Chapter 7 Solving Ordinary Differential Equations with Runge-Kutta Methods `

1. Objectives

In this chapter, we will explore Runge-Kutta mathods for solving ordinary differential equations. The goal is to gain a better understanding of some of the more popular Runge-Kutta methods and the corresponding numerical code.

Specifically you will be able to

- describe the mid-point method
- construct a Runge-Kutta tableau from equations or equations from a tableau
- describe how a Runge-Kutta method estimates truncation error
- edit a working Matlab code to use a different method or solve a different problem

2. Readings

There is no required reading for this chapter, beyond the contents of this chapter itself. However if you would like additional background on any of the following topics, the refer to the sections indicated below.

Runge-Kutta Methods:

– Press, et al. Press et al. (1992), Section 16.1

– Burden & Faires Burden and Faires (1981), Section 5.4

3. Solving Ordinary Differential Equations with the Runge-Kutta methods

Ordinary differential equations (ODEs) arise in many physical situations. For example, there is the first-order Newton cooling equation discussed in chapter 5, and perhaps the most famous equation of all, the Newton's Second Law of Mechanics F=ma.

In general, higher-order equations, such as Newton's force equation, can be rewritten as a system of first-order equations . So the generic problem in ODEs is a set of N coupled first-order differential equations of the form,

$$\frac{d\mathbf{y}}{dt} = f(\mathbf{y}, t) \tag{3.1}$$

where $\mathbf{y} = (y_1, ..., y_N)$.

For a complete specification of the solution, boundary conditions for the problem must be given. Typically, the problems are broken up into two classes:

- **Initial Value Problem (IVP)** : the initial values of $\mathbf{y}$ are specified.
- **Boundary Value Problem (BVP)** : $\mathbf{y}$ is specified at the initial and final times.

For this chapter, we are concerned with the IVP's. BVP's tend to be much more difficult to solve and involve techniques which will not be dealt with in this chapter.

Now as was pointed out in Ch. 5, in general, it will not be possible to find exact, analytic solutions to the ODE. However, it is possible to find an approximate solution with a finite difference scheme such as the forward Euler method . This is a simple first-order, one-step scheme which is easy to implement. However, this method is rarely used in practice as it is neither very stable nor accurate.

The high-order Taylor methods discussed in Ch. 5/6 are one alternative but involve higher-order derivatives that must be calculated by hand or worked out numerically in a multi-step scheme. Like the forward Euler method, stability is a concern.

The Runge-Kutta methods are higher-order, one-step schemes that makes use of information at different *stages* between the beginning and end of a step. They are more stable and accurate than the forward Euler method and are still relatively simple compared to schemes such as the multi-step predictor-corrector methods or the Bulirsch-Stoer method. Though they lack the accuracy and efficiency of these more sophisticated schemes, they are still powerful methods that almost always succeed for non-stiff IVPs.

3.1 The Midpoint Method: A Two-Stage Runge-Kutta Method

The forward Euler method takes the solution at time t_n and advances it to time t_{n+1} using the value of the derivative $f(y_n, t_n)$ at time t_n

$$y_{n+1} = y_n + hf(y_n, t_n)$$

where $h \equiv \Delta t$.

