Solving Ordinary Differential Equations with Matlab

September 3, 1996

1 Printing and Saving Graphics Output

When plotting requests are made to Matlab, the program opens a separate window called a *figure* window. It is possible to have more than one *figure* window open at one time. To print the output from the current *figure* window, choose the **Print** option from the **File** menu. It is also possible to have the graphics output written to disk as a postscript file by choosing **Print To File** from within the **Print** dialog box.

2 An Introductory Matlab Session

In this section we present a few commands to introduce some features of Matlab to the new user. After starting up Matlab, enter these commands into the *command* window and observe the results.

2.1 Getting On-Line Help

>> **more on**	%% turn on page control to prevent information from scrolling off the page
>> **help**	%% get a list of Matlab topics
>> **help elfun**	%% get a list of elementary math functions
>> **help path**	%% get help on appending the **path** variable
>> **help plot**	%% get info on **plot** command
>> **help ode45**	%% get info on ODE solver **ode45**

2.2 Matrix and Vector Manipulation

>> **A = [1 2 3 ; 4 5 6 ; 7 8 9]**	%% 3x3 matrix; use semicolon to separate rows
>> **M = [-1 6 3 ; 0 2 -3] ;**	%% 2x3 matrix; Note that the trailing semicolon is used to prevent the result from being echoed to the screen.
>> **B = A' ;**	%% matrix transpose; if elements are complex this involves taking complex conjugate.
>> **B**	%% echo the contents of **B**

>> **x = [1 2 3]**	%% row vector; can also use comma to separate elements within a row
>> **y = [1 ; 2 ; 3]**	%% column vector; semicolon delimits rows
>> **x'**	%% transpose of row vector $\rightarrow$ column vector
>> **xc = x(:)**	%% arrange as a column vector; this has no effect if already a column vector
>> **x1 = [0 : 10]**	%% initialize a row vector (11 elements); default increment = 1
>> **x1(3)**	%% the third element in the vector **x1**
>> **x2 = [0 : 0.1 : 10]**	%% initialize a row vector using 0.1 increments (1 row by 101 columns)
>> **size(A)**	%% get the dimensions of the matrix **A**
>> **size(x)**	%% get the dimensions of the vector **x**
>> **length(x)**	%% get the length of the vector **x**

2.3 Matrix Arithmetic

These operations follow the rules of linear algebra and matrix arithmetic.

>> **A * x**	%% wrong dimensions; $(3\times3)*(1\times3)$
>> **A * y**	%% correct dimensions; $(3\times3)*(3\times1) \rightarrow 3\times1$ (column vector)
>> **x * y**	%% scalar product; $(1\times3)*(3\times1) \rightarrow 1\times1$ (scalar)
>> **y * x**	%% $(3\times1)*(1\times3) \rightarrow 3\times3$ matrix
>> **A * B**	%% $(3\times3)*(3\times3) \rightarrow 3\times3$ matrix
>> **A + B**	%% matrix addition; dimensions must agree
>> **z = A \ y**	%% solution to the linear equation **A * z = y**
>> **3.0 * A**	%% multiply each element of **A** by 3.0

2.4 Array Arithmetic

These operations are applied element by element so the dimensions of the two matrices must agree exactly.

>> **A .* B**	%% array multiplication; still a 3x3 matrix %% but the arithmetic is performed element by element.
>> **x .* y**	%% dimensions must agree for array arithmetic
>> **x .* y'**	%% that's better; now a 1x3 row vector
>> **A.'**	%% matrix transpose without taking complex conjugate.

Can also use right division (./), left division (.\), and exponentiation (.^).

2.5 Elementary Plotting Commands

>> **clear A**	%% free up the memory being used by matrix **A**
>> **clear**	%% clear all vectors, arrays... (fresh start)
>> **x = [0 : 0.1 : 10];**	%% row vector (101 elements)
>> **y = sin(5 * x);**	%% another row vector
>> **z = cos(5 * x);**	
>> **plot(x , y)**	%% 2D plot of y as a function of x (opens *figure* window)
>> **plot(x , z)**	%% 2D plot of z as a function of x (erases previous plot)
>> **clf**	%% clear graphics screen
>> **plot(x , y)**	%% try again
>> **hold on**	%% this time hold onto previous plot
>> **plot(x , z , 'r--')**	%% plot the second curve using a red dashed line
>> **xlabel('X')**	%% label the x-axis (must use single quotes '...' to identify string constants)
>> **ylabel('Y')**	%% label the y-axis

```
>>  title('How about that !')     %% add a title

>>  zoom on                        %% enable 'zoom' feature; zoom in by clicking
                                   %% the mouse in the figure

>>  zoom out                       %% to restore figure to its original scale
                                   %% (or double click with the mouse)
```

3 M-files

We give a brief introduction to the use of Matlab M-files. There are two fundamental uses for M-files:

1) as a *script* file containing a sequence of Matlab commands

2) for defining new functions

3.1 M-files as Script Files

We use the Euler method to solve an initial value problem for the scalar logistic equation,

$$N'(t) = (r - aN)\,N \qquad\qquad N(t_0) = N_0.$$

Begin by creating a new file which will contain the sequence of Matlab commands necessary to perform this task. For Matlab running under DOS-Windows or on a Mac, use the **FILE** pull-down menu and select **New M-File**. This loads an editor and we enter the following text. Note that the percent symbol (%) denotes a comment which will be ignored by Matlab.

```
%%%%%%%%%%%%%%%%%%  logistic.m  %%%%%%%%%%%%%%%%%%
%%
%%  A sample M-file for solving the scalar logistic equation
%%  using the Euler method.
%%
%%          N' = (r - a*N)*N             N(t0) = N0

%% define the parameters at top of file
r = 1.0;                %% intrinsic growth rate
a = 0.1;                %% intraspecific competition
t0 = 0.0;               %% initial time
N0 = 1.0;               %% initial value, N(t0)
h = 0.10;               %% integration stepsize
nsteps = 100;           %% number of integration steps.
K = r/a;                %% for convenience

clear  t  N  s          %% best to first clear the memory
t(1) = t0;              %% t contains the timesteps
N(1) = N0;              %% N contains the numerical solution
s(1) = N0;              %% s contains the exact solution (for comparison)

%% begin the integration loop
for i = 1:nsteps,
  t(i+1) = t(i) + h;                     %% advance time
  slope = (r - a*N(i))*N(i);             %% direction vector
  N(i+1) = N(i) + h*slope;               %% next solution iterate
  s(i+1) = (N0*K) / (N0 + (K - N0)*exp(-r*t(i+1)));  %% exact solution
end;

%% now plot the results, overlaying the numerical and exact solutions
clf;                    %% clear the graphics window
plot(t,N,'y');          %% plot numerical solution using solid yellow line
```

```
hold on;                %% hold onto first plot
plot(t,s,'w--');        %% plot exact solution using dashed white line

%%%%%%%%%%%%%%%%%%%%  End of logistic.m  %%%%%%%%%%%%%%%%%%%%
```

The sequence of commands in `logistic.m` includes specifying equation parameters, initial conditions, integration step size, performing the Euler method and finally, plotting the results. To run this set of commands, just enter the command **logistic** in the command window. Now we are able to run numerical experiments on this model with minimal input from the command window. To solve the equation with a different set of parameters just return to the M-file window, make the necessary changes in the file, save the changes and execute the M-file.

3.2 M-files for User Defined Functions

In order to use the ODE solvers provided by Matlab we must provide a function for calculating the vector field $f(t, x)$, that is, the right hand side of the differential equation $x'(t) = f(t, x)$. As an example, consider solving the same logistic equation using the Matlab routine **ode23** to be discussed in the next section. We must first define the vector field function **vfLog** which calculates $f(N) = (r - aN)\,N$. The name of the M-file will be the same as the name we choose for the function plus the `.m` filename extension. The first uncommented line of the M-file must contain the function declaration indicating the inputs and outputs. Open a new M-file (to be named `vfLog.m`) and create the following file:

```
%%%%%%%%%%%%%%%%%%%%  vfLog.m  %%%%%%%%%%%%%%%%%%%%
function [f] = vfLog(t, N)

%%  vector field for logistic equation (single species).
%%
%%       N'(t) = f(t,N) = (r - a*N)*N

r = 1.0;                 %% intrinsic growth rate
a = 0.1;                 %% intraspecific competition

f = (r - a*N)*N;

%%%%%%%%%%%%%%%%%%%%  End of vfLog.m  %%%%%%%%%%%%%%%%%%%%
```

4 Using Matlab's ODE Routines

Matlab provides two functions for the numerical solution of systems of first order ordinary differential equations, **ode23** and **ode45**. In order to use these ODE solvers we must provide a function for the vector field as described previously. The Matlab ODE functions are variable-step methods, meaning that the step size is automatically varied to achieve the specified solution accuracy in an efficient manner. This is in contrast to fixed-step methods such as the Euler and fourth order Runge-Kutta algorithms provided in section 6.

4.1 Solving a Scalar First Order ODE

Consider solving the following initial value problem for the logistic growth model on the interval $0 \le t \le 10$,

$$X'(t) = (1.0 - 0.1\,X)\,X \qquad\qquad X(0) = 1.0$$

The M-file `vfLog.m`, as described in section 3.2, calculates the right hand side of the differential equation.

