single line:

```
>> [tv, uv] = ode45('funsys', [0 1], [2;0]);
```

Finally, we switch this problem to the Python module *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]]

# integration
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)
```

Now we can see clearly that in order to switch to another ODE software, users need to make much modifications, not only the input and output parameters, but also programming codes to restart integration in another programming language. Users are required to study extensively a lot of details for all these softwares,– input parameters, return values, method choices, user interfaces, and possibly basic usage of underlying programming languages.

1.2.3 Our aim: a unified interface for various ODE solvers

If the result from applying an ODE solver is not satisfactory to the users, they may probably need to try a few other ODE solvers in order to obtain a better (or perhaps the optimal) result. For an easy and smooth switchover in this process, it is desired that all ODE solvers have a very similar or even unified interface for the user to use. This is the original motivation when I started this project.

An unified interface will definitely ease the user's switch-over process inbetween many solvers. With the help of *odesolvers*, ODE users can state their problem easily and trivially switch among a variety of ODE solvers, not only from one numerical method to another one, but also from one software library to another library as well. In the field of scientific computing, there is no such kind of tool available at present, as far as we know.

From the example in 1.2.2, we can see that both Matlab and scipy module supply users a clear syntax and modern scripting programming style, but there are only a few numerical methods are implemented in both of them. Our basic goal is to create an unified interface in a similar syntax with more numerical methods or solvers integrated.

On the other hand, many users without much experience with Fortran and C want to reuse the old well-tested and efficient codes in traditional ODE packages, and apply these solvers in a modern and flexible programming environment. Considering the inconvenience and troubles for usage of ODEPACK, I hope to integrate these traditional solvers into our new interface with clearer syntax and simplified user interface, especially for those complicated ODE solvers.

I think we have a new generation of users in mind. They want more high-level languages. So the interface should look extremely simple to the novices, yet flexible enough the for those professionals. The goal is that both beginning students and professional researchers would favor our interface.

1.3 Method choices in *odesolvers*

1.3.1 Python: preferred language for *odesolvers*

As one of the most flexible and friendly programming language, Python has gained a popularity rapidly in the field of scientific computing. Python has many great features for computational science, such as slicing functionality for multi-dimension arrays, clean and friendly syntax, rich modularization, strong support for GUI programming and a variety of powerful libraries.

For example, taking advantage of automatically dynamic memory administration, which is one of Python's outstanding feature, it is trivial to get the memory problem solved in Python. Comparing with Fortran, C, and C++, which are the most popular compiled languages for scientific computing, Python removes much more debugging problems with memory management.

Furthermore, considering the diverse underlying programming languages for existing ODE softwares, it should be desirable to implement this interface in a language with strong support for mixed language programming. As a dynamically typed language, Python is implemented in C and can be easily extended with new functions written in C [3]. Besides, powerful tools like F2PY and SWIG, are developed to make Fortran, C and C++ code callable directly from Python.

However, there may be some unavoidable efficiency loss for compilation of Python scripts. As a scripting language, there are always some additional costs for its interpretation and compiling process. Fortunately, in Python code speedups can be achieved through optimized array operations, and possibly by migrating some numerical computing code to other compiling languages (e.g. Fortran, C/C++) with its mixed programming style.

1.3.2 Solvers in *odesolvers*

Generally, numerical ODE methods for solving initial value problems can be classified as explicit methods and implicit methods, and can also be classified as one-step methods and multi-step methods in terms of internal-step number. In a recent report, Numerical Methods for Differential Equations [20], many representative numerical ODE methods are illustrated and classified as four types:

1. Explicit one-step methods Euler's method, Midpoint method, 2nd order Runge-Kutta method, 4th order Runge-Kutta and embedded Runge-Kutta methods.

2. Implicit one-step methods Backward Euler's method, Trapezoid method, Implicit midpoint method (or 2nd order Gauss method),

3. Explicit Multi-step methods 2nd order Adams-Bashforth methods, 3rd order Adams-Bashforth methods, 4th order Adams-Bashforth methods,

4. Implicit Multi-step methods 2-step Adams-Moulton method, 3-step Adams-Moulton method, Backward Differentiation Formulas (BDF).

Most of ODE software, including very complicated ones, are based on these numerical methods. We can refer this point to the numerous ode solvers listed in Table 1.1.

Therefore I choose to implement all these classical numerical methods into *odesolvers* directly with Python language.

Furthermore, a set of other popular ODE methods are also implemented in Python code.

For example, in Matlab, there is a routine named *ode45*, which is considered to be the most popular method in ODE solvers. It is based on two explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair as the default method, and the Fehlberg pair as an alternative choice. Specially for these two underlying methods in ode45, I have implemented several solvers for them in *odesolvers*:

1. Fehlberg Runge-Kutta method

   class *RungeKuttaFehlberg*: Python implementation based on its own scheme.

   class *Fehlberg_RK*: Python implementation as a member of collection *RungeKutta* in the general scheme.

   class *RKF45*: Wrapper for *rkf45.f*.

2. Dormand&Prince Runge-Kutta method

   class *DormandPrince_RK*: Python implementation as a member of collection *RungeKutta* in the general scheme.

   class *Dopri5*: wrapper for *dopri5.f*.

Generally, the Python versions are slower than wrappers of Fortran subsroutines, but are quick enough for simple ODE systems on today's computers. In order to provide users an adequate choice for complicated ODE problems, I integrated these wrappers for Fortran subroutines. Moreover, when applying these wrappers for Fortran code, we provide users an option to supply user-defined functions (like *f, jac*) in Fortran code, and thus apply these underlying methods as fast as Fortran programs. In 5.2, an efficiency comparison among these solvers are presented.

Moreover, several important ODE packages are selected and wrapped as black boxes behind the friendly user interface of *odesolvers*. For example, 7 solvers in ODEPACK are integrated, which possibly makes *odesolvers* to be the most complete set for ODEPACK in Python.

A comparison of underlying numerical schemes among *odesolvers* and several important ODE packages are illustrated briefly in Table 1.2. We can see that *odesolvers* contains most of these methods, and thus provide users with a variety of method choices.

Important numerical ODE methods	Sympy	Scipy	Matlab	Sundails	Odelab	PyDSTool	Lawrence	Odesolvers
Adams-Bahsforth method order 2								√
Adams-Bahsforth method order 3								√
Adams-Bahsforth method order 4								√
Adams-Bashforth-Moulton 2-step								√
Adams-Bashforth-Moulton 3-step			√					√
Implicit Rosenbrock of order (3,2)			√					
BDF's method of order 1 to 2			√					√
BDF's method of order 3 to 5			√					
CVode			√					
Vode	√					√		√
Zvode	√							
High-order Taylor series method	√						√	
Euler method					√	√		√
Implicit Euler					√			√
ThetaRule					√			√
Midpoint with fixed-point iterations.								√
Leapfrog method								√
Standard RK 2, or Heun, Trapezoid).					√			√
Standard Runge-Kutta of order 3					√			√
Standard Runge-Kutta of order 4					√		√	√
Shampine-Bogacki of order (2,3)			√					√
Runge-Kutta methods of order (3,4)					√			√
Cash-Karp RK of order (5,4)								√
Dormand-Prince of order (5,4)		√	√			√		√
Dormand-Prince of order 8(5,3)		√						
Fehlberg Runge-Kutta of order (4,5)		√	√			√		√
Lsode							√	√
Lsodes								√
Lsoda		√					√	√
Lsodar								√
Lsodi								√
Lsodis								√
Lsoibt								√
rkc.f, Runge-Kutta-Chebyshev							√	√
rkf45.f								√

Table 1.2: Numerical methods in ODE packages

Chapter 2

odesolvers

odesolvers is designed with following principles in mind:

1. Simple and minimal for novice users
   This interface should be easy to use for new comers with a clean, Matlab-like syntax. Users can readily get their solution with only basic control and inputs.

2. Functional and flexible for advanced users
   Behind the simple user interface, advanced users can easily access to the hidden advanced features and control the integration as they want.

3. Extensible for future development
   This interface aims to be a general interface for all ODE solvers, not only a wrapper for several selected packages instead.

2.1 Solvers in a class hierarchy

In this package, ODE methods are coded in a syntax very close to its algorithmic description. However, there are some details of implementation with respect to how parameters are set and controlled to achieve maximum flexibility in general sense. This is possibly not so easy to understand for a newcomer to the package, but these details can be safely ignored by users.

Each solver is implemented as a separate class, and all these classes are collected in a class hierarchy. The base class for this hierarchy is named as *Solver*. Generic features of ODE solvers are defined in this base class, which ensure the consistency and uniform of parameters, methods, and user interface in the whole hierarchy.

The superclass *Solver* provides all the functionalities that are common to all solvers, while specific numerical algorithms and schemes are implemented in subclasses. In most solver classes, two key methods are involved for integration: *solve()* for performing the time loop and *advance()* to advance the solution one time step.

Up till now, there are 43 numerical methods and ODE software integrated in *odesolvers*. Some of them are implementation for simple algorithms directly in Python, while the others are wrappers for well-known ODE solvers. Appendix A contains a brief list for all the existing solvers and corresponding methods.

In this chapter, general features of this interface would be introduced. Section 2.2 illustrates the unified user interface with several examples. Section 2.3 describes general features from a different angle of future developers. Section 2.4 contains summary for a list of attractive features for this package. Two important solver collections *Odepack* and *RungeKutta* are presented as representatives in the following two chapters respectively.

2.2 Unified interface for users

First of all, in order to clarify the expressions used in this thesis, a general form of ODE is given as follows.

$$u' = f(u, t) \tag{2.1}$$

where f is a function to specify the right side of equations.

$$u(0) = u_0 \tag{2.2}$$

u_0 is the initial value to start with.

$$\text{time\_points} = [t_0, t_1, t_2, ...] \tag{2.3}$$

$[t_0, t_1, t_2, ...]$ is a sequence to specify the desired set of output time points.

2.2.1 Simple user interface

According to these three components of an ODE problem, (2.1, 2.2, 2.3), three key methods have been defined in *odesolvers* respectively:

1. Solver definition : *__init__(self,f,**kwargs)*
   Construct an instance in desired solver class. Legal parameters can be attached as keyword parameters in the parameter list.

2. Initial status : *set_initial_condition(self,u0)*
   Set up the initial value to start with. *u0* can be defined as either scalar or vector.

3. Time range specification: *solve(self,time_points,terminate=None)*
   Produce a solution of the ODE system at the time points specified in *time_points*, but only until the user-specified function *terminate* returns true.

These three methods are available in all existing classes, and are sufficient to establish solvers for given ODE problem, including those complicated ones. Several associate methods can also be useful for users:

4. *switch_to(self, solver_target, printInfo=False,**kwargs)*
   Switch to a new solver with same values of current useful attributes. A new solver instance in the target solver will be returned. Optional parameters can be reset or supplemented as keyword-parameters. Users can switch easily between solvers without restart solution from initialization with the same or almost same values.

5. *set(self, strict=False, **kwargs)*
   Reset or supplement the values of optional parameters.

6. *get(self, parameter_name=None)*
   Return value of a specified parameter or a dictionary that contains values of all the specified inputs.

7. *get_all_solver_names_info(printInfo=False)*
   A static method which returns a name list of all available solvers in *odesolvers*.

8. *get_parameter_info(self,printInfo=False)*
   Return a dictionary containing information (name, type, range, etc.) for all the legal input parameters in current solver.

2.2.2 Example 1: a scalar ODE system

In this section, a population model is introduced to illustrate how we can solve ODE problems easily and flexibly with the help of *odesolvers*.

Logistic model with population growth problems is defined as a single ODE equation:

$$u'(t) = au(t)\frac{1-u(t)}{C} \tag{2.4}$$

where *u(t)* denotes the number of population at time point *t*, *a* (> 0) is the population growth rate, *C* (> 0) is the population capacity (i.e. the maximum sustainable population).

The analytical solution of this model is:

$$u(t) = \frac{u_0}{u_0 + e^{-at}(C - u_0)} \tag{2.5}$$

where u_0 denotes the value of initial population.

Basic usage

Let us try to applying this logistic model to predict the number of population in the next six years, where the population growth rate is 0.8, the population capacity is 1.0 million and the initial value of population is 0.5 million.

The problem can be solved in a few lines applying the standard 4th-order Runge-Kutta method,

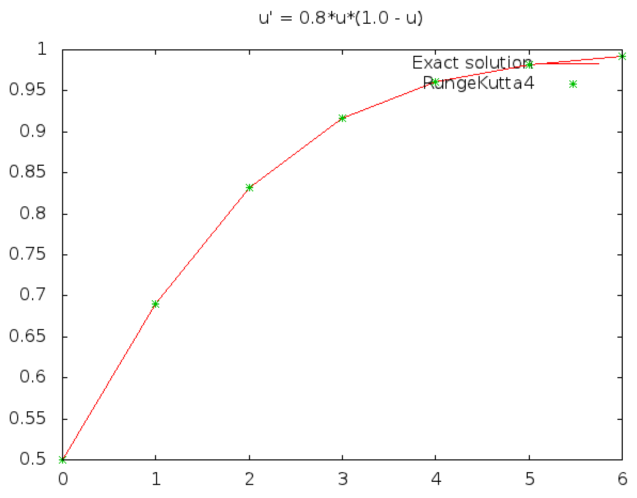


Figure 2.1: Basic usage in logistic population model

```
import odesolvers
import numpy
# a = 0.8, C = 1.0 in logistic model
f = lambda u,t : 0.8*u*(1.0 - u)
u0 = 0.5         # initial value
time_points = numpy.arange(0.0, 7.0, 1.0)

method = odesolvers.RungeKutta4(f)
method.set_initial_conditions(u0)
u,t = method.solve(time_points)
```

Let us plot the returned value together with the curve of analytical solution in Figure 2.1.

Extra parameters for function *f(u,t)*

In *odesolvers*, there are two input parameters (*f_args* and *f_kwargs*, for positional and keyword parameters respectively) that can be used to supply extra parameters for user-defined function *f*.

With help of these attributes, users can define their equation system more flexibly. According to the logistic population problem, we can make use of *f_args* to hold parameters *a* and *C* and define the ODE equation as equation (2.4).

Suppose we want to make a comparison: Setting population growth rate with different values, how much would the value of population be affected?

Firstly, we need to defined a function that take growth rate *a* and population capacity C as extra parameters, additional to the common ones (u,t).

```
def f(u, t, a, C):
    return a*u*(1.0 - u)/C
```

This is definitely incompatible with the general form *f(u,t)*. Then we need to make use of *f_args* to make it recognizable in *odesolvers*.