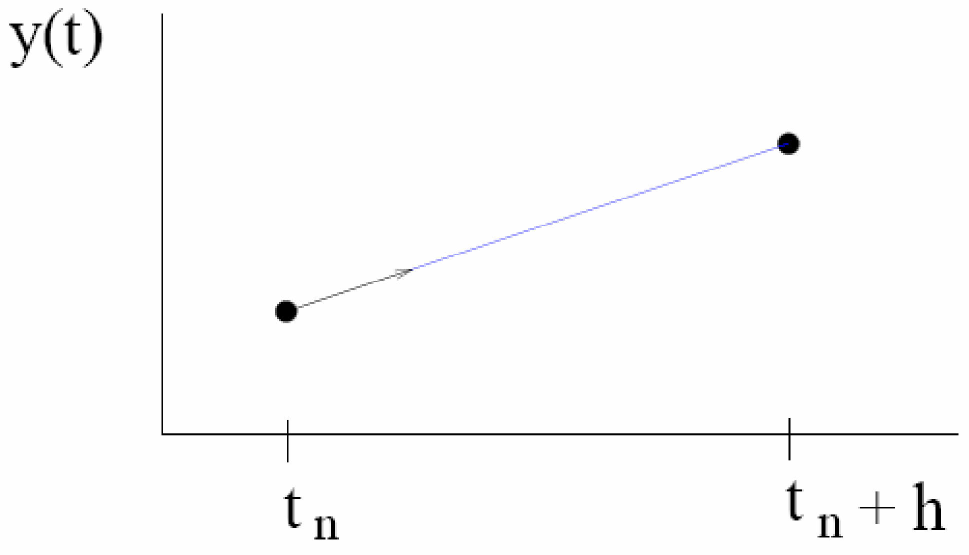


Figure 1: The forward Euler method is essentially a straight-line approximation to the solution, over the interval of one step, using the derivative at the starting point as the slope.

The idea of the Runge-Kutta schemes is to take advantage of derivative information at the times between t_n and t_{n+1} to increase the order of accuracy.

For example, in the midpoint method, the derivative at the initial time is used to approximate the derivative at the midpoint of the interval, $f(y_n + \frac{1}{2}hf(y_n, t_n), t_n + \frac{1}{2}h)$. The derivative at the midpoint is then used to advance the solution to the next step. The method can be written in two *stages* k_i,

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \tfrac{1}{2}k_1, t_n + \tfrac{1}{2}h) \\ y_{n+1} &= y_n + k_2 \end{aligned} \tag{3.2}$$

The midpoint method is known as a 2-stage Runge-Kutta formula.

3.2 Second-Order Runge-Kutta Methods

As was shown in lab 2 , the error in the forward Euler method is proportional to h. In other words, the forward Euler method has an accuracy which is *first order* in h.

The advantage of the midpoint method is that the extra derivative information at the midpoint results in the first order error term cancelling out, making the method *second order* accurate. This can be shown by a Taylor expansion of equation (3.2).

Problem 1: Even though the midpoint method is second-order accurate, it may still be less accurate than the forward Euler method. In the demo below, compare the accuracy of the two methods

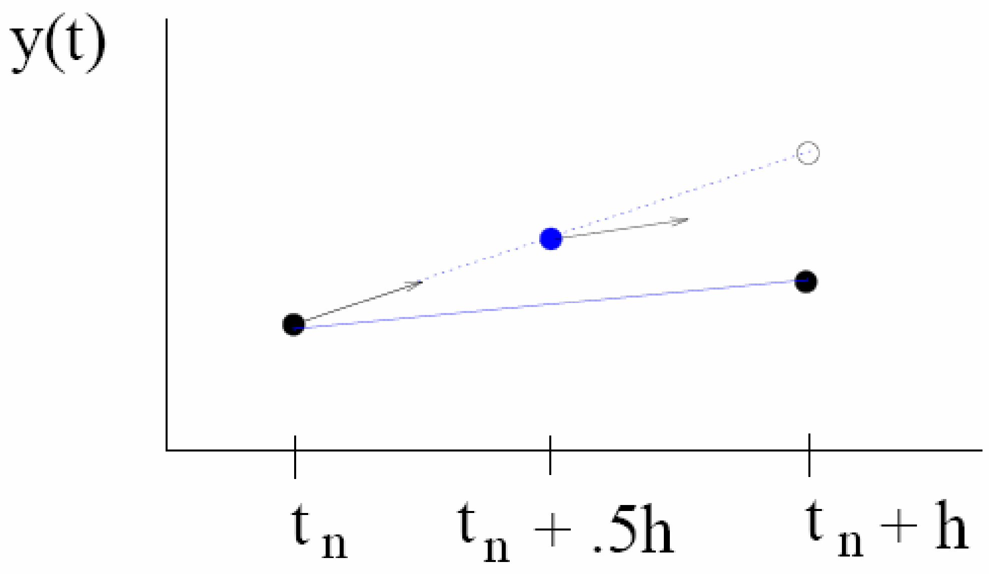


Figure 2: The midpoint method again uses the derivative at the starting point to approximate the solution at the midpoint. The derivative at the midpoint is then used as the slope of the straight-line approximation.

on the initial value problem

$$\frac{dy}{dt} = -y + t + 1, \quad y(0) = 1 \tag{3.3}$$

which has the exact solution

$$y(t) = t + e^{-t} \tag{3.4}$$

1. Why is it possible that the midpoint method may be less accurate than the forward Euler method, even though it is a higher order method?
2. Based on the numerical solutions of (3.3), which method appears more accurate?
3. Cut the stepsize in half and check the error at a given time. Repeat a couple of more times. How does the error drop relative to the change in stepsize?
4. How do the numerical solutions compare to $y(t) = t + e^{-t}$ when you change the initial time? Why?

In general, an *explicit* 2-stage Runge-Kutta method can be written as,

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + b_{21}k_1, t_n + a_2 h) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 \end{aligned} \tag{3.5}$$

The scheme is said to be *explicit* since a given stage does not depend *implicitly* on itself, as in the backward Euler method , or on a later stage.

Other explicit second-order schemes can be derived by comparing the formula (3.5) to the second-order Taylor method and matching terms to determine the coefficients a_2, b_{21}, c_1 and c_2.

Mathematical Note: See Appendix A.1 for the derivation of the midpoint method.

3.3 The Runge-Kutta Tableau

A general s-stage Runge-Kutta method can be written as,

$$\begin{aligned} k_i &= hf(y_n + \sum_{j=1}^{s} b_{ij}k_j, t_n + a_i h), \quad i = 1, ..., s \\ y_{n+1} &= y_n + \sum_{j=1}^{s} c_j k_j \end{aligned} \tag{3.6}$$

An *explicit* Runge-Kutta method has $b_{ij} = 0$ for $i \le j$, i.e. a given stage k_i does not depend on itself or a later stage k_j.

The coefficients can be expressed in a tabular form known as the Runge-Kutta tableau.

i	a_i	b_{ij}				c_i
1	a_1	b_{11}	b_{12}	...	b_{1s}	c_1
2	a_2	b_{21}	b_{22}	...	b_{2s}	c_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
s	a_s	b_{s1}	b_{s2}	...	b_{ss}	c_s
$j =$		1	2	...	s	

An explicit scheme will be strictly lower-triangular.

For example, a general 2-stage Runge-Kutta method,

$$\begin{aligned} k_1 &= hf(y_n + b_{11}k_1 + b_{12}k_2, t_n + a_1 h) \\ k_2 &= hf(y_n + b_{21}k_1 + b_{22}k_2, t_n + a_2 h) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 \end{aligned} \tag{3.7}$$

has the coefficients,

i	a_i	b_{ij}		c_i
1	a_1	b_{11}	b_{12}	c_1
2	a_2	b_{21}	b_{22}	c_2
$j =$		1	2	

In particular, the midpoint method is given by the tableau,

i	a_i	b_{ij}		c_i
1	0	0	0	0
2	$\frac{1}{2}$	$\frac{1}{2}$	0	1
$j =$		1	2	

Problem 2: Write out the tableau for

1. Heun's method (A.23)
2. the fourth-order Runge-Kutta method (3.8) discussed in the next section.

3.4 Explicit Fourth-Order Runge-Kutta Method

Explicit Runge-Kutta methods are popular as each stage can be calculated with one function evaluation. In contrast, implicit Runge-Kutta methods usually involves solving a non-linear system of equations in order to evaluate the stages. As a result, explicit schemes are much less expensive to implement than implicit schemes.

However, there are cases in which implicit schemes are necessary and that is in the case of *stiff* sets of equations. See section 16.6 of Press et al. (1992) for a discussion. For these labs, we will focus on non-stiff equations and on explicit Runge-Kutta methods.

The higher-order Runge-Kutta methods can be derived by in manner similar to the midpoint formula. An s-stage method is compared to a Taylor method and the terms are matched up to the desired order.