```
u0 = 0.5
time_points = numpy.arange(0.0, 7.0, 1.0)
u_solutions = {}
```

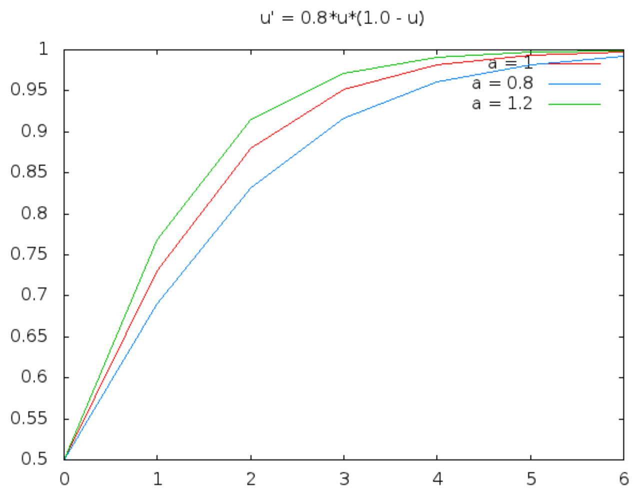


Figure 2.2: Usage of extra parameters in logistic population model

```
for a in [0.8, 1.0, 1.2]:      # 3 different values of a
    key = 'a = %g' % a
    C = 1.0

    method = odesolvers.RungeKutta4(f, f_args=(a,C))
    method.set_initial_condition(u0)
    u_solutions[key], t = method.solve(time_points)
```

Returned values are plotted in the Figure 2.2.

Stop events

Users often want to simulate until some property of the solution is fulfilled. In *odesolvers*, a special parameter *terminate* for method *solve*() is defined for this purpose. *terminate(u,t,step_number)* is a user-given function returning a boolean value. As long as it return *False* at current step, the iteration will be interrupted.

With help of this parameter, we can go further with the logistic population model: When will the population exceed 0.9 million with different growth rates?

The stop event can be specified as :

```
def terminate(u, t, step_number):
    tol = 1e-6
    return (u[step_number] - 0.9) < tol
```

Simply adding this function as a keyword argument for function *solve(self, time_points, terminate=terminate)*, we can get the solution illustrated in Figure 2.3.

As parameter *terminate* is defined in the superclass *Solver*, all the solvers in this hierarchy could make use of it to control the stop condition. Suppose package *scipy.integrate.odeint* about which the user complained, has been integrated into this interface, the user could apply *terminate* to stop iteration based on value of *u*, and get the desired solution.

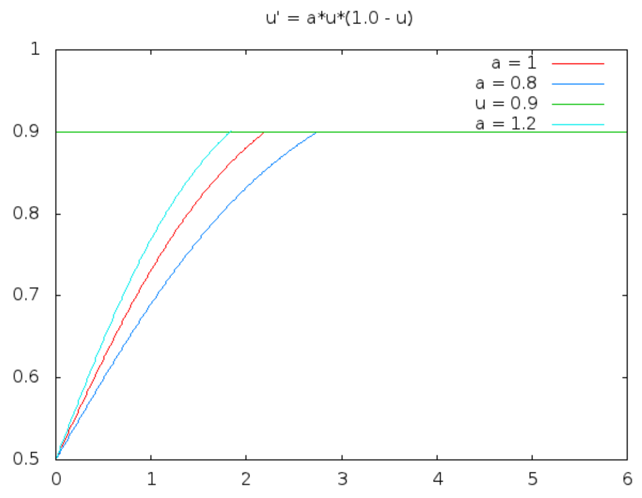


Figure 2.3: Stop events in logistic population model

2.2.3 Example 2: Van der Pol oscillator problem

Basic usage with nonscalar ODE system

Let us recall the Van der Pol oscillator problem, which we used to illustrate the inconvenience to switch between different ODE software in Section 1.2.2.

$$u'' = 3(1-u^2)u' + u \tag{2.6}$$

$$u_0 = (2.0, 1.0) \tag{2.7}$$

$$\text{time\_points} = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0] \tag{2.8}$$

We can solve this problem as simple as in scalar ODEs:

```
import odesolvers
import numpy

def f(u,t):          # Define ODE system with vectors
     u_0, u_1 = u
     return [u_1, 3.0*(1.0 - u_0*u_0)*u_1 + u_0]
u0 = [2.0, 0.0]       # Specify initial value as vector
time_points = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

method = odesolvers.RungeKutta4(f)
method.set_initial_conditions(u0)
u,t = method.solve(time_points)
```

Switch to another solver

Suppose we want to try another solver that are more advanced and reliable (like Lsode in ODEPACK), and compare the solution values returned from these two solvers.

In Section 1.2.2, we have described how troublesome to switch inbetween different ODE software for this ODE problem.

With help of *odesolvers*, this job can be done in only two additional lines.

```
# A new instance applying method Lsode
method_new = method.switch_to(odesolvers.Lsode)

# Restart integration
u_new,t_new = method_new.solve(time_points)
```

2.3 Unified features for future developers

Our *odesolvers* aims to be a general interface for all ODE solvers. So I hope to make it easy-to-extend through a simple interface to future developers with a minimal effort.

Figure 2.4 illustrates the simplified interfaces of *odesolvers* towards the ordinary users ('Users') and future developers ('Developers'). The relevant functions of 2.4 are organized in several groups by their usage frequency from high to low.

Other details of implementation, should be hidden backward as black boxes, thus can be safely ignored by not only ordinary users but also future developers.

2.3.1 Consistency problem with parameters in different solvers

As description in Section 1.2.2, a same parameter (like error tolerance) can be defined with different names given by different developers. This will lead to inconsistency in our interface, and make it confusable both for users and code readers.

Furthermore, different ODE solvers may have different forms or requirements on using the same parameter. For example, in some traditional ODE solvers, function *f* is often supplied with a parameter list *(t, u)*. However, in other ODE software, *f* might be supplied with a parameter list *(u, t)*. If the users try to switch between some solvers, the different orders of parameters in the input list could make the uses of these solvers incompatible, and thus lead to errors.

2.3.2 *_parameters* : A complete dictionary for parameter properties

- A dictionary *_parameters* is defined as global variable in *odesolvers*, and contains general information about property settings of all the possible parameters in the whole hierarchy.

```
_parameters = dict(

    adda = dict(
        help = '''\
        User-supplied subroutine which adds the matrix A = A(t,y)
        to another matrix stored in the same form as A.''',
        type = callable),

    atol = dict(
        help='absolute tolerance for solution',
        type=(float,list,tuple,numpy.ndarray),
        default=1E-8),

    beta = dict(
        # intended for adaptive methods like dopri5 or dop853
        help='Beta parameter for stabilized step size control',
```

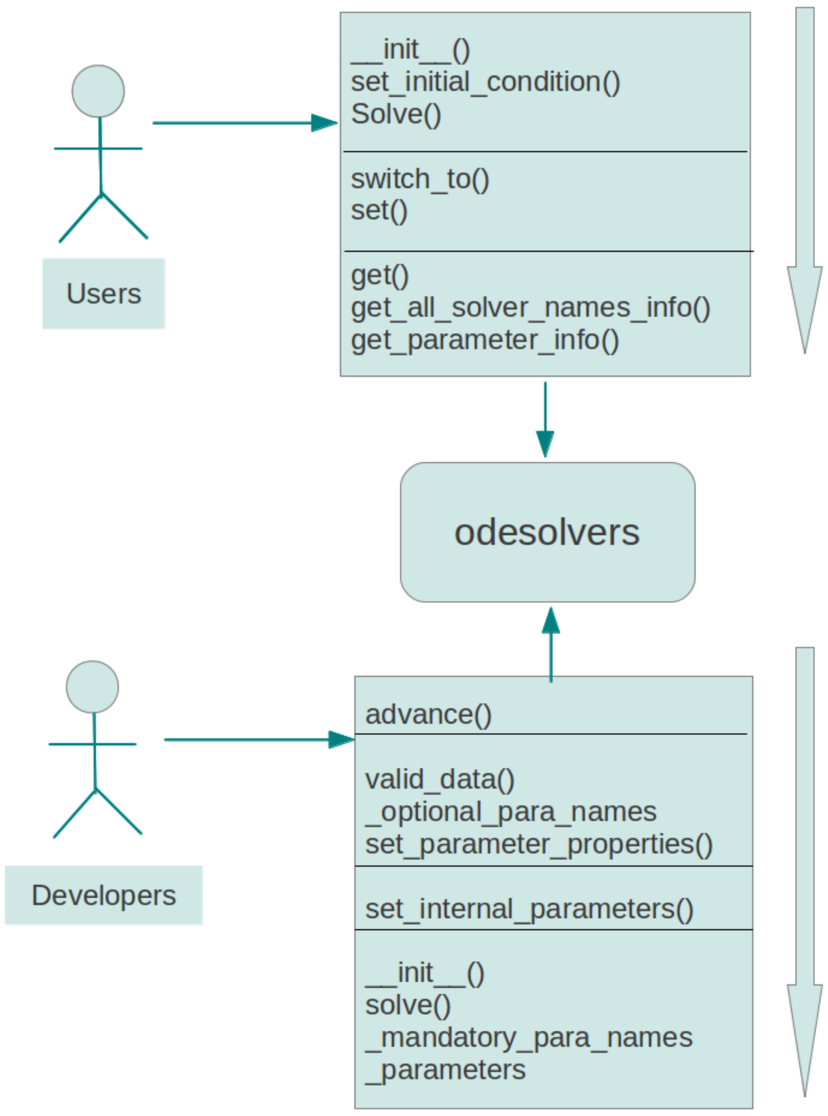


Figure 2.4: Interfaces for users and future developers

```
        type=float),
    ...
    )
```

- Two string lists *_optional_para_names* and *_mandatory_para_names* are defined in each class, and contain the names of all the valid optional input parameters and mandatory input parameters respectively.

  For example, in superclass *Solver*, these two string lists are defined as the following:

  ```
  _required_parameters = ['f',]
  _optional_parameters  = ['f_args','f_kwargs','complex_valued']
  ```

  In subclass *Adaptive(Solver)*, i.e. the superclass of all adaptive methods, two arguments *'rtol'* and *'atol'* are introduced as new optional arguments with no need to set up their properties, for instance, type, help message, default value, etc. All these information can be extracted from the global dictionary *odesolvers._parameter*.

  ```
  _optional_parameters = ['rtol', 'atol']
  _optional_parameters.extend(Solver._optional_parameters)
  ```

- In the start of initialization step, properties of valid input parameters in the current class are extracted from *_parameters*, and stored as a new dictionary variable *self._legal_paras* in the instance of the current solver.

  ```
  def __init__(self, **kwargs):

      self._legal_paras = dict(
          (key,value.copy()) for key, value in _parameters.items()
          if (key in self._optional_parameters) or \
             (key in self._required_parameters))
  ...
  ```

- If some unusual input parameters are to be introduced in the new solver, developers should carefully search and try to select appropriate parameter names from the existing items in *_parameters*. If none of existing items in *_parameters* is appropriate, new items should be supplemented into the global dictionary, and then add their names into *_optional_para_names* (or seldom *_mandatory_para_names*).

- In this way, the consistency of name and properties for all existing parameters is assured. And in subclasses of *odesolvers*, only those valid input parameters are accessible.

  Let us look at an interactive Python session to demonstrate the usage of parameter-property dictionaries:

  ```
  >>> import odesolvers
  >>> print odesolvers.ODE._parameters.keys()
  ['f_kwargs', 'atol', 'adda_lsoibt', 'method_order', 'jac_column_tu', 'jac_lsodi', 'jac_kwargs', 'mb', 'mya
  >>> print odesolvers.ODE._parameters['atol']
  {'default': 1e-08, 'type': (<type 'float'>,
  ```

```
 <type 'list'>, <type 'tuple'>, <type 'numpy.ndarray'>),
 'help': 'absolute tolerance for solution'}

>>> method = Adaptive(lambda u,t:u)   # Instance initialization
>>> method._optional_parameters
['rtol', 'atol', 'f_args', 'f_kwargs', 'complex_valued']
>>> method._required_parameters
['f']

>>> import pprint
>>> print pprint.pformat(method._legal_paras)
{'atol': {'default': 1e-08,
          'help': 'absolute tolerance for solution',
          'type': (<type 'float'>,
                   <type 'list'>,
                   <type 'tuple'>,
                   <type 'numpy.ndarray'>)},
 'complex_valued': {'default': False,
                    'help': 'True if f is complex valued',
                    'type': <type 'bool'>},
 'f': {'help': 'right-hand side f(u,t) defining the ODE',
       'type': <built-in function callable>},
 'f_args': {'default': (),
            'help': 'extra parameters to f:
                     f(u,t,*f_args,**f_kwargs)',
            'type': (<type 'tuple'>,
                     <type 'list'>,
                     <type 'numpy.ndarray'>)},
 'f_kwargs': {'default': {},
              'help': 'extra parameters to f:
                     f(u,t,*f_args,**f_kwargs)',
              'type': <type 'dict'>},
 'rtol': {'default': 1e-06,
          'help': 'relative tolerance for solution',
          'type': (<type 'list'>,
                   <type 'tuple'>,
                   <type 'numpy.ndarray'>,
                   <type 'float'>)}}
```

Property-information of all the legal input parameters in *Adaptive* (namely *atol, rtol, f, f_args, f_kwargs, complex_valued*), are extracted and stored in instance variable *method._legal_paras*.

In next section, we can see how flexibly we can make use of these properties, both for validity checking and value transforming.

2.3.3 Automatic check and transform with flexible property-settings

In dictionary *self._legal_paras*, each item is defined as a sub dictionary to describe properties of a specific parameter.

For each parameter, there are 9 kinds of properties can be defined by developers:

type A list of possible types for this parameter.

Function *check_input_types()* will check automatically whether parameters are in right types according to this property. For example,

```
_legal_paras['f']['type'] = callable
_legal_paras['atol']['type'] = (float,list,tuple,numpy.ndarray)
```

where *atol* can be a scalar or sequence.

default The default value if this parameter is not specified by users.

Function *__init__()* will set the parameter with default value. For example,

```
_legal_paras['atol']['default'] = 1e-8
```

range A list of possible values, or a range in form of (low,high).

Function *check_input_ranges()* will check whether this parameter is in right range. For example,

```
_legal_paras['theta']['range'] = [0,1]
        _legal_paras['nonlinear_method']['range'] = ('Picard','Newton')
```

extra_check A function return boolean value for extra value check.

Function *check_extra()* will call the corresponding function to check this parameter. For example,

```
_legal_paras['yoti']['extra_check'] = \
        lambda float_sequence: numpy.asarray(\
                map(lambda x: isinstance(x, float), float_sequence)).all()
```

where yoti should be a sequence of floats.

help Short description for this parameter to be printed out for explanation.

condition_list Some relevant parameters need to be input when this parameter is input with a specific value.

Function *check_conditional_parameters()* will check whether conditional parameters are input sufficiently according to value of this parameter.