Methods of order $M > 4$ require $M+1$ or $M+2$ function evaluations or stages, in the case of explicit Runge-Kutta methods. As a result, fourth-order Runge-Kutta methods have achieved great popularity over the years as they require only four function evaluations per step. In particular, there is the classic fourth-order Runge-Kutta formula:

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \tfrac{k_1}{2}, t_n + \tfrac{h}{2}) \\ k_3 &= hf(y_n + \tfrac{k_2}{2}, t_n + \tfrac{h}{2}) \\ k_4 &= hf(y_n + k_3, t_n + h) \\ y_{n+1} &= y_n + \tfrac{k_1}{6} + \tfrac{k_2}{3} + \tfrac{k_3}{3} + \tfrac{k_4}{6} \end{aligned} \tag{3.8}$$

Problem 3: In the demo below, compare compare solutions to the test problem (3.3)

$$\frac{dy}{dt} = -y + t + 1, \quad y(0) = 1$$

generated with the fourth-order Runge-Kutta method to solutions generated by the forward Euler and midpoint methods.

1. Based on the numerical solutions of (3.3), which of the three methods appears more accurate?
2. Again determine how the error changes relative to the change in stepsize, as the stepsize is halved.

4 Numerical ODE Routines

We will be implementing the Runge-Kutta method in Octave, which has some support for simple functional (as opposed to object-oriented) programming. Our Octave script will have three components:

- **Algorithm**

  First, a routine to implement the Runge-Kutta algorithm (3.6) which specifies how to advance a solution y_n, at time t, to y_{n+1}, at time $t+h$.

- **User defined function**

  The user of the numerical routines specifies the problem to be solved with a routine that evaluates the derivatives f (the right-hand-side of the ODE system (3.6)).

- **Driver**

  The driver routine reads in the initial conditions, timestep and the coefficients for the user-defined function, and loops the algorithm from the starting to ending times.

4.1 Simple Example: Functional Approach

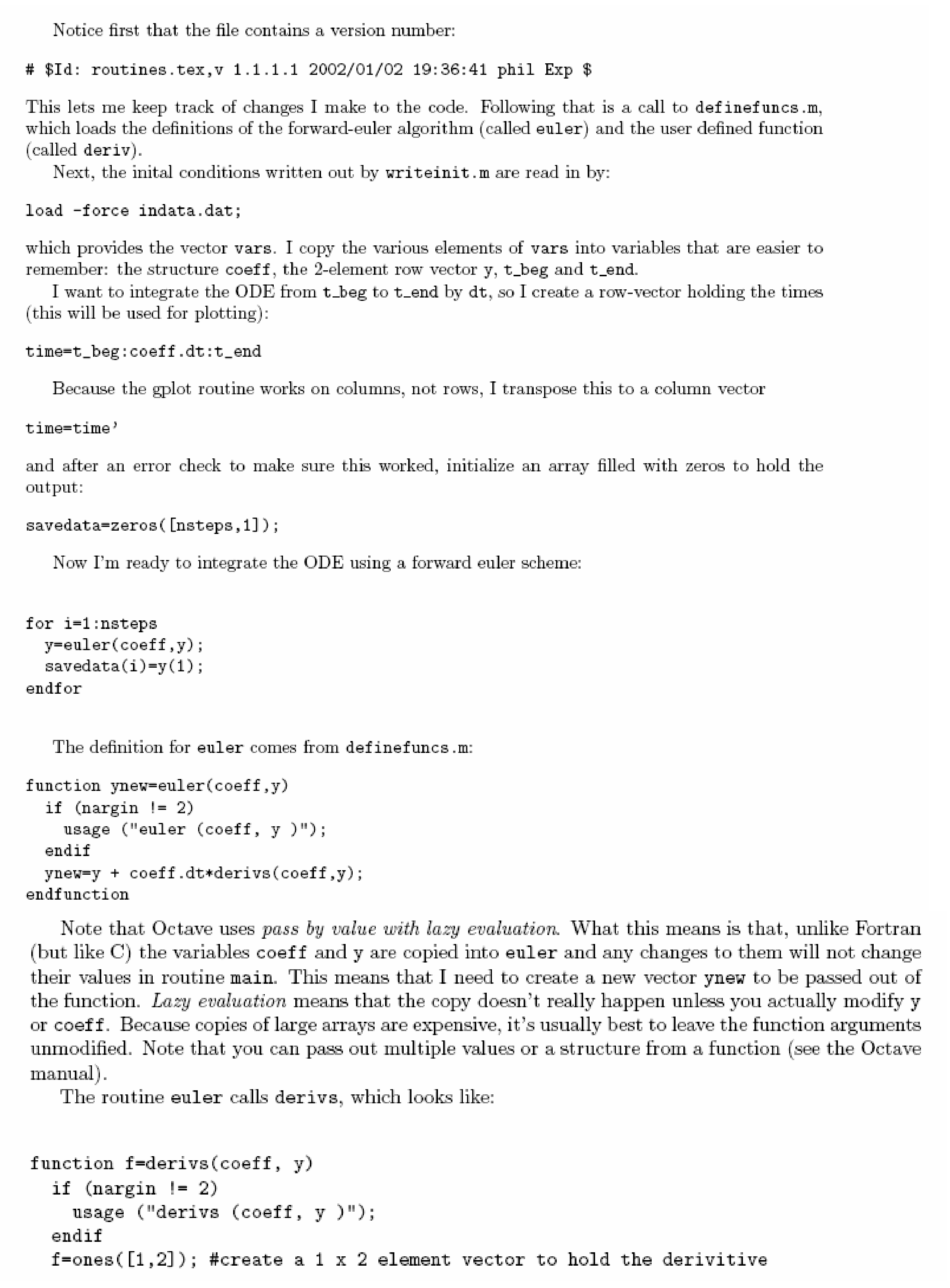
Here's a brief example written in Octave which solves the the damped, simple harmonic ocillator using the forward Euler method. The code discussed below can be found in the file lab4.tar[4], which is in the Unix tar format. To unpack these scripts in your directory, open a bash shell and type:

This should extract writeinit.m[5], main.m[6] and definefuncs.m[7] into your home directory.
First, use emacs to take a look at writeinit.m, which should look like:

```
vars    =[0.,        100., 0.1,  0.  , 1.,  0. , 1. ];
varnames=["t_beg";"t_end";"dt";"c1";"c2"; "y1";"y2"];
indata_descrip = "input variables for example1";
save -ascii indata.dat vars varnames indata_descrip
```

This script, when run by typing `writeinit` from the octave prompt, should produce the file `indata.dat`. This file is then read into octave to set the start and stop times for the integration (`t_beg, t_end`) the timestep `dt`, the coefficients for the derivitive function (`c1, c2`), and the initial conditions at the start of the integration (`y1,y2`).
The driver `main.m` looks like:

```
# $Id: routines.tex,v 1.1.1.1 2002/01/02 19:36:41 phil Exp $
#define the functions derivs, euler, midpoint and rk4ODE
definefuncs;
#load the input data writen by writeinit.m
load -force indata.dat;
t_beg=vars(1);
t_end=vars(2);
coeff.dt=vars(3);  #create a structure coeff to hold values to be passed to
                   #the ode function
coeff.c1=vars(4);
coeff.c2=vars(5);
y=zeros([1,2]);
y(1)=vars(6);
y(2)=vars(7);
time=t_beg:coeff.dt:t_end;  #create the time vector
time=time';                 #transpose it from [1, nsteps] to [nsteps, 1]
                            #we need to do this because gplot plots
                            #columns, not rows
nsteps=rows(time);

if(nsteps <= 2)             #here's and example of rudimentary error checking
  error("need at least two steps");
endif

#create a column vector to hold y at each timestep
#for later plotting

savedata=zeros([nsteps,1]);

for i=1:nsteps
  y=euler(coeff,y);
  savedata(i)=y(1);
endfor

data=[time,savedata];
gplot data with linespoints
```



Notice first that the file contains a version number:

```
# $Id: routines.tex,v 1.1.1.1 2002/01/02 19:36:41 phil Exp $
```

This lets me keep track of changes I make to the code. Following that is a call to `definefuncs.m`, which loads the definitions of the forward-euler algorithm (called `euler`) and the user defined function (called `deriv`).

Next, the inital conditions written out by `writeinit.m` are read in by:

```
load -force indata.dat;
```

which provides the vector `vars`. I copy the various elements of `vars` into variables that are easier to remember: the structure `coeff`, the 2-element row vector `y`, `t_beg` and `t_end`.

I want to integrate the ODE from `t_beg` to `t_end` by `dt`, so I create a row-vector holding the times (this will be used for plotting):

```
time=t_beg:coeff.dt:t_end
```

Because the gplot routine works on columns, not rows, I transpose this to a column vector

```
time=time'
```

and after an error check to make sure this worked, initialize an array filled with zeros to hold the output:

```
savedata=zeros([nsteps,1]);
```

Now I'm ready to integrate the ODE using a forward euler scheme:

```
for i=1:nsteps
  y=euler(coeff,y);
  savedata(i)=y(1);
endfor
```

The definition for `euler` comes from `definefuncs.m`:

```
function ynew=euler(coeff,y)
  if (nargin != 2)
    usage ("euler (coeff, y )");
  endif
  ynew=y + coeff.dt*derivs(coeff,y);
endfunction
```

Note that Octave uses *pass by value with lazy evaluation*. What this means is that, unlike Fortran (but like C) the variables `coeff` and `y` are copied into `euler` and any changes to them will not change their values in routine `main`. This means that I need to create a new vector `ynew` to be passed out of the function. *Lazy evaluation* means that the copy doesn't really happen unless you actually modify `y` or `coeff`. Because copies of large arrays are expensive, it's usually best to leave the function arguments unmodified. Note that you can pass out multiple values or a structure from a function (see the Octave manual).

The routine `euler` calls `derivs`, which looks like:

```
function f=derivs(coeff, y)
  if (nargin != 2)
    usage ("derivs (coeff, y )");
  endif
  f=ones([1,2]); #create a 1 x 2 element vector to hold the derivitive
  f(1)=y(2);
  f(2)= -1.*coeff.c1*y(2) - coeff.c2*y(1);
endfunction
```

You should convince yourself that with the values of `c1` and `c2` given in `indata.dat`, this represents a 2nd order ODE with an easy-to-find analytic solution.

Problem 5: Try out `main.m`:

1. Note that the parameter c_1 has been set to zero.
   (a) What kind of solutions would you expect for $y(1)$?
   (b) What kind of numerical solutions do you get? Hand in some plots of $y(1)$.
   (c) How do you explain this discrepancy?
2. Note that the file `define_funcs.m` also contains implementations for the mid-point and fourth-order Runge-Kutta methods. Modify the scripts to call these instead, and hand in plots of the results. Is there and improvement? Why?

Problem 6: The source code[8] discussed in Section 4 solved the damped, harmonic oscillator with the forward Euler method.

1. Write a new routine that implements using Heun's method (A.23) along the lines of the routines in define_funcs.m[9].
2. Write a new driver main.m[10] that solves the Newton cooling equation described in lab 1[11].

Hand in a print out of the code.

Problem 7:

1. Solve the damped oscillator with the Midpoint method which is already part of `definefuncs.m`
2. Solve the problem with Heun's method (A.23) (you will have to add this method to `definefuncs.m` first).
3. Compare with the solution generated by the midpoint method.
4. Now solve the following test equation by both the midpoint and Heun's method and compare.

$$f(y,t) = t - y + 1.0$$

   Choose two sets of initial conditions and investigate the behaviour.
5. Is there any difference between the two methods when applied to either problem? Should there be? Explain.
6. Add the Newton cooling equation and solve it by any of the above methods. Compare with solutions generated in the demo .