```
_legal_paras['iter_method']['condition\_list'] = \
     {'1':(('jac','jac\_fortran'),); ...; '5':('ml','mu')]}
```

When *iter_method* is set to 1, either *jac* or *jac_fortran* need to be supplied; when *iter_method* is 5, both *ml* and *mu* need to be supplied.

'returnArrayOrder', 'paralist_old' and 'paralist_new' These properties are used in function *func_wrapper()*, and will be described in next section.

As long as developers define these properties with proper values, the relevant functions would check the value of input parameters automatically, according to the property-settings in *self._legal_paras*, as in Figure 2.5.

2.3.4 *func_wrapper()*: Transform user-supplied function to desired form

Function *func_wrapper()* is used to handle the following three types of problems.

Incompatible order in parameter list

For user-defined functions in modern ODE solvers, the usual order of input parameter list is to start with *'u,t,...'*; however in some other ODE software (like ODEPACK), the input parameter list is vice versa, i.e. often they are required to start with *'t,u,...'*.

For any user-supplied function, we hope to hold the parameter list in a unified form to start with *'u,t,...'* in *odesolvers*. This unified feature not only simplify user interface, but also make it compatible when switching inbetween different solvers. Thus in order to make use of some underlying software, we need to wrap user-supplied functions with parameter lists in different orders, e.g. *jac(u,t) –> jac(t,u)*.

Incompatible array-index between Fortran and Python

In wrappers for Fortran software, because arrays start at different indices in Fortran and Python, sometimes we need to make special wrapping for this incompatible feature.

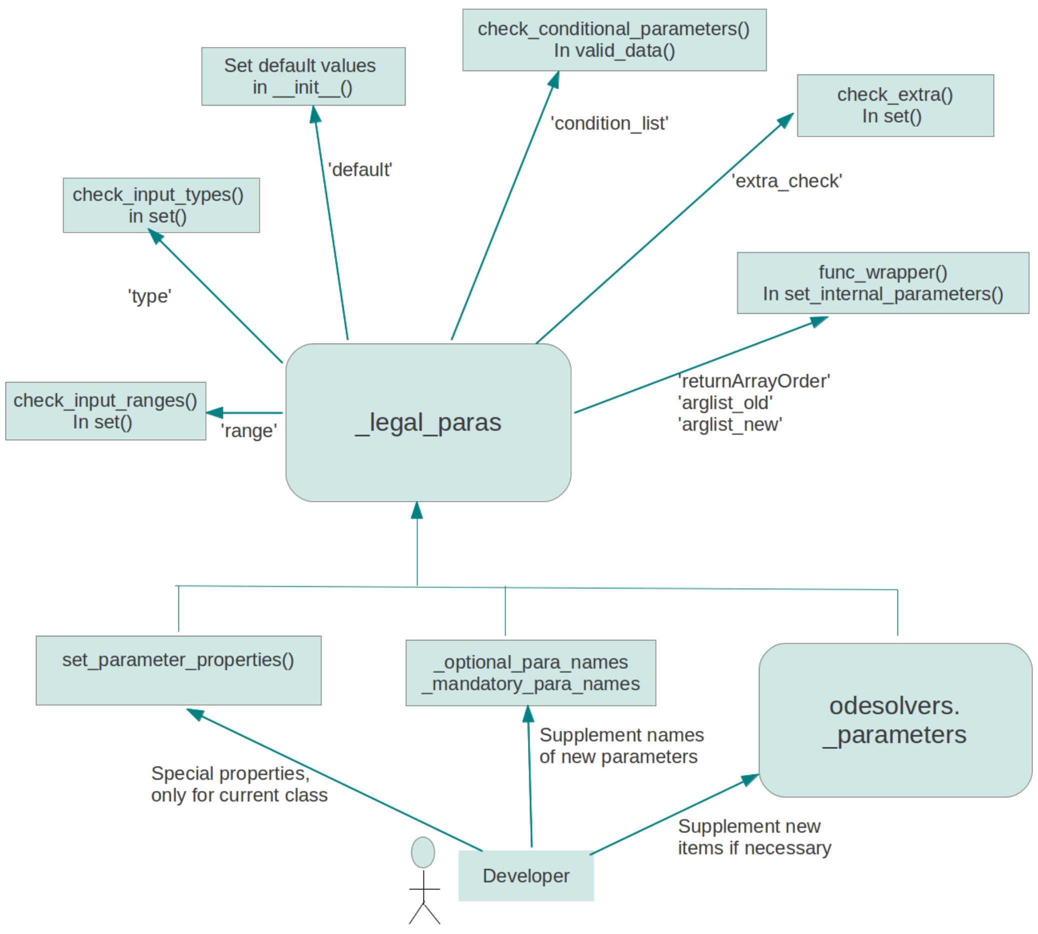


Figure 2.5: Property settings in _legal_paras

returnArray-Order	paralist _old	paralist _new	New form of wrapped function
None	'u,t'	't,u'	lambda t,u: jac(u,t)
C	'u,t'	None	lambda u,t: numpy.asarray(jac(u,t))
C	'u,t'	't,u'	lambda t,u: numpy.asarray(jac(u,t))
Fortran	'u,t'	None	lambda u,t: numpy.asarray(jac(u,t),order='Fortran')
Fortran	'u,t'	't,u'	lambda t,u: numpy.asarray(jac(u,t),order='Fortran')

Table 2.1: Functions are wrapped automatically

For instance, for a user-defined function *jac_column(t,u,column_index)* in subroutine *dlsodes* in ODEPACK, *column_index* is a parameter with an automatic value in Fortran code. In Fortran language, *column_index* will get a value starting from 1; however in Python, this parameter starts from 0 instead. Our Python users will take this assumption when they code this function in Python. This is why we need to wrap the parameter list of user-defined *jac_column* from *'u, t, column_index'* to *'u, t, column_index+1'*. That is, define the Jacobian function as *'lambda u, t, column_index: jac_column(u, t, column_index-1)'*.

Store arrays in Fortran order for better efficiency

The return value of user-defined functions should be wrapped to Numpy arrays with great numerical features, e.g. vectorization and array slicing.

Furthermore, in order to avoid unnecessary array copy by F2PY, it is always recommended to explicitly transform all multi-dimension Numpy arrays to Fortran ordering in the Python code. See a detailed discussion around this topic in [3].

Thus we should try to wrap return values of user-supplied functions to be Numpy arrays in appropriate orders as much as possible (e.g. *lambda u,t: numpy.asarray(jac(u,t), order='Fortran')*).

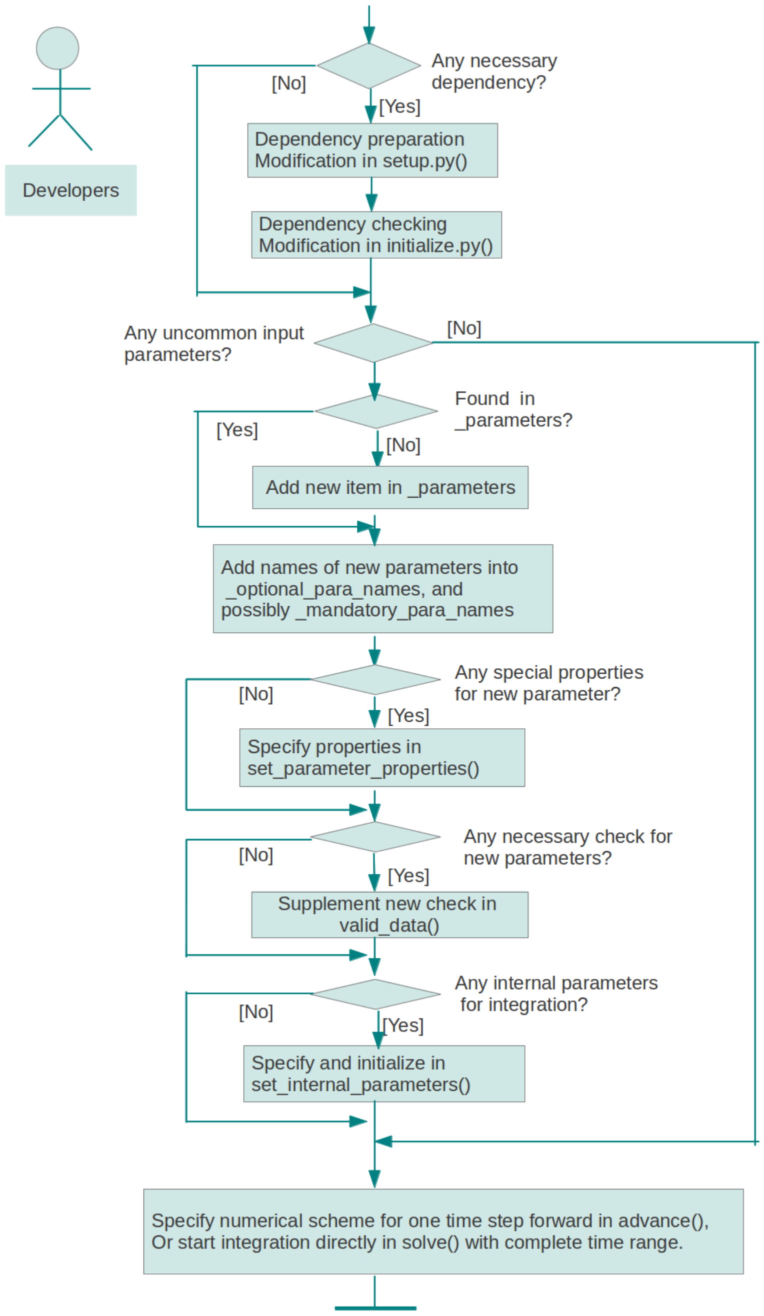
Make use of 3 properties *paralist_old*, *paralist_new* and *returnArrayOrder*, function *func_wrapper()* would wrap user-supplied functions to desired forms automatically.

Table 2.1 demonstrates in which form the function with different properties would be transformed.

Complete code of function *func_wrapper()* is attached in Appendix B.3.

2.3.5 Normal routine to integrate a new solver

In the previous sections, how to extend *odesolvers* with a new solver is introduced. Now I would like to describe a normal routine on how to integrate a new solver into *odesolvers*:



Developers
Any necessary dependency?
[No]
[Yes]
Dependency preparation Modification in setup.py()
Dependency checking Modification in initialize.py()
Any uncommon input parameters?
[No]
Found in _parameters?
[Yes]
[No]
Add new item in _parameters
Add names of new parameters into _optional_para_names, and possibly _mandatory_para_names
Any special properties for new parameter?
[No]
[Yes]
Specify properties in set_parameter_properties()
Any necessary check for new parameters?
[No]
[Yes]
Supplement new check in valid_data()
Any internal parameters for integration?
[No]
[Yes]
Specify and initialize in set_internal_parameters()
Specify numerical scheme for one time step forward in advance(), Or start integration directly in solve() with complete time range.

Figure 2.6: Normal routine to integrate a new solver

1. Dependency preparation.

   This step is involved only when developers intend to wrap an existing package into *odesolvers*.

   If the original package is not written in Python language, developers need to apply some specific tools (like F2PY or SWIG) to create an extension module to make the package accessible from our Python code.

   Otherwise, if the original package is also a Python software, developers need to install and import the desired package as a Python module.

   By an attempt to import these necessary modules (often set in method *initialize()*), we can check whether the necessary dependencies are installed properly.

2. Definition of legal parameters and their properties

   Each solver has a set of specific parameters depending on its underlying method. For example, adaptive solvers will be more likely to apply attributes for step control, like *first_step, min_step, max_step*. And a collection of methods probably need to provide a parameter for users to make method choice, like *ode_method*.

   As described in Section 2.3.2, developers should try to search in dictionary *_parameters* for suitable items to represent the desired parameters. If there is no suitable items found in this dictionary, developer need to supplement new items in it.

   There is no need to define input parameters one by one. With the help of variable *_optional_para_names* and *_mandatory_para_names*, all parameters with names in these two name lists would be considered as legal in new solver.

   Furthermore, if a parameter in new solver has some properties different from general settings in *_parameters*, developers can reset(or supplement) these properties in function *set_parameter_properties()*.

3. Special check in *valid_data()*

   For some complicated solvers with many relevant input parameters, there are possibly special relationship requirements inbetween some specific input parameters.

   For example, in class *odesolvers.Lsodes*, there are special requirements for the values of two input integer arrays *ia* and *ja*:

   - *ia* and *ja* must be input simultaneously.
   - len(*ia*) == *neq* + 1
   - *ia[neq]* = len(*ja*)

   Most of the automatic checking are taken in initialization step. We need to take extra check for the above requirements after all the inputs are initialized. Thus a new function *check_iaja()* is defined in *odesolvers.Lsodes*, and injected into function *valid_data()*.

4. Internal settings in *set_internal_parameters()*

   When I tried to wrap some complicated ODE software, some parameters are found to be dependent on values of other parameters, although they are required as inputs for these underlying ODE software.

   For example, as an input parameter for *rkc.f*, *info[1]* is an integer flag to indicate whether function *spcrad* is supplied by users.

   This kind of parameters are required by underlying software, but unnecessary to be valued from user's input. This is why I called them as internal parameters.

   Function *set_internal_parameters()* is used to initialize this kind of parameters before they are passed to the underlying software.

5. Step forward in *advance()*

   In function *advance()*, solution value for next time point should be returned. This is the only mandatory step to implement a new solver.

   For simple numerical methods (like ForwardEuler method), numerical scheme is implemented directly in this function. If the new solver is a wrapper to another module(either Python module, or extension module), iteration in underlying package will be ready to start if Python code pass all the necessary parameters to the underlying module according to its user interface.

   In the user interfaces of some ODE software, like *sympy.mpamath.odeint*, solving procedure is started directly with the whole sequence of time points, but not step by step. Then developers should turn to start iteration directly in function *solve()*.

For a simple example, standard Runge-Kutta3 method is implemented in only several lines:

```
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
```

In the next chapter, implementation procedure of ODEPACK would be described in details. Readers can refer it as another implementation example and get a more comprehensive view for integration of complicated solvers.

2.4 Why *odesolvers*?

Generally, there are three different kinds of users for ODE software: 1. novices (like students) who are new to ODE; 2. scientists (like biologists) who know much about ODE and modelling, but has limited experience with programming; 3. professional researchers who are familiar both in ODE methods and programming.

From my point of view, *odesolvers* has many attractive features to make it stand out from other ODE software and attractive for all kinds of users, especially for the novices and scientists without everyday programming experience.

Completeness
odesolvers is probably one of the most complete collection of numerical methods for solving ODE initial value problems. We can readily refer this point to 1.2. Novices can easily apply classic numerical methods for simple ODE problems, and complicated ODE solvers are also available for scientists and professional researchers.

Simple user interface
User interface is designed in a minimal sense to simplify its usage, and enough to define all kinds of ODE systems. This feature is not only attractive for novices, but also very helpful for scientists without much programming experience. For example, in chapter 3, we will see how much the usage of ODEPACK can be simplified with extra pre-check and automatic error-handelling. Even for very experienced programmers, this feature would be attractive to save their programming time.

Free-of-charge and Easy-to-install
odesolvers is implemented as an open-source software and an independent package in Python. The only mandatory dependencies are Numpy and Python, with some other optional dependencies for those wrappers. That is, users can install this package, and make use of many basic solvers without any extra installation. Many sophisticated packages are often complained for troublesome installation with a lot of dependencies.

Great potential for future development As described in Section 2.3.5, this package provide great extensibility for future developers. I believe this would attract developers to extend the current package with more ODE solvers in the future.

Easy-to-join with other Python tools Currently no additional functionalities (like plotting, trajectory) are included in this package. However, in *odesolvers*, the desired solution is returned as a standard Numpy array. Users can trivially apply their favored Python libraries for data analysis.

An suitable facility for university courses This interface can be used as the entry point for new comers of ODE who want to implement a numerical ODE method or test some simple ODE problems. Students can define and test their own solver easily with the clean syntax, and make use of general features in this package.

Chapter 3

Odepack

Firstly, to avoid the possible confusion in terms, in this thesis we use:

- 'ODEPACK' for the original Fortran package;
- '*Odepack*' for the wrapper collection in odesolvers;
- '*dlsode*' and *'dlsode*', which are the original subroutine names in Fortran code, for the Fortran subroutines;
- '*Lsode*' and '*Lsoda*', the class names in odesolvers, to represent the Python wrappers for *dlsode* and *dlsoda* respectively.

In this chapter, I will focus on integration of solvers in ODEPACK, which is a very important contribution of this project. The design issues and difficulties in its implementing procedure and its attractive features will be described.

3.1 Background knowledge

ODEPACK is a widely-used ODE package that consists of 9 Fortran solvers for the initial value problem. The ODE systems can be given either in explicit form,

$$\frac{du}{dt} = f(t,u)$$

or in linearly implicit form,

$$A(t,u)\frac{du}{dt} = g(t,u)$$

The ODEPACK solvers are written in the standard Fortran 77 since its first version in 1983. Fortran 77 was primarily intended for calculations, and hence remains as one of the best computational languages ever developed, and is still widely in used for large computing problems in scientific field. Taking advantage of Fortran 77, ODEPACK is generally considered to be an efficient solver with high reliability in computation.

On the other hand, the main disadvantages of ODEPACK are also associated with the relative antiquity of the underlying language. For instance, input/output facilities are fairly limited comparing with other ODE software. In Fortran 77, subroutines pass parameters by reference rather than by value. For example, when I was totally new in Fortran 77 and trying to read Fortran code in ODEPACK, I was completely confused when I found that an item in a single-dimension array was passed to another subroutine as a multi-dimension array. In Fortran 77, array-dimensions are often required to be explicitly defined in parameter list to illustrate the sizes of array parameters. This is obviously very different with flexible syntax in modern languages like Python.

Due to by-reference parameter-passing and strict type-safe feature, subroutines in ODEPACK always demand users to put in many parameters in strict orders and requirements, which cause frequently run-time errors with weird messages to users. For example, if a user forget to list f as the first parameter for any ODEPACK solver, an interrupt will occur with the following confusing message:

```
Warning: error exceeds 100 * tolerances
DLSODES- ISTATE(=I1) illegal.
      In above message, I1 =  -3
DLSODES- Run aborted. . Apparent infinite loop.
```

The above error message above is definitely incomprehensible for users without extensive experience with the errors typically encountered in ODEPACK.

For another example, if users define f with a wrong parameter list by fault, they will get an immediate interrupt from ODEPACK, accompanying only two annoying words:

```
Segmentation fault.
```

In order to solve this kind of errors, users need to dig into the documentation (and possibly detailed code) of ODEPACK carefully, and try to figure out what is wrong with their input. Even for some professional users, it could take several days to fix a simple typo.

Despite these shortcomings, ODEPACK is still widely-used in scientific computing, and is regarded as one of the most important ODE packages. Therefore, there is an increasing demand of Python wrappers for ODE-PACK.

Up till now, it seems that only two solvers (*dlsode* and *dlsoda*) in ODEPACK are well-known to be wrapped into Python. Wrapper for *dlsode* is available in an ODE interface at Lawrence Livermore National Labs, and the other is module *scipy.integrate.odeint* for *dlsoda*. It will be very helpful for users to have a Python interface that is accessible to other solvers in ODEPACK.

3.2 Implementation procedure

Let us recall the normal developing routine, which is suggested in Section 2.3.5:

- Dependency preparation.

- Definition of legal parameters and their properties
- Special check in function *valid_data()*
- Internal setting in function *set_internal_parameters()*
- Step forward in *advance()* or start iteration directly in *solve()*

The above routine is basically following in my implementation. However, in order to achieve the best performance and efficiency, I have tried several ways with different emphases.

3.2.1 First version: one step forward through a direct call to Fortran subroutines in *advance()*

Subroutines in ODEPACK return the solution value at the next time point, as same as what we need in function *advance()*. Naturally, I decided to include these subroutines directly in an extension module, and make a call straight from *advance()* for one step forward. As long as the extension module is generated in a matchable form with our interface, we can call the Fortran solvers directly with required parameters in *advance()*, and get the solution value at next time point.

Dependency preparation

In this section, I took an assumption that readers have basic knowledge and perhaps programming experience with F2PY – Fortran to Python interface generator. In order to call Fortran subroutines from our Python code, F2PY is used as a powerful tool to build an extension module. This section describes the details about how to build the extension module and make necessary modification in the signature file for this module. If readers do not intend to access Fortran subroutines, you may just skip to the next part.

F2PY is widely used as the connection between Fortran and Python codes, and provide many great features like automatically handling different storage order between Python and Fortran, automatically generating signatures of Fortran subroutines according to the original Fortran code [5].

F2PY claims to be able to generate extension module for Fortran subroutines almost automatically. Unfortunately, it is not so easy in my wrapping procedure for ODEPACK.

The problem is, we need to generate the extension module in a pythonic sense and optimize it to match perfectly with our Python interface.

For example, in this Fortran package, user-provided functions are always required with much longer parameter lists comparing with the parameter lists of same functions in *odesolvers*.

In the basic solver *dlsode*, subroutine *JAC* is used to define either full Jacobian matrix or banded Jacobian matrix:

```
SUBROUTINE JAC (NEQ, T, Y, ML, MU, PD, NROWPD)
INTEGER  NEQ, ML, MU, NROWPD
DOUBLE PRECISION  T, Y(*), PD(NROWPD,*)
```

where *ML* and *MU* represent the lower- and upper-band of banded Jacobian matrix respectively, and *NEQ* and *NROWPD* are used to define the size of array *Y* and *PD*.

On the other side, in our Python interface *odesolvers*, Jacobian matrix is usually defined in form of full matrix. Furthermore, in *odesolvers*, all user-supplied functions should have their parameter list starting with order *'u,t'* to keep the unified feature. Hence the general form for *jac* in *odesolvers* is as simple as *jac(u,t)*, and *jac(u,t,ml,mu)* should be sufficient for banded Jacobian matrix.

I hope to generate an extension module which contains interfaces for all the desired solvers in ODEPACK, with pythonic signatures for their callback functions.

1. Download source code and generate a signature file automatically.

   Firstly, we need to download the source code of ODEPACK from Netlib repository [23]. The Fortran package consists of three source files, namely *opkdmain.f*, *opkda1.f* and *opkda2.f*.

   The signature file for ODEPACK can be generated with a single line :

   ```
   F2PY -m _odepack -h odepack.pyf –overwrite-signature opkdmain.f
   ```

2. Hide unnecessary input parameters as many as possible

   In F2PY, with an extra mark *'intent(hide)'* or *'optional, depend(...)'*, parameters can be removed from the mandatory input list, and the signature of callback functions will be definitely simpler and more pythonic.

   As described above, in ODEPACK, there are always many input parameters for subroutines and their callback functions. In the long parameter lists for callback functions, many parameters can be treated as optional or even hidden in Python.

   For example, variable *neq* (Number of equations) is required as the first parameter for user-defined function *f* in ODEPACK. This is reasonable in Fortran language with a pass-by-reference feature. But for the callback functions in Python, it is totally unnecessary and can be safely ignored.

3. Explicitly classify output parameters of Fortran subroutines and callbacks.

   In Fortran language, all parameters of subroutines are used as both input and output value, while in Python functions return a tuple of variables as output value. This difference makes it impossible for F2PY to wrap Fortran code without explicitly documenting which parameters are specially for input and which parameters are used for output.

   For example, the desired solution value should be returned in a variable *y* from Fortran. But in the automatically generated interface, it is treated as an ordinary input parameter without any comments:

```
        double precision dimension(*) :: y
```

We need to comment the signatures of these variables explicitly:

```
        double precision dimension(*), intent(in,out) :: y
```

4. Supplement signatures for callback functions which are not called in local subroutine

   This kind of callback functions are defined in the parameter list of a specific Fortran subroutine, and passed to other associate subroutines without any directly call in local subroutine. Then it is an impossible mission for F2PY to detect signatures of these external callback functions through compiling local subroutine.

   For example, in the basic solver *dlsode*, there are two external function parameters (*f* and *jac*), which are passed to another associate subroutine *dstode* in *opkda1.f*

   In the F2PY-generated .pyf file, interface of *dlsode* is defined as:

```
python module _odepack ! in
   interface ! in :_odepack
      subroutine dlsode
(f,neq,y,t,tout,itol,rtol,atol,itask,istate,iopt,rwork,lrw,iwor
k,liw,jac,mf) ! in :_odepack:opkdmain.f
         use dlsode__user__routines
         external f
         external jac
         ...
         end subroutine dlsode
         ...
   end interface
end python module odepack
```

   Both *f* and *jac* are directed to *dlsode__user__routines* as external functions. That is, when F2PY applies this signature file to generate extension module, F2PY will try to search for their signatures in *dlsode__user__routines*:

```
python module dlsode__user__routines
   interface dlsode_user_interface
      subroutine f(neq,t,y,e_rwork_lf0_e)
         ...
      end subroutine f
   end interface dlsode_user_interface
end python module dlsode__user__routines
```

   The absence of *jac* in this user-interface will lead to a runtime-error:

```
        error: 'jac' undeclared (first use in this function)
```

   We need to supplement signature of *jac* explicitly and try to define it in a pythonic form (like *jac(u,t)*) just as in *odesolvers*.

```
subroutine jac(neq,t,y,ml,mu,pd,nrowpd)
    in :_odepack:opkdmain.f:dlsode:unknown_interface
    integer dimension(*),intent(hide) :: neq
    double precision :: t
    double precision dimension(neq[0]) :: y
    integer intent(hide) :: ml
    integer intent(hide) :: mu
    integer intent(hide) :: nrowpd
    double precision dimension(*,neq[0]), intent(out) :: pd
 end subroutine f
```

Note that we can never modify the fixed form of parameter list in Fortran code. We can only try to make it more pythonic in several ways: 1) hide unnecessary input parameters like *neq*, *ml, mu, nrowpd*; 2) classify output parameters explicitly.

In Python, we can extract and print out this new signature:

```
def jac(t,y): return pd
Required parameters:
  t : input float
  y : input rank-1 array('d') with bounds (neq[0])
Return objects:
  pd : rank-2 array('d') with bounds (*,neq[0])
```

That is, in order to pass *jac* smoothly as the desired callback function for *dlsode*, it is required to be defined in form of *jac(t,y)*, which is still incompatible with the general form *jac(u,t)* in our *odesolvers*.

This is exactly why we need to wrap this kind of user-defined functions with new parameter lists in different orders, as we described in Section 2.3.4.

Moreover, when *jac* is used to define banded Jacobian matrix, the hidden parameters *ml, mu* should be supplemented as extra parameters for *jac*.

5. Modify F2PY-generated signatures for other callback functions

   For the other callback functions that are called directly, their parameter types and dimension might be unclear or faulty in the automatically F2PY-generated interface. In F2PY, this kind of signatures are guessed according to the form in which these functions are called firstly in Fortran code.

   Checking the code lines of *dlsode* in ODEPACK, we can find the first call of *f* in line 1391:

   CALL F (NEQ, T, Y, RWORK(LF0))

   The corresponding signature of *f* is :

```
subroutine f(neq,t,y,e_rwork_lf0_e)
   in :_odepack:opkdmain.f:dlsode:unknown_interface
   integer dimension(*) :: neq
   double precision :: t
   double precision dimension(*) :: y
   double precision :: e_rwork_lf0_e
end subroutine f
```

   In order to make this signature suitable for the general form *f(u,t)* in our Python interface, there are several issues to be modified: 1) hide unnecessary parameter *neq*; 2) change the weird parameter name *e_rwork_lf0_e* to be a suitable one – *ydot*; 3) explicitly mark the output variable with '*intent(out)*'; 4) edit the dimension of *y* to specify its length requirement.

   Both of the original F2PY-generated signature and the modified version is listed in Table 3.1.

Original F2PY-generated signature	Modified signature
subroutine f(neq,t,y,e_rwork_lf0_e) integer dimension(*) :: neq double precision :: t double precision dimension(*) :: y double precision :: e_rwork_lf0_e end subroutine f	subroutine f(neq,t,y,ydot) integer dimension(*),intent(hide) :: neq double precision :: t double precision dimension(neq[0]) :: y double precision dimension(*), intent(out) :: ydot end subroutine f
f(neq,t,y,e_rwork_lf0_e) –> neq,t,y, e_rwork_lf0_e	f(t,y) –> ydot

Table 3.1: Signatures of callback *f*

Finally, after all the above issues are handled in signature file, the extension module *_odepack* is ready to generate:

```
F2PY -c odepack.pyf opkdmain.f opkda1.f opkda2.f
```

This step is one of the most time-consuming and struggling parts in my project, since a lot of components are involved in this process:

- 7 Fortran subroutines
- 128 parameters for these seven subroutines
- 17 callback functions
- 96 parameters for these callback functions.

In order to optimize this extension module to be more pythonic, I have to dig into ODEPACK documentation and sometimes even Fortran code in associate subroutines, learn the details of these components and think about many questions: Is this parameter mandatory? optional? Or dependent? Which parameters are input and output for this callback function? How much can I simplify the parameter list for this subroutine?

Definition of legal parameters and their properties

Among all the components of *odesolvers*, solvers in ODEPACK are the most complicated solvers considering their long lists of parameters and many callback functions.

- Select input parameters as few as possible for current solver. Put in names of these parameters respectively in *self._optional_para_names* and possibly *self._mandatory_para_names*.
- 23 new parameters are introduced to *odesolvers*, and hence their general property settings (like type, help message) are supplemented in the global dictionary *_parameters*.

- Define property *range, condition_list, extra_check* for value check in the initialization step.

- Define property *paralist_old* and *paralist_new* for all callback functions in these solvers.

  In ODEPACK, all the user-defined functions are required to supply their parameter lists starting with *'t,u'*, which are in the contrary order of parameter lists in *odesolvers*. In order to keep the consistent form in *odesolvers*, we should not require users to change this order. As long as *paralist_old* and *paralist_new* are set with proper values, function *func_wrapper()* will finish this job automatically.

- Define property *returnArrayOrder* for all callback functions which return multi-dimension arrays, as we described in Section 2.3.4. This step is very important for a better efficiency.

  In order to make wrappers for Fortran software, wrapping the return value from user-defined functions in Fortran order storage would avoid unnecessary array copy and hence improve efficiency performance. [28]

  Numpy arrays apply the storage scheme for C arrays, i.e. row major storage, while Fortran store arrays column by column. In order to keep the storage scheme transparent, F2PY need to copy these arrays to new ones with column major storage in the wrapper code. That is, each time the user-defined functions is called back from Fortran code, the returned value in C storage order would be copied to a new array in column major storage. Comparing with the cost of repeatedly creation of these new copies, the cost for transforming arrays to Fortran storage in Python can be safely omitted, especially for the complicated ODE system with arrays in big sizes.

  Compiling with *-DF2PY_REPORT_ON_ARRAY_COPY=1*, we can get informed for each time an array is copied. For example, in the developing process of *Odepack*, I tried to explicitly change the storage order of banded Jacobian matrix returned from user-defined function *jac_banded* from 'Fortran' to 'C'. Applying *odesolvers.Lsode* to solve the same ODE problem as in Section 5.1, the number of array copies increased from 3 to 42 significantly.

In this step, all the property settings are defined in function *set_parameter_properties()*, as we illustrated in Figure 2.5

Special check in *valid_data()*

In ODEPACK, special requirements for relationship among parameters need to be injected in to function *valid_data()*. For example, as a pair of integers to describe the lower-band and upper-band of banded Jacobian matrix, *ml* and *mu* must be supplied simultaneously by users.

Internal settings in *set_internal_parameters()*

In this step, quite a few mandatory parameters for Fortran subroutines are initialized and get valued depending on values of other parameters. For simplest example, *itol* is a flag to indicate whether relevant tolerance and absolute tolerance are input as scalars or sequences. This can be easily detected in Python with checking types of *rtol* and *atol*.

Step forward in *advance()*

Fortran solvers are called directly in this function after the extension module is imported. All the required parameters (some are supplied by users, while others are initialized as internal parameters) need to be passed according to the strict orders defined in extension module.

Odepack: A new collection for solvers in ODEPACK

Following the above development routine, when I finished wrapping three solvers in ODEPACK separately, I found that there are many duplicated or similar parts in common for these wrappers. That was why I defined an 'abstract' class *Odepack(Solver)* as the superclass for these solvers in ODEPACK. Common methods were moved into this superclass as much as possible. This simplified and shortened my original Python code significantly.

15 ODE problems are implemented as test cases for these solvers. All the test cases are proved to be correct and smoothly. Everything seems fine, except a critical disadvantage – poor efficiency, especially when long sequence for time_points is supplied.

Figure 3.1 illustrates its efficiency loss comparing with *scipy.integrate.vode* and the original Fortran package.

The test problem is $u' = A * u$, where number of equation is 25, and A is a banded lower triangular matrix derived from 2-dimension advection PDE.

The result for this efficiency test is really frustrating. When I tried another test with 10000 time steps, it took 185 seconds to get the same result which was returned in 0.03 second from the original Fortran package.

3.2.2 Second version: a wrap-subroutine in Fortran part

In order to achieve a better efficiency, I made a try to move time-consuming job of *advance()* to the Fortran part, i.e. the initialization of work arrays.

In the first version, work arrays (*rwork* and *iwork*) are initialized in Python, and commented as '*intent(in, out)*' to pass the optional input and output information. However in ODEPACK, these work arrays need to be defined as very long arrays to store the temporary variables. This would lead to unnecessary copies of huge arrays when they are passed as parameters between Fortran and Python.

For example, in the test case we applied in Figure 3.1, *rwork* (real work array) is initialized with length 522 but containing only 7 optional inputs,

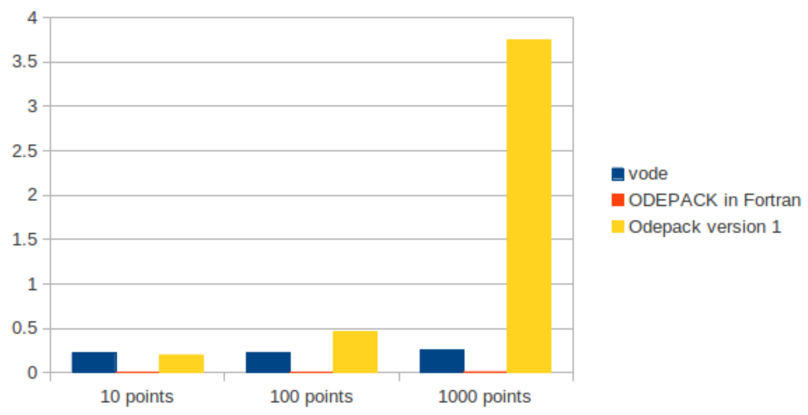


Figure 3.1: Efficiency lost in the first version

while *iwork* (integer work array) is initialized with length 45 and containing only 9 optional inputs. That is, in each time step, these huge arrays are copied repeatedly both for input and output.

Therefore, I decided to initialize these work arrays in Fortran and pass only the desired optional information between Python and Fortran part with much shorter arrays.

1. Implement a subroutine *advance_odepack()* in Fortran, which

    - take all the mandatory inputs for the desired ODEPACK solver, together with optional inputs from short arrays;
    - initialize work arrays with lengths required by ODEPACK solvers;
    - set optional inputs in work arrays;
    - call desired Fortran subroutine to step forward;
    - finally return the solution value on next time step, together with short arrays for optional outputs.

2. Generate a new extension module which only wrap the above subroutine.

    A string parameter '*subroutinename*' is defined to represent the desired solver. Then we can take all the possible inputs from Python code, and steer to the specified Fortran subroutine (like *dlsode, dlsoda*, etc..)

    The input parameter list are very long to contain all the possible callback functions for ODEPACK, such as *jac_column, jac_lsodi*,

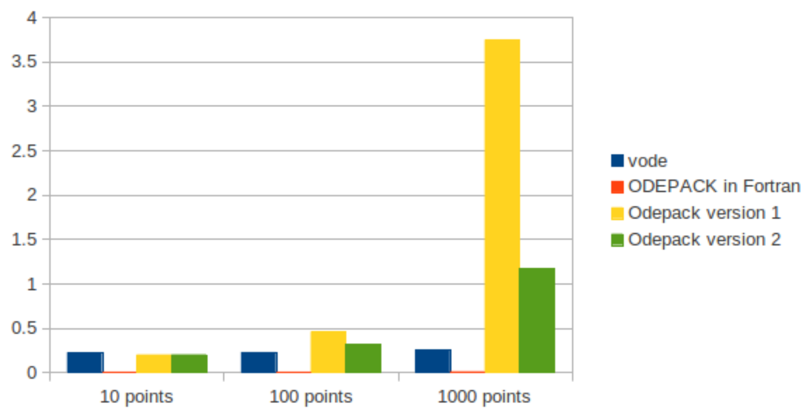


Figure 3.2: Efficiency improvement in the second version

etc.. Fortunately, we can reuse many signatures for these callback functions in the first version.

In this version, we do not need to call subroutines in ODEPACK from Python code. Hence in the signature file, only subroutine *advance_odepack* is involved.

3. Modification in Python code.

   In order to get desired solution through calling *advance_odepack*, we need to pass only short work arrays together with all possible parameters. Note that all the possible callback functions need to passed to Fortran even if they are not involved in current solver. So these functions need to be initialized as dummy functions with parameter lists suitable for signature of *advance_odepack*.

Figure 3.2 illustrates the improved efficiency of this version applying the same test case as in Figure 3.2.

The efficiency of this version is significantly improved comparing with the first version. But it is still much worse comparing with Vode and original ODEPACK. When I tried another test with 10000 time steps, it took 6 seconds to get the same result which was returned in 0.03 second from the original Fortran package.

3.2.3 Third version: migrate time loop to Fortran part, and start iteration in *solve()*

From Figure 3.1 and Figure 3.2, we can see that the running times depend much on the number of time steps, and seems growing exponentially. Naturally I got an idea: it might be much better if I migrate the time loop into Fortran part, and make a call directly from *solve()*.

1. Implement another wrap-subroutine *solve()* in Fortran, which loop on time points, call desired Fortran solvers for each step, and finally return the solution array on all time points.

   The input parameters are:

   - callback function *terminate* to control stop events.
   - a string for name of desired Fortran solver.
   - a complete time sequence.
   - an 2-dimension array which include all solution values, which is used to check the stop events
   - all possible parameters and short arrays for optional inputs, in the same manner as in the second version

2. Generate the third extension module which only wrap the above subroutine.

   Complete 2-dimension arrays for solution values are passed as inputs from Python to this new subroutine. But this point lead to only little loss in efficiency. In most cases, if iteration in ODEPACK is going smoothly, this subroutine will be called only once.

3. Modification in Python code.

   Based on second version, there are not much work for modification in Python. Except for parameters in the second version, we need to pass complete arrays for *time_points* and independent variable together with *terminate* function to control the stop events, and finally make a call to extension module in *solve()*.

Figure 3.3 illustrates the new efficiency performance of this version applying the same test case as in Figure 3.2. Note that the running time of first version with 1000 time steps are removed from the figure to get a better scale.

As same as original Fortran package, the running times of the third version seems to remain stable on different number of time steps. To get a better view, I applied *time* module to get the system time right before and after the calling to extension module in function *Odepack.solve()*.

```
def solve(self, time_points, terminate=None):
        import time
        ...
        time_start = time.time()
        # start iteration with a call to extension module
        nstop, urr, istate, rinfo, iinfo = \
              apply(_odepack.solve, \
```

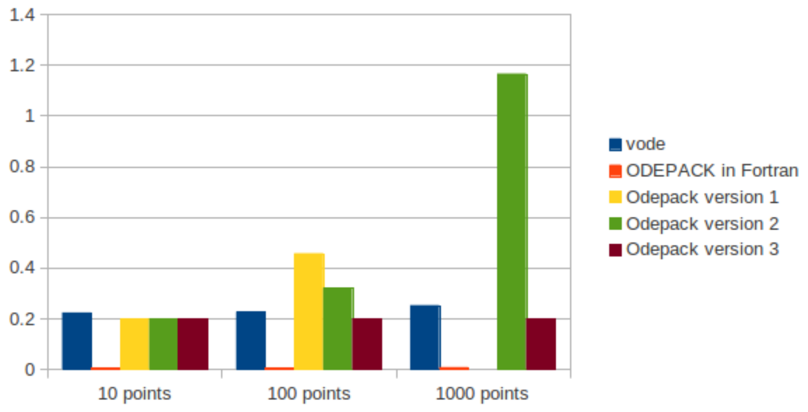


Figure 3.3: More improvement in the third version

```
                  (terminate_int, itermin, nstop, f, urr,
                   self.t, self.itol, self.rtol, self.atol,
                   istate, self.iopt, self.rwork_in, self.lrw,
                   self.iwork_in, self.liw, jac,
                   jac_column, self.jac_lsodi, self.jac_lsoibt,
                   self.jac_lsodis, self.adda_lsodi,
                   self.adda_lsoibt, self.adda_lsodis,
                   res, self.ydoti, self.mf, g, self.ng,
                   solver_name), \
                   self._extra_args_fortran)
time_stop = time.time()
print 'It takes %s for iteration in Fortran part.' \
       % (time_stop - time_start)
...
```

In this way, we can measure the total running time, the time for calculating in Fortran part, and finally get the time of our Python initialization and setting in Python interface.

Usually, users worry about possible efficiency loss because callbacks to Python are well-known to be expensive. Fortran needs to call back to Python for computing *f* or *jac*, at all time steps. But according to the time analysis in Table 3.2, we can see that in this test case, calculation in odesolvers seems to be as fast as in Fortran package.

It could be even better for more complicated ODE problems, in term of the efficiency difference between the original ODEPACK package and this wrapper in Python. As an experiment result a few years ago, scientists have estimated that the overhead of callback is equivalent to about 70 arithmetic operations. Therefore, if *f* is a long function with many arithmetic operations (e.g., calling intrinsic functions such as the sine, hyperbolic sine, exponential, or logarithm), the overhead of the wrapping code may be relatively ignorable comparing with the work inside *f*.

Number of time steps	ODEPACK in Fortran	Odepack in odesolvers		
	Total time	Setting in Python	Calculation in Fortran	Total
10	0.003	0.171	3e-4	0.174
100	0.003	0.181	3e-4	0.184
1000	0.003	0.165	3e-4	0.168
10000	0.003	0.209	3e-4	0.212
100000	0.003	0.461	3e-4	0.464

Table 3.2: Time analysis for the 3rd version

3.2.4 Better efficiency: supplying user-defined functions as Fortran subroutines

For users who care most about efficiency, there is another choice to make it even better: user-defined functions can be supplied in Fortran form to avoid the possible efficiency loss of callbacks.

Functions can also be supplied as a multi-line string which contains the Fortran subroutines. Then this string will be complied with F2PY in the start of initialization step. It is also possible if the users prefer to compile functions by themselves and supply the F2PY compiled-function as input.

With this choice, even very complicated ODE problems can be solved efficiently as in the original Fortran package. However, comparing with supplying functions directly in Python, the Fortran subroutine need to be defined in very strict form and parameter list.

For example, function *f* are required to be defined in the following form:

```
subroutine f(neq,t,u,du)
  CF2PY intent(hide)      neq
  CF2PY intent(out)       du
        integer           neq
        double precision  t,u(neq),du(neq)
        ...
        return
        end
```

That is, users need to have enough experience with programming in Fortran. Then some readers may ask: why bother these users to apply wrappers in *odesolvers*, but not directly applying Fortran package instead?

Answer for this question is the main topic of Section 3.3. This wrapper not only keeps the efficiency and reliability of the original package, but also supplies the great features of *odesolvers*.

3.3 Why Odepack?

Referring to the efficiency analysis in Chapter 5, we can find that the efficiency loss of *Odepack* is not as significant as imagination.

Type	subroutine dlsode in in Fortran	odesolvers.Lsode in Python
Mandatory	f,neq,y,t,tout,itol,rtol, itask,istate,iopt,rwork, liw,jac,mf,atol, lrw,iwork	f
Optional	f_extra_args, hmax,hmin,maxord ml,mu jac_extra_args,h0, mxstep,mxhnil	jac,rtol,atol,ode_method, min_step,max_step, nsteps,ml,mu,order, iter_method,jac_banded first_step,max_hnil,f_args,

Table 3.3: Comparison of parameter lists

For users who need ultimate performance, ODEPACK with a driver in Fortran 77 can be regarded as superior to this Python wrapper. But for those who want to rapidly prototype an ODE system and do modelling in a more convenient way, *Odepack* will be a good alternative for them.

Comparing with usage of original Fortran package, users will enjoy several significant advantages of using our Python version as outlined in the following.

3.3.1 Fewer parameters, fewer errors

Long parameter lists of ODEPACK are shortened and simplified significantly. In function *set_internal_parameters()*, numerous parameters will get their proper values if they are not explicitly specified from input. A comparison for parameter lists is listed in Table 3.3.

That is, in the simplest form, we can even get solution with only one parameter *f*. This feature will definitely reduce the possibilities for input errors. Generally, the fewer parameters to be managed by users, the fewer errors will occur.

3.3.2 Clearer syntax

Another advantage is obtained from the clear and compact syntax in *odesolvers*. Specially for these solvers which hold many parameters, users can make use of function *get_parameter_info()* to get all the useful information, e.g. type, default value, explanation message, etc..

Comparing with the intelligible parameter list in Fortran package, users get a much more friendly and flexible interface for these complicated solvers.

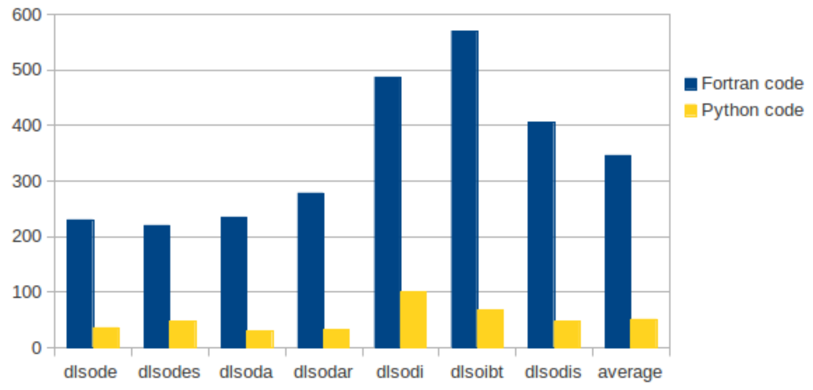


Figure 3.4: Comparison of code length

3.3.3 Trivially switch to other solvers in *Odepack*

Although ODEPACK is considering as the most important package for solving ODE problems, sometimes user can not get satisfactory solution in acceptable running time for some ODE problems.

Taking advantage of the unified feature of *odesolvers*, users can easily switch from ODEPACK to a lot of other algorithms offered in *odesolvers*. Even if the user-defined functions are supplied as Fortran subroutines or multi-line strings with the parameter lists starting with *'t,u,...'*, these functions will be wrapped automatically to the general form *'u,t,...'* in *Odepack*, and ready to switch to other solvers smoothly.

3.3.4 Shorter code

In the netlib repository [23], 9 demonstration programs can be found in a file named *opkddemos.f*. I tried to implement all these demo problems in our Python interface.

Based on my Python code and *opkddemo.f* from ODEPACK, I make a chart table to compare the number of necessary code lines to implement the same demo problems through ODEPACK and my implementation.

As illustrated in Figure 3.4, the average code length has been decreased from 346 lines to 50 lines, which has been reduced with a percentage 86%. Definitely we benefit much from the precise syntax of Python, but I think it owes much more to our simplified interface and shortened parameter list in *Odepack*.

3.3.5 Better error-check, clearer messages

Validity check is a very important and feasible way to provide a more friendly user interface. Inappropriate values of input parameters should be detected and informed before the starting of iteration. This help users to find out what might be wrong and avoid coming errors in advance.

As discussed in the previous section, due to the long parameter lists and Fortran programming style, interrupt errors often occur with(or without) confusing messages when some input parameters is set with inappropriate values.

For example, suppose that a user forgot to input function *jac* when he or she set parameter *iter_method* explicitly as 1, which implies a user-defined full Jacobian matrix.

In ODEPACK, an annoying message will come out – 'Segmentation fault.'. While in our interface, the error message will be:

```
ValueError:       Error! Insufficient input!
      jac must be set when iter_method is 1!
```

This is certainly a more friendly message for the users to know what is wrong and fix the problem easily.

3.3.6 Fewer interrupts with automatic error-handling

Due to the complicated structure and algorithms in ODEPACK, there are many different kinds of possible interrupt errors. Among them, some common errors can be repaired automatically in my implementation.

For example, when the status flag *istate* is returned with a value -2, it indicates that there are too much accuracy requirement. Together with this returned istate, a suggested tolerance factor is returned in optional outputs.

Then it is unnecessary to make a special interrupt for this error. We can simply multiply tolerance attributes (*rtol* and *atol*) with the suggested factor, and restart the iteration automatically.

6 negative values of return status istate represent 6 kinds of typical errors respectively.

In method *advance()*, I tried to fix repairable errors to continue (or restart) iteration automatically. All these automatic changes are informed to users. Users can avoid many annoying interrupts, and get corresponding information explicitly.

Status flag	Error	ODEPACK in Fortran	Odepack in Python
-1	Excessive amount of work. step number > max_step	Interrupt	Handled without interrupt. max_step is increased automatically.
-2	Too much accuracy was required.	Interrupt	Handled without interrupt. Tolerance parameters are multiplied with suggested factor.
-3	Illegal input was detected, before taking any integration steps.	Interrupt	Handled without interrupt for most cases..
-4	Repeated error-test failures.	Interrupt	Interrupt with message printing.
-5	Repeated convergence-test failures.	Interrupt	Interrupt with message printing.
-6	Error become zero with pure absolute error control.	Interrupt	Handled without interrupt. Increase *atol* with a default value.

Table 3.4: Common errors and handelling

Chapter 4

A collection of Runge-Kutta methods

4.1 Motivation

Dormand&Prince is a widely-used Runge-Kutta explicit method to solve ODE problems. When I attempted to implement this popular method, I found it has many features in common with another popular method that I implemented before – Fehlberg.

According to the definition for Runge-Kutta methods in Wikipedia dictionary [18], all the Runge-Kutta explicit methods has a uniformed table (namely "Butcher Tableau") to store their coefficients. As long as we have a complete Butcher Tableau at hand, we can specify numerical formula for the corresponding explicit Runge-Kutta method with coefficients in the table. Hence it might be feasible to set up a general superclass for all the explicit Runge-Kutta methods. This is exactly why I was motivated to develop a collection for Runge-Kutta methods.

After implementing a superclass *RungeKutta* and several Runge-Kutta methods, I found that it is difficult to include all the Runge-Kutta methods into my Python code, merely because I can not find many Butcher tableau from the web. Only those Butcher tableau for common Runge-Kutta methods are available in the respective websites. For example, I know there are several Runge-Kutta methods that are very complicated with the 8/9-th order, but the detailed data of their Butcher tableau could not be found directly from the web.

Then I was inspired with another idea: Why cannot we leave the freedom to users for supplying their own Runge-Kutta methods? If a user has a complete Butcher tableau for a new or uncommon Runge-Kutta method in hand, the user can supply this tableau from input, and apply the self-defined solver as an ordinary Runge-Kutta method. On this basis, I defined a separate class *MyRungeKutta* for this purpose.

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
...					
c_s	$b_{s,1}$	$b_{s,2}$		$b_{s,s-1}$	
	b_1	b_2	...	b_{s-1}	b_s

Table 4.1: Butcher tableau for unadaptive RK methods

4.2 Mathematics about explicit Runge-Kutta methods

The family of explicit Runge-Kutta methods is given by:

$$u_{n+1} = u_n + \sum_{i=1}^{s} b_i k_i$$

where

$$\begin{aligned}
k_1 &= hf(u_n, t_n) && (4.1)\\
k_2 &= hf(u_n + a_{21}k_1, t_n + c_2 h) && (4.2)\\
k_3 &= hf(u_n + a_{31}k_1 + a_{32}k_2, t_n + c_3 h) && (4.3)\\
\ldots &= && (4.4)\\
k_s &= hf(u_n + a_{s1}k_1 + a_{s2}k_2 + \ldots + a_{s,s-1}k_{s-1}, t_n + c_s h) && (4.5)\\
& && (4.6)
\end{aligned}$$

To specify a particular method, one needs to provide the integer s (the number of stages), and the coefficients a_{ij} (for 1 <= j < i <= s), b_i (for i = 1, 2, ..., s) and c_i (for i = 2, 3, ..., s). These data are usually arranged in a mnemonic device, known as a "Butcher tableau" (named after John C. Butcher):

The adaptive methods are designed to produce an estimate of the local truncation error of a single Runge-Kutta step. This is done by having two methods in the tableau, one with order *p* and the other one with order *p-1*.

The lower-order step is given by

$$u^*_{n+1} = u_n + \sum_{i=1}^{s} b^*_i k_i$$

where the k_i are the same as for the higher order method. Then the local error is

$$e_{n+1} = u_{n+1} - u^*_{n+1} = h\sum_{i=1}^{s}(b_i - b^*_i)k_i$$

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
...					
c_s	$b_{s,1}$	$b_{s,2}$		$b_{s,s-1}$	
	b_1	b_2	...	b_{s-1}	b_s
	b_1^*	b_2^*	...	b_{s-1}^*	b_s^*

Table 4.2: Butcher tableau for adaptive RK methods

which is $O(h^p)$. This estimated error is used to adjust step size whether it exceeds the desired error tolerance.

For adaptive Runge-Kutta methods, Butcher tableau is extended with an extra row to give the values of b_i*:

For the adaptive Runge-Kutta methods, the local error between two levels will be used for the purpose of error-control. The main idea is to calculate local error *e* with current step size *h*. If *e* exceeds tolerance error *tol*, a factor *s* need to be determined to approximately adjust *h* to be a new value h_{new}, which makes the new local error smaller than *tol*.

According to the suggestion in «Numerical Methods for Engineers» [25], the following formula for factor s is applied in this collection:

$$s = 0.8(tol/e)^{1/p}$$

4.3 Designing issues

This collection of Runge-Kutta methods are completely implemented in Python language. Similar to *Odepack*, I tried to implement all the common features of Runge-Kutta methods into an 'abstract' superclass *RungeKutta*, and then include all the methods as subclasses underneath. Butcher tableau are stored in subclasses, and common methods in superclass are applied to get desired solutions.

4.3.1 Superclass *RungeKutta*(Adaptive)

RungeKutta is defined as a subclass of *Adaptive*, and hence inherit tolerance parameters (*atol*l and *rtol*) from *Adaptive*. Besides, three input parameters (*min_step*, *max_step* and *first_step*) are introduced for users to control the step size.

There are three components to specify the numerical scheme:

- *_butcher_tableau* A 2d-array to store the coefficients in standard form.
- *_method_order* An integer number for 1-level Runge-Kutta methods, or a pair of integers for adaptive ones.

- *advance()* One step forward with the general formula of explicit Runge-Kutta methods, with specified coefficients extracted from *_butcher_tableau* of the current solver. For adaptive methods, the formula for adjusting factor described in Section 4.2 is applied to adjust step size.

4.3.2 Hard-coded methods: Subclasses of Runge-Kutta

A new Runge-Kutta method can be integrated into this collection trivially. Setting values for two class attributes *_butcher_tableau* and *_method_order*, that is all we need to do.

For example, the standard Runge-Kutta4 method is implemented in 8 lines:

```
class RungeKutta4_RK(RungeKutta):
    _butcher_tableau = numpy.array(\
        [[0., 0., 0., 0., 0.],
         [.5, .5, 0., 0., 0.],
         [.5, 0., .5, 0., 0.],
         [1., 0., 0., 1., 0.],
         [0., 1./6., 1./3., 1./3., 1./6.]])
_method_order = 4
```

Another example is the implementation of the popular Fehlberg45 method.

```
class FehlBerg_RK(RungeKutta):
    _butcher_tableau = numpy.array(\
    [[0., 0., 0., 0., 0., 0., 0.],
     [1./4. 1./4., 0., 0., 0., 0., 0.],
     [3./8., 3./32., 9./32., 0., 0., 0., 0.],
     [12./13., 1932./2197., -7200./2197., 7296./2197., 0., 0., 0.],
     [1., 439./216.,-8., 3680./513.,-845./4104., 0., 0.],
     [1./2., -8./27., 2., -3544./2565., 1859./4104., -11./40., 0.],
     [0., 25./216., 0., 1408./2565., 2197./4104., -1./5., 0.],
     [0., 16/135., 0., 6656./12825.,28561./56430.,-9./50.,2./55.]])
    _method_order = (4,5)
```

Any explicit Runge-Kutta method can be included in the same way as the above two examples. In the current version only 8 Runge-Kutta methods are included in this collection, simply because I do not have any more Butcher tableau to extend it.

The implemented methods are: *RungeKutta2_RK, RungeKutta3_RK, ForwardEuler_RK, DormandPrince_RK, RungeKutta4_RK, FehlBerg_RK, CashKarp_RK, BogackiShampine_RK.*

As long as we get complete data for any unimplemented Runge-Kutta method, we can integrate it easily with short code. This makes this collection very extensible for future development.

4.4 User-defined Runge-Kutta methods: class *MyRungeKutta*

In *MyRungeKutta*, two input parameters (namely *Butcher_tableau* and *method_order()*) are defined to specify the coefficients and method order in standard form.

4.4.1 *get_order()*: estimate method order with the given Butcher tableau

Firstly, we think about how to estimate orders for unadaptive Runge-Kutta methods.

The order p is estimated by computing errors for an ODE problem with the specified coefficients in user-supplied Butcher tableau. This ODE problem should have a known analytical solution to calculate the exact errors.

Two step-size dt_1 and dt_2 are applied to solve the sample ODE problem. Through a comparison with the exact solution, we calculate their corresponding errors err_1 and err_2, roughly $O(dt_1^p)$ and $O(dt_2^p)$.

That is

$$\frac{err_1}{err_2} = \frac{O(dt_1^p}{O(dt_2^p}$$

Take logarithms of both sides in above equation,

$$\frac{log(err_1)}{log(err_2)} = \frac{p * log(dt_1)}{p * log(dt_2)}$$

Now order p is ready to be estimated:

$$p = \frac{log(err_1)/log(err_2)}{log(dt_1)/log(dt_2)}$$

As each 2-level adaptive method consists of two 1-level Runge-Kutta methods with different orders, we can extract the coefficients for these two 1-level methods from the Butcher Tableau separately.

The Butcher Tableau of adaptive methods is always a *n+1*n* array. The first *n-1* rows are coefficients for internal steps that are common to both high-order and lower-order levels. Thus, we can form two separate *n*n* arrays for these two levels, and evaluate their orders separately as ordinary 1-level methods.

For example, Table 4.3 contains the Butcher tableau of the popular Dormand&Prince Runge-Kutta method.

Suppose a user try to use *get_order()* to estimate the method order of above table. The above 9*8 array would be dissected to be two 8*8 square matrices:

Let us look at the following session to demonstrate the usage of this function:

```
>>> method = MyRungeKutta(f, \
      butch_tableau=DormandPrince_RK._butcher_tableau)
>>> method.get_order()
      The order of user-defined method is not provided.
      The value of calculated order is (5, 4)
```

0					
1/5	1/5				
3/10	3/40	9/40			
4/5	44/45	-56/15	32/9		
8/9	19372/6561	-25360/2187	64448/6561	-212/729	
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656
1	35/384	0	500/1113	125/192	-2187/6784
1	35/384	0	500/1113	125/192	-2187/6784
5179/57600	0	7571/16695	393/640	-92097/339200	187/2100

Table 4.3: Butcher tableau for Dormand&Prince method

0							
1/5	1/5						
3/10	3/40	9/40					
4/5	44/45	-56/15	32/9				
8/9	19372/6561	-25360/2187	64448/6561	-212/729			
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656		
1	35/384	0	500/1113	125/192	-2187/6784	11/84	
1	35/384	0	500/1113	125/192	-2187/6784	11/84	0

Table 4.4: Butcher tableau for Level 1 in Dormand&Prince method

0					
1/5	1/5				
3/10	3/40	9/40			
4/5	44/45	-56/15	32/9		
8/9	19372/6561	-25360/2187	64448/6561	-212/729	
1	9017/3168	-355/33	46732/5247	49/176	-5103/18656
1	35/384	0	500/1113	125/192	-2187/6784
5179/57600	0	7571/16695	393/640	-92097/339200	187/2100

Table 4.5: Butcher tableau for level 2 in Dormand&Prince method

Chapter 5

Efficiency issues

odesolvers is a general collection of ODE solvers, and it consists of implementations in different kinds, like wrapper to other Python packages, wrapper to traditional ODE packages in other programming languages, scripts directly in Python. Accordingly, it is hard to draw one conclusion to evaluate the efficiency of the whole package.

In this chapter, we propose to select some representatives from the solvers in the current package, then to evaluate their efficiency separately.

5.1 Odepack

5.1.1 Example 1: Supply *f* as Python function to ODEPACK wrappers

There are two well-known ODEPACK wrappers in Python: *ode.lsode* in the ODE interface of Lawrence Livermore National Labs, and the other is *scipy.integrate.odeint* for *dlsoda*.

In this section, I will make an efficiency comparison with these two ODEPACK wrappers, by using the same test case in Section 3.2.

The test problem is $u' = A * u$, where number of equation is 25, and A is a banded lower triangular matrix derived from 2-dimension advection PDE.

In Figure 5.1, we find that the running times of *dlsode* in Lawrence interface depends much on the number of time steps, while *dlsoda* in *scipy* and our wrapper *Odepack* remain stable with a comparable efficiency level.

5.1.2 Example 2: Supply *f* as Fortran subroutine to ODEPACK wrappers

In the interface of Lawrence lab and our wrapper *Odepack*, users can supply *f* as a Fortran subroutine to avoid the efficiency loss of callbacks. Therefore, I tried another test with supplying *f* as a Fortran subroutine to *odesolvers*, the Lawrence interface and the original ODEPACK package.

We can go further with the analysis of the running times in *odesolvers*. In the same way we applied in Section 3.2.3, we can measure the time for calculating in Fortran part.

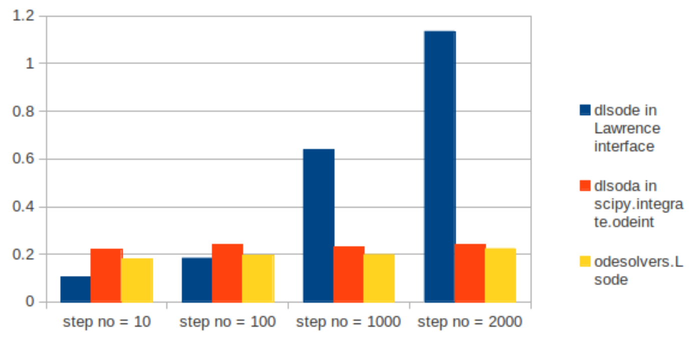


Figure 5.1: Supply f as Python function to ODEPACK wrappers

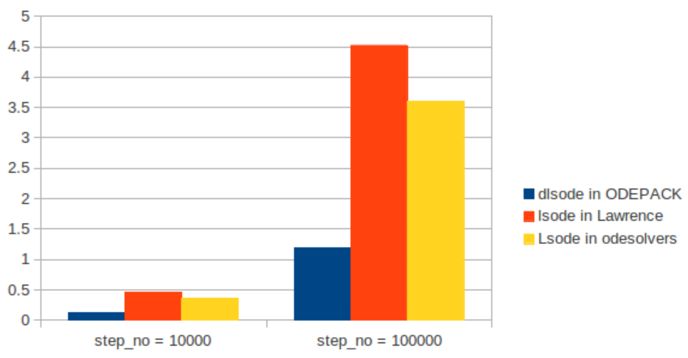


Figure 5.2: Supply f as Fortran subroutines to ODEPACK wrappers

In this test case, the time for calculating job in Fortran part remain stable around 3.35e-4 seconds, whatever the step number is set, e.g. 10 or 1000000. It means that in this test case, the slightly increasing time with more time steps is mainly used in the initialization and setting job in Python.

5.1.3 Conclusion

Based on the two examples above, we can draw the following conclusion:

- Limited efficiency loss comparing with ODEPACK in Fortran

  Odepack is always slower than ODEPACK with a driver in Fortran, which is regarded as one of the best computational languages. But since we have migrate the CPU intensive part to Fortran, the calculation part can be taken almost as fast as in Fortran code, especially when users supply functions as Fortran subroutines.

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

  All of these wrappers provide a clean syntax and friendly interface to users. But in term of efficiency, *Odepack* appears best in both examples.

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

5.2 Efficiency of Runge-Kutta methods in different implementation

In this section, we selected several Runge-Kutta solvers from *odesolvers* and try to compare their efficiency:

1. class *RungeKuttaFehlberg* Python implementation based on its own scheme.

2. class *RKF45* Wrapper for *rkf45.f*.

3. class *Fehlberg_RK* Python implementation as a member of collection *RungeKutta* in the general scheme.

4. class *Dopri5* Wrapper for *dopri5.f*.

atol	rtol	RungeKutta Fehlberg	RKF45	Fehlberg _RK	Dopri5
1e-6	1e-6	0.27	0.11	0.48	0.15
1e-10	1e-10	0.41	0.23	0.51	0.21
1e-16	1e-16	3.3	0.97	20	1.7

Table 5.1: Efficiency of Runge-Kutta solvers

Applying these solvers to solve the same ODE problem as in Section 5.2 , we got the following result:

All these solvers are applying Runge-Kutta methods with order (4,5). In this test case, they returned the same value for each accuracy level, with different efficiency level.

The four solvers in Table Referencestab:rk effi, can be categorized into three groups in terms of efficiency level:

- *Dopri5* and *RKF45*: these two solvers are wrappers to Fortran solvers, and show the best efficiency.

- *RungeKuttaFehlberg* and *Fehlberg_RK*: these two solvers are programmed in Python by me, but show a big difference in the level of efficiency. This is because of more loops in the collection *RungeKutta*, in order to fit the general schemes.

Due to the difference rooted from their implementations, solvers appear to have different efficiency. Nevertheless, for simple ODE problems, this level of efficiency is sufficient to obtain a satisfactory result in today's computer. For those users who care much about the efficiency, they have adequate choices to achieve a higher level of efficiency, for example supplying Fortran subroutines to *RKC* or *RKF45* in *odesolvers*.

Chapter 6

Concluding Remarks

Solving ODE is an old topic, and a lot of numerical methods and software are available for it. The users build up their mathematical models around ODE, and they only want to get a simple and easy solution of their own problems.

However, the users of ODE software are all different. They may be novice users who are just new to ODE, or scientists who may know much more about ODE and modelling but less into programming, and possibly professional users who are familiar with both mathematical modelling and programming. Thus their understanding of ODE software and their expectation on software usages will be very different also.

A new Python interface is introduced in this thesis, with the aim to meet the needs from all users in solving ODE problems. Before this interface is developed, there was no existing software that can collect so many ODE solvers into one package with a unified user interface. And this work has been proven to be meaningful to some users. Although this interface has not been published formally, it has already attracted some interests among specialists in the ODE field, for instance Olivier Verdier, the creator of *odelab*.

This interface is based on a class hierarchy with only basic structure (namely *ODESolver*), which is first introduced by my supervisor, Professor Hans Petter Lantangen in his book «A Primer on Scientific Programming with Python» [24]. Until now, it has been extended with many facilities and 43 solvers.

My contribution mainly lies in the following:

- Wrap solvers in a widely used Fortran package ODEPACK.

  This is the most time-consuming part of this project. I figured out all the implementation details of the ODEPACK wrapping on my own effort. In order to achieve the best performance and efficiency, these wrappers has been modified for many times. For example, three different extension module have been defined with different emphases. This lead to a lot of relevant modification job in programming and testing, e.g. wrapper code in Python and Fortran, F2PY signature files and test cases as well. Eventually, the final version is achieved with a significantly improved efficiency.

- Parameters handelling

  From my point of view, this is my most important contribution for this project. A global dictionary for parameter properties is defined together with a set of relevant methods common in the whole class hierarchy. As described in Section 2.3, this mechanism ensures the uniformed features both for users and developers, and make this interface with great extensibility and potential.

- Implementation of Runge-Kutta collection. This collection hide all the details of explicit Runge-Kutta methods behind, and provide an extremely simple interface for future development. All the explicit Runge-Kutta methods can be integrated with trivial work. With the help of subclasses *MyRungeKutta*, users can easily make experiments both for developing and for testing with a single Runge-Kutta Butcher tableau. As far as we know, there is probably no other general user-friendly implementation where the user can easily make such kind of experiments.

- Implementation of many numerical methods and wrapper for ODE software. Among the current 43 solvers listed in Appendix A, I implemented 33 solvers independently, e.g. three-step AdamsBashMoulton method, MidpointImplicit method, wrappers for *rkc.f*, *rkf45.f, PyDSTool*, etc..

This thesis aims not only to introduce this unified Python interface for ODE software, but also tries to demonstrate the process how to integrate various types of software and create a unified interface which can fulfill the diverse requirements of different users. This kind of interface shall provide a set of common features applicable for all integrated components.

With this aim in mind, this thesis emphasizes on the structure of the solver class hierarchy and the implementation procedure of creating the interface, but does not describe the details of these integrated components one by one; thus only a few representatives of these components are introduced in detail in this thesis.

Due to the time limit, some work are left to the future:

- to integrate a set of standard benchmark problems as test samples;
- to improve the efficiency of *RungeKutta* collection;
- to integrate some new ODE packages like odelab and PySUNDIALS.

Bibliography

[1] Gopal K. Gipta, Ron Sacks-Davis and Peter E. Tischer A, *Review of Recent Developments in Solving ODES*

[2] Alan C. Hindmarsh. *Brief Description of ODEPACK - A Systematized Collection of ODE Solvers Double Precision Version*, available at http://www.netlib.org/odepack/opkd-sum

[3] Hans Petter Langtangen, *Python Scripting for Computational Science*, Second edition, 2005.

[4] SciPy homepage. www.scipy.org

[5] Pearu Peterson. *F2PY: Fortran to Python interface generator,* available at http://cens.ioc.ee/projects/F2PY2e/

[6] Embedded Runge-Kutta Methods in Mathematics Source Library C & ASM, available at http://mymathlib.webtrellis.net/diffeq/embedded_runge_kutta/

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing*, Second Edition

[8] Chappra & Cannle, *Numerical Methods for Engineers,* , Second Edition, McGraw-Hill, 1985.

[9] Eric Jones, Travis Oliphant, Pearu Peterson. *Scipy: Open Source scientific tools for Python*, 2001

[10] Matlab software package, available at http://www.mathworks.com

[11] The Python Wiki, available at http://wiki.python.org/moin/FrontPage

[12] M.B. Allen III, I.Herrera, and G.F. Pinder. *Numerical Modelling in Science and Engineering.*, Wiley, 1998

[13] PyDSTool software package, available at http://www2.gsu.edu/~matrhc/PyDSTool.htm

[14] Numerical Python software package, available at http://sourceforge.net/projects/numpy

[15] G. van Rossum and F. L. Drake, *Numpy Disutils - Users Guide*, available at http://scipy.org/Documentation/numpy_disutils

[16] G. can Rossom and F.L. Drake. *Python Library Reference*, available at http://docs.python.org/release/2.5.2/lib/lib.html

[17] Sympy homepage, available at code.google.com/p/sympy

[18] WikiPedia page for Runge-Kutta methods, available at http://en.wikipedia.org/wiki/Runge-Kutta_methods

[19] WikiPedia page for Ordinary differential equations, available at http://en.wikipedia.org/wiki/Ordinary_differential_equation

[20] Ya Yan Lu. *Numerical Methods for Differential Equations*

[21] Dana Petcu. *Software issues in solving initial value problems for ordinary differential equations*

[22] Test set for IVP solvers, available at http://www.dm.uniba.it/~testset/testsetivpsolvers

[23] Netlib Repository, available at http://www.netlib.org

[24] Hans Petter Langtangen. *A Primer on Scientific Programming with Python*, First Edition, 2009

[25] Chappra & Cannle. *Numerical Methods for Engineers*, Second Edition, McGraw-Hill, 1985

[26] Clive G. Page. *Professional Programmer's Guide to Fortran77*, available at http://www.star.le.ac.uk/~cgp/prof77.html#tth_sEc1.3

[27] B.P. Sommeijer, L.F. Shampine, J.G. Viewer. *RKC: An explicit solver for parabolic PDEs.*

[28] The Sage Group. *Numerical Computing with Sage*, Release 4.0.

Appendix A

List of available solvers

A.1 Numerical methods implemented in Python

MySolver
: Users can define their own solver through supplying a function myadvance() to step forward.

ForwardEuler
: u(n+1) = u(n) + dt*f(u(n), t(n))

MidpointIter
: A midpoint/central difference method with one or two fixed-point iterations to solve the nonlinear system. Either Forward Euler scheme or Heun scheme would be recovered corresponding to specified number of iteration steps (1 or 2).

Heun
: u(n+1) = u(n) + 0.5*dt*(f(u(n),t(n)) + f(u(n)+dt*f(u(n),t(n)),t(n+1))) Forward Euler scheme as start value.

Leapfrog
: u(n+1) = u(n-1) + dt2*f(u(n), t(n)), where dt2 = t(n+1) - t(n-1). Forward Euler is used for the first step.

LeapfrogFiltered
: Since Leapfrog gives oscillatory solutions, this solver applies a common filtering technique: u(n) = u(n) + gamma*(u(n-1) - 2*u(n) + u(n+1)) with gamma=0.6 as in the NCAR Climate Model.

AdamsBashforth2
: Second-order Adams-Bashforth method. u(n+1) = u(n) + dt/2.*(3*f(u(n), t(n)) - f(u(n-1), t(n-1)))

AdamsBashforth3
: Third-order Adams-Bashforth method. u(n+1) = u(n) + dt/12.*(23*f(u(n), t(n)) - 16*f(u(n-1), t(n-1)) + 5*f(u(n-2), t(n-2)))

AdamsBashforth4
Fourth-order Adams-Bashforth method. u(n+1) = u(n) + dt/24.*(55.*f(u(n), t(n)) - 59*f(u(n-1),t(n-1)) + 37*f(u(n-2), t(n-2)) - 9*f(u(n-3), t(n-3)))

AdamsBashMoulton2
Two step Adams-Bash-Moulton method. Third accuracy order. predictor = u(n) + dt/12.*(23.*f(u(n), t(n)) - 16*f(u(n-1),t(n-1)) + 5*f(u(n-2), t(n-2))) corrector = u(n) + dt/12.*(8.*f(u(n), t(n)) - f(u(n-1), t(n-1)) + 5*f(predictor, t(n+1)))

AdamsBashMoulton3
Three step Adams-Bash-Moulton method. Fourth accuracy order. predictor = u(n) + dt/24.*(55.*f(u(n), t(n)) - 59*f(u(n-1),t(n-1)) + 37*f(u(n-2), t(n-2)) - 9*f(u(n-3), t(n-3))) corrector = u(n) + dt/24.*(19.*f(u(n), t(n)) - 5*f(u(n-1), t(n-1)) + f(u(n-2), t(n-2)) + 9*f(predictor, t(n+1)))

RungeKutta2
u(n+1) = u(n) + dt*f(u(n) + 0.5*(dt*f(u(n),t(n))),t(n) + 0.5*dt)

RungeKutta4
u(n+1) = u(n) + (1/6.0)*(K1 + 2*K2 + 2*K3 + K4) where K1 = dt*f(u(n), t(n)) K2 = dt*f(u(n) + 0.5*K1, t(n) + 0.5*dt) K3 = dt*f(u(n) + 0.5*K2, t(n) + 0.5*dt) K4 = dt*f(u(n) + K3, t(n) + dt)

RungeKuttaFehlberg
Runge-Kutta-Fehlberg method of order 4-5. Details for scheme can be found in wikipedia page for Runge-Kutta methods [18].

RungeKutta3
u(n+1) = u(n) + (1/6.0)*(K1 + 4*K2 + K3) where K1 = dt*f(u(n), t(n)) K2 = dt*f(u(n) + 0.5*K1, t(n) + 0.5*dt) K3 = dt*f(u(n) - K1 + 2*K2, t(n) + dt)

BackwardEuler
Implicit Backward Euler method, applying Newton or Picard method as nonlinear solver. u(n+1) = u(n) + dt*f(u(n+1), t(n+1))

ThetaRule
A typical weighted method with factor theta. u(n+1) = u(n) + dt*(theta*f(u(n+1),t(n+1)) + (1 - theta)*f(u(n),t(n))) where theta is a float in [0,1].

MidpointImplicit
Implicit Midpoint method, applying Newton or Picard method as nonlinear solver. u(n+1) = u(n) + dt*f((u(n+1) + u(n))/2, t(n) + dt/2)

Backward2step
Implicit Backward Euler method with 2 steps. u(n+1) = u(n)*4/3 - u(n-1)/3 + (t(n+1)-t(n-1))*f(t(n+1), u(n+1))/3

AdaptiveResidual
Accept a specified solver, and integrates with this solver in an adaptive way with calculated residual as the error-check criteria.

A.2 Methods collected with superclass *RungeKutta*

RungeKutta2_RK
Same as solver *RungeKutta2*, except that it is implemented in form of a 1-level Runge-Kutta method.

RungeKutta3_RK
Same as solver *RungeKutta3*, except that it is implemented in form of a 1-level Runge-Kutta method.

RungeKutta4_RK
Same as solver *RungeKutta4*, except that it is implemented in form of a 1-level Runge-Kutta method.

ForwardEuler_RK
Same as solver *ForwardEuler*, except that it is implemented in form of a 1-level Runge-Kutta method.

DormandPrince_RK
Order (5,4) Runge-Kutta method with Dormand&Prince pairs applying. This is the default method applying in well-known solver *ode45* in Matlab. Data for Butcher tableau in this scheme can be found in a webpage for Runge-Kutta methods [6].

FehlBerg_RK
Order (4,5) Runge-Kutta method with FehlBerg pairs applying. This is the second method applying in well-known solver *ode45* in Matlab. Data for Butcher tableau in this scheme can be found in a webpage for Runge-Kutta methods [6].

CashKarp_RK
Order (5,4) Runge-Kutta method with CashKarp pairs applying. Data for Butcher tableau in this scheme can be found in a webpage for Runge-Kutta methods [6].

BogackiShampine_RK
Order (3,2) Runge-Kutta method with BogackiShampine pairs applying. Data for Butcher tableau in this scheme can be found in a webpage for Runge-Kutta methods [6].

MyRungeKutta
User-supplied RungeKutta method with providing Butcher-table in an 2d-array. The order should be provided if it is known. If not, the order would be estimated internally.

A.3 Wrappers for ODEPACK

Lsode

Wrapper for basic solver *dlsode* in ODEPACK package. In the stiff case, it treats the Jacobian matrix df/du as either a dense (full) or a banded matrix, and as either user-supplied or internally approximated by difference quotients. It uses Adams methods (predictor-corrector) in the non-stiff case, and Backward Differentiation Formula (BDF) methods (the Gear methods) in the stiff case. The linear systems that arise are solved by direct methods (LU factor/solve).

Lsodes

Wrapper for *dlsodes* in ODEPACK, a variant of basic solver *dlsode*. In the stiff case it treats Jacobian matrix in general sparse form. It can determine the sparsity structure on its own, or optionally accepts this information from the user. It then uses parts of the Yale Sparse Matrix Package (YSMP) to solve the linear systems that arise, by a sparse (direct) LU factorization/backsolve method.

Lsoda

Wrapper for *dlsoda* in ODEPACK, a variant of basic solver *dlsode*. It automatically selects between nonstiff (Adams) and stiff (BDF) methods. It uses the nonstiff method initially, and dynamically monitors data in order to decide which method to use.

Lsodar

Wrapper for *dlsodar* in ODEPACK, a variant of solver *dlsoda* with a rootfinding capability added.

Lsodi

Wrapper for *dlsodi* in ODEPACK, the basic solver for linearly implicit ODE systems in which all the matrices involved are assumed to be either dense or banded.

Lsoibt

Wrapper for *dlsoibt* in ODEPACK, a variant for the basic solver *dlsodi* to solve linearly implicit ODE systems. All the matrices involved are assumed to be block-tridiagonal. Linear systems are solved by the LU method.

Lsodis

Wrapper for *dlsodis* in ODEPACK, a variant for the basic solver *dlsodi* to solve linearly implicit ODE systems. All the matrices involved are assumed to be sparse in structure. Either determines the sparsity structure or accepts it from the user, and uses parts of the Yale Sparse Matrix Package to solve the linear systems that arise, by a direct method.

A.4 Wrappers for other ODE software

Sympy_odefun
: Wrapper for the *sympy.mpmath.odefun* method, which applies a high-order Taylor series method.

Vode
: Wrapper for *scipy.integrate.ode.vode*, a wrapper to vode.f intends to solve initial value problem of stiff or nonstiff ode systems. Applies backward differential formulae for iteration.

Dopri
: Wrapper for *scipy.integrate.ode.dopri5*, which applies Dormand&Prince method order 5.

Dop853
: Wrapper for *scipy.integrate.ode.dopri5*, which applies Dormand&Prince method order 8(5,3).

RKC
: Wrapper for *rkc.f*, which is based on a family of explicit Runge-Kutta-Chebyshev formulas of order two.

RKF45
: Wrapper for *rkf45.f*, which is based on Runge-Kutta Fehlberg 4th-5th order.

Vode_pyds
: Wrapper for *Vode_ODESystem* in *PyDSTool*. Underlying numerical method is *vode.f*. If Jacobian matrix is not supplied by users, *PyDSTool.DIFF()* would be used to estimate Jacobian matrix approximately.

Appendix B

Complete code for examples

B.1 Logistic population model in Section 2.2.2

B.1.1 Basic usage in 2.2.2