7. Hand in some sample plots along with the parameter values and initial conditions used.

5. Runge-Kutta methods implemented in Matlab

The Runga-Kutta method is halfway between a 4th order and 5th order method. Consequently, the inbuilt Matlab program to run the method is called **ODE45**. The syntax to solve an initial value problem with the ODE45 routine in Matlab is:

- Mathematical formulation: $\frac{dy}{dt} = f(t, y), \quad y(0) = y_0 \quad 0 \le t \le T$
- Matlab formulation: `>> ode45('func', [0,T],y0)`

where **func** is a Matlab M-file function which accepts variables (t,y) and returns $f(t,y)$.

Example: Use ODE45 to find a numerical solution to $\frac{du}{dt} = ku\,(1\text{-}u)$, over the time interval [0,100] where $k = 0.2$ and $u(0) = 0.001$. This equation is a common modeling tool for epidemiology, growth of marketshare and population dynamics.

To solve the problem:

Firstly construction a function representing the right-hand-side of the equation. For example, if we call it **RHS.m**, then it will be something like:

```
function output = RHS(t,u)

%  Evaluates the rhs of the ode

k = 0.2;

output = k*u*(1-u);
```

Secondly, call the ODE45 procedure from the *Command Window*. You could alternatively store the commands in an M-file and run them that way.

```
>> ode45('RHS' , [0,100] , 0.001)
```

Thirdly, to visualize the solution, you can produce a tabulated data or a graph. The procedure ODE45 should produce a graph for you automatically. Alternatively, run the program again with the command:

```
>> [t,u] = ode45('RHS' , [0,100] , 0.001);
```

The semicolon will suppress output. The solution will go into the vectors t and u. Now, to view the solution as data or graphically call, at the *Command Window*,

```
>> [t,u]
```

or

```
>> plot(t,u)
```

Exercise: Find the initial condition y_0, so that the initial value problem:

$$\frac{du}{dt} = y\,(t - y), \quad y(0) = y_0,$$

satisfies $y(1) = 0.5$. You might use trial-and-error or a more sophisticated method.

6. Runge-Kutta method for systems of ODEs

The numerical schemes have so far solved first order differential equations. However, it is possible to solve second order ODEs, for example

$$\frac{d^2y}{dt^2} + \sin(y) = 0\,, \qquad y(0) = 1,\ dy/dt(0) = 0 \tag{1}$$

and coupled systems of ODEs, for example

$$\begin{aligned} \frac{dx}{dt} &= x + y \\ \frac{dy}{dt} &= y - 3x \qquad x(0) = 1,\ y(0) = -1 \end{aligned} \tag{2}$$

(a) Systems of ODEs

We will start with *systems* of equations. The syntax remains very similar to the scalar case.

Example: Solve the system (2) with ODE45 over $0 \le t \le 10$.

This time, the function for the right hand side must accept variables t and (x,y). The form of ODE45 requires that the (x,y) variables arrive as a single vector, and we call it v here. The function should also return a (column) vector with two components, as the right hand side has two parts.

```
function output = systemRHS(t,v)
%  Evaluates the rhs of the coupled ode system
x = v(1);
y = v(2);
output1 = x + y;
output2 = y - 3*x;
output=[output1; output2];
```

Then at the *Command Window*, again call ODE45:

```
>> [t,v] = ode45('systemRHS' , [0,10] , [1, -1]);
```

This time, the initial conditions x(0)=1 and y(0)=-1 are described by a *vector*. The output will be a vector of timesteps, t, as well as vector v which describes x and y.

If you want to see x against time, then call

```
>> plot(t, v(:,1))
```

If you want to see y against time, then call

```
>> plot(t, v(:,2))
```

If you want to see x against y (called a phase plot) then call

```
>> plot(v(:,1), v(:,2))
```

A Mathematical Notes

A.1 Note on the Derivation of the Second-Order Runge-Kutta Methods

A general s-stage Runge-Kutta method can be written as,

$$\begin{aligned} k_i &= hf(y_n + \sum_{j=1}^{s} b_{ij}k_j, t_n + a_i h), \quad i = 1, ..., s \\ y_{n+1} &= y_n + \sum_{j=1}^{s} c_j k_j \end{aligned} \tag{A.13}$$

where $\sum_{j=1}^{s} b_{ij} = a_i$.