```
import odesolvers
import numpy

f = lambda u,t : 0.8*u*(1.0 - u)
u0 = 0.5                          # initial value
# desired time range
time_points = numpy.arange(0.0, 7.0, 1.0)

method = odesolvers.RungeKutta4(f)
method.set_initial_conditions(u0)
u,t = method.solve(time_points)
```

B.1.2 Extra parameters for *f(u,t)* in Section 2.2.2

```
import odesolvers
import numpy

def f(u, t, a, C):              # f with extra parameters
    return a*u*(1.0 - u)/C
time_points = numpy.arange(0.0, 7.0, 1.0)
u0 = 0.5
C = 1.0                          # Population capacity

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

B.1.3 Stop event in Section 2.2.2

```
import odesolvers
import numpy

def f(u, t, a, C):              # f with extra parameters
    return a*u*(1.0 - u)/C
time_points = numpy.arange(0.0, 7.0, 0.01)
u0 = 0.5
C = 1.0                          # Population capacity

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,C))
    u_solutions[key], t = method.solve(time_points,
                                       terminate=_terminate)
```

B.2 Logistic population model in Section 2.2.3

B.2.1 Basic usage in Section 2.2.3

```
import odesolvers
import numpy

def f(u,t):          # Define ODE system with vectors
     u_0, u_1 = u
     return [u_1, 3.0*(1.0 - u_0*u_0)*u_1 + u_0]
u0 = [2.0, 0.0]       # Specify initial value as vector
time_points = [0.0, 2.0, 4.0, 6.0, 8.0, 10.0]

method = odesolvers.RungeKutta4(f)
method.set_initial_conditions(u0)
u,t = method.solve(time_points)
```

B.2.2 Switch between solvers in Section 2.2.3

```
import odesolvers
import numpy
def f(u,t):
     u_0, u_1 = u
     return [u_1, 3.0*(1.0 - u_0*u_0)*u_1 + u_0]
u0 = [2.0, 0.0]
time_points = [0.0, 2.0, 4.0, 6.0, 8.0, 10.0]
method = odesolvers.RungeKutta4(f)
method.set_initial_conditions(u0)
u,t = method.solve(time_points)

# A new instance applying method Lsode
method_new = method.switch_to(odesolvers.Lsode)
# Restart integration
u_new,t_new = method_new.solve(time_points)
```

B.3 Function func_wrapper() in Section 2.3.3

```
def func_wrapper(self):

    import numpy as numpy
    _legal_paras = self._legal_paras
    # Extract function parameters to be wrapped
    func_list = [[name, \
        _legal_paras[name].get('returnArrayOrder',None),
        _legal_paras[name].get('paralist_old',None),
        _legal_paras[name].get('paralist_new',None)]
        for name in _legal_paras \
        if hasattr(self, name) and \
           'type' in _legal_paras[name] and \
           _legal_paras[name]['type'] is callable and \
           ('paralist_new' in _legal_paras[name] or \
            'returnArrayOrder' in _legal_paras[name])]
    # 'paralist_new' in _legal_paras[name]
    #    --> new parameter list is defined to be wrapped
    # 'returnArrayOrder' in _legal_paras[name]
    #    --> this function return an array, and should be
    #        wrapped either in Fortran or C (default) order.

    func_input = {}
    for name, order, para_old, para_new in func_list:
        # e.g. name = 'jac',
        #      order = Fortran or None or C
        #      para_old = 'u, t'
        #      para_new = 't, u'
        func_input[name] = getattr(self,name)
        name_new = 'ftu' if ((name == 'f') and \
                         (not hasattr(self, 'ftu_fortran' ))) \
                         else name
        wrap_string = 'lambda %s: ' % \
            (para_new if para_new is not None else para_old)
        wrap_string += 'numpy.asarray(' if order is not None else ''
        wrap_string += 'func_input["%s"](%s)' % (name,para_old)
        wrap_string += ', order="Fortran"' if order=='Fortran' else ''
        wrap_string += ')' if order is not None else ''

        setattr(self,name_new,eval(wrap_string,locals()))
return None
```

B.4 Test case in Section 3.2 and chapter 5

```
from odesolvers import Lsoda
import numpy as np

def f5(u,t):
   udot = np.zeros(25,float)
   for j in range(5):
      for i in range(5):
         k = i+j*5
         udot[k] = -2.*u[k] + u[k - 1]*(i>0) + u[k - 5]*(j>0)
return (np.asarray(udot))

def _jac5(u,t,ml,mu):
   pd = np.zeros(6*25,float).reshape(6,25)
   pd[0,:], pd[1,:], pd[5,:], pd[1,4:24] = -2., 1., 1., 0.
   return pd

import sys
try:
   n_points = int(sys.argv[1])
except:
   n_points = 100   # default number of time-steps

t0, tn, u0 = 0., 4.,  [1]+24*[0]
dt = (tn-t0)/n_points
time_points = np.arange(t0, tn + dt, dt)

method = Lsoda(f5,rtol=1e-6,atol=1e-6,jac_banded=_jac5, ml=5, mu=0)
method.set_initial_condition(u0)
u,t = method.solve(time_points)
```

```
def f(u,t):
    udot = numpy.zeros(25,float)
    for j in range(5):
        for i in range(5):
            k = i+j*5
            udot[k] = -2.*u[k] + u[k - 1]*(i>0) + u[k - 5]*(j>0)
    return (numpy.asarray(udot))
t0, tn, u0 = 0., 4.,  [1]+24*[0]


import time
times = {}
t_points = numpy.arange(t0, tn, (tn - t0)/n_points)
for tol in (1e-2,1e-6,1e-10,1e-16):
    for solver in (RungeKuttaFehlberg, RKF45, Fehlberg_RK, Dopri5):
        time_start = time.time()
        method = solver(f, rtol=tol, atol=tol)
        method.set_initial_condition(u0)
        u,t = method.solve(t_points)
        time_stop = time.time()
        print '\n%s takes %.2g seconds when accuracy rtol=atol=%g' \
            % (solver.__name__, (time_stop - time_start),  tol)
```