In particular, an *explicit* 2-stage Runge-Kutta method can be written as,

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + ak_1, t_n + ah) \\ y_{n+1} &= y_n + c_1 k_1 + c_2 k_2 \end{aligned} \tag{A.14}$$

where $b_{21} = a_2 \equiv a$. So we want to know what values of a, c_1 and c_2 leads to a second-order method, i.e. a method with an error proportional to h^3.

To find out, we compare the method against a second-order Taylor method,

$$y(t_n + h) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + O(h^3) \tag{A.15}$$

So for the y_{n+1} to be second-order accurate, it must match the Taylor method (A.15). In other words, $c_1 k_1 + c_2 k_2$ must match $hy'(t_n) + \frac{h^2}{2}y''$. To do this, we need to express k_1 and k_2 in terms of derivatives of y at time t_n.

First note, $k_1 = hf(y_n, t_n) = hy'(t_n)$.

Next, we can expand k_2 about $(y_n.t_n)$,

$$k_2 = hf(y_n + ak_1, t_n + ah) = h(f + haf_t + haf_y y' + O(h^2)) \tag{A.16}$$

However, we can write y'' as,

$$y'' = \frac{df}{dt} = f_t + f_y y' \tag{A.17}$$

This allows us to rewrite k_2 in terms of y'',

$$k_2 = h(y' + hay'' + O(h^2)) \tag{A.18}$$

Substituting these expressions for k_i back into the Runge-Kutta formula gives us,

$$y_{n+1} = y_n + c_1 hy' + c_2 h(y' + hay'') \tag{A.19}$$

or

$$y_{n+1} = y_n + h(c_1 + c_2)y' + h^2(c_2 a)y'' \tag{A.20}$$

If we compare this against the second-order Taylor method, (A.15), we see that we need,

$$\begin{aligned} c_1 + c_2 &= 1 \\ ac_2 &= \tfrac{1}{2} \end{aligned} \tag{A.21}$$

for the Runge-Kutta method to be second-order.

If we choose $a = 1$, this implies $c_2 = 1$ and $c_1 = 0$. This gives us the midpoint method.

However, note that other choices are possible. In fact, we have a *one-parameter family* of second-order methods. For example if we choose, $a = 1$ and $c_1 = c_2 = \frac{1}{2}$, we get the *modified Euler method*,

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + k_1, t_n + h) \\ y_{n+1} &= y_n + \tfrac{1}{2}(k_1 + k_2) \end{aligned} \tag{A.22}$$

while the choice $a = \frac{2}{3}$, $c_1 = \frac{1}{4}$ and $c_2 = \frac{3}{4}$, gives us *Heun's method*,

$$\begin{aligned} k_1 &= hf(y_n, t_n) \\ k_2 &= hf(y_n + \tfrac{2}{3}k_1, t_n + \tfrac{2}{3}h) \\ y_{n+1} &= y_n + \tfrac{1}{4}k_1 + \tfrac{3}{4}k_2 \end{aligned} \tag{A.23}$$

References

Burden, R. L. and J. D. Faires, 1981: *Numerical Analysis*. PWS-Kent, Boston, 4th edition.

Press, W. H., S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, 1992: *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2nd edition.

Glossary

driver: A routine that calls the other routines to solve the problem.

embedded Runge-Kutta methods: Two Runge-Kutta methods that share the same stages. The difference between the solutions give an estimate of the local truncation error.

explicit: In an explicit numerical scheme, the calculation of the solution at a given step or stage does not depend on the value of the solution at that step or on a later step or stage.

fourth-order Runge-Kutta method: A popular fourth-order, four-stage, explicit Runge-Kutta method.

implicit: In an implicit numerical scheme, the calculation of the solution at a given step or stage does depend on the value of the solution at that step or on a later step or stage. Such methods are usually more expensive than implicit schemes but are better for handling stiff ODEs.

midpoint method: A two-stage, second-order Runge-Kutta method.

stages: The approximations to the derivative made in a Runge-Kutta method between the start and end of a step.

tableau: The tableau for a Runge-Kutta method organizes the coefficients for the method in tabular form.