>> **[t, X] = ode23('vfLog', 0.0, 10.0, 1.0) ;**

The inputs to **ode23** are:

1. the name of the function defining the vector field = **'vfLog'**
2. the initial time t_0 = **0.0**
3. the final time t_f = **10.0**
4. the initial value $X(t_0)$ = **1.0**

The outputs from **ode23** are:

1. **t** is a *column* vector containing the integration times used by the ODE solver. If n_t is the total number of timesteps, then $\mathbf{t}(1) = t_0$ and $\mathbf{t}(n_t) = t_f$.
2. **X** is a matrix containing the numerical solution. Because $X(t)$ is a scalar function in this problem, the matrix **X** is a *column* vector of length n_t, where $\mathbf{X}(i) = X(\mathbf{t}(i))$. More generally, if the solution variable $X(t)$ is a vector of length n, the solution matrix **X** has n_t rows and n columns, with row i containing the n-dimensional solution vector $X = (X_1, \ldots, X_n)$ at time $\mathbf{t}(i)$.

To view a two-dimensional plot of the numerical solution,

>> **plot(t, X)** %% t on horizontal axis, $X(t)$ on vertical axis

To make this process easier to repeat, create the following M-file and save it under the name `log23.m`. This file also computes the exact solution using the function **solLog** (see section 3 for the code).

```
%%%%%%%%%%%%%%%%%%%%  log23.m  %%%%%%%%%%%%%%%%%%%%
%%
%%  Solves scalar logistic equation using Matlab's ode23().
%%
%%              X' = (r - a*X)*X              X(t0) = X0

t0 = 0.0;               %% initial time
tf = 10.0;              %% final time
X0 = 1.0;               %% initial value

clear  t  X  s          %% clear the arrays
```

```
[t, X] = ode23('vfLog', t0, tf, X0);     %% numerical solution
s = solLog(t, X0);                       %% exact solution

%% plot the results; overlay numerical and exact solutions
clf;                    %% clear the graphics window
plot(t,X,'y');          %% plot numerical solution using yellow
hold on;                %% hold onto first plot
plot(t,s,'w--');        %% plot exact solution using dashed white line

%%%%%%%%%%%%%%%%%%%%  End of log23.m  %%%%%%%%%%%%%%%%%%%%
```

4.2 Solving a System of ODEs

In this section we show how to solve a system of ordinary differential equations. The numerical methods for solving ODEs are for first order systems so higher order scalar equations must be converted to equivalent systems of first order equations in the usual way. The M-file `vfPend.m` presented in section 5 defines the vector field for the linear oscillator model with damping and periodic external forcing (see Boyce and DiPrima, §3.8–3.9).

The following script file `pend45.m` shows how to solve the linear oscillator equation using the Matlab routine **ode45** and plot the resulting solution. The main thing to note is how we handle the higher dimensional solution matrix where the solution $X(t)$ is now a vector of length 2. The first component of the solution vector, $X_1(t)$, is given by the first column of the solution matrix, **X**[**:** , **1**], and the second component, $X_2(t)$, is given by **X**[**:** , **2**]. To solve the ODE with a different set of equation parameters, the user must make the necessary changes in the M-file `vfPend.m`.

```
%%%%%%%%%%%%%%%%%%%%  pend45.m  %%%%%%%%%%%%%%%%%%%%
%%
%%  Solves linear oscillator equation using Matlab's ode45().
%%
%%        m*X'' + b*X' + k*X = F(t)       X(0) = X0,  X'(0) = Y0

t0 = 0.0;               %% use 0.0 to compare with exact solution
tf = 10.0;              %% final time
X0 = [0.10, 0.10];      %% inital point is a vector [X(t0), X'(t0)]
h  = 0.10;              %% step size for Euler method

clear  t1  t2  Xode  Xeul     %% clear the arrays

[t1, Xode] = ode45('vfPend', t0, tf, X0);     %% numerical solution
[t2, Xeul] = eul('vfPend', t0, tf, X0, h);    %% Euler method

%% plot the results X(t) and X'(t)
clf;                          %% clear the graphics window
plot(t1, Xode(:,1), 'y');     %% plot X(t) using yellow
hold on;                      %% hold onto previous plot
plot(t1, Xode(:,2), 'r--');  %% plot X'(t) using red
xlabel('t');
ylabel('X1, X2');
title('X1(t) and X2(t) using Matlab ode45()');
pause                         %% wait for user to strike a key
```

```
%% plot the solution in the phase plane; X'(t) vs X(t)
clf;
plot(Xode(:,1), Xode(:,2), 'y'); %% numerical solution
xlabel('X1');
ylabel('X2');
title('Solution in Phase Space')
pause

%% compare the solutions from ode45() and eul()
clf;
plot(t1, Xode(:,1), 'y');        %% ode45()
hold on;
plot(t2, Xeul(:,1), 'w--');      %% eul()
xlabel('t');
ylabel('X1');
title('Runge-Kutta (solid)  vs.  Euler (dashed)');

%%%%%%%%%%%%%%%%%%%%%%  End of pend45.m  %%%%%%%%%%%%%%%%%%%%%%
```

5 Some Sample M-files

In this section we provide Matlab code for some user defined functions. The first two M-files are the vector field definition and the exact solution for the scalar logistic growth model (see §2.6 in Boyce and DiPrima). The second pair of M-files are the vector field and exact solution for the periodically forced linear oscillator with damping (see §3.9 in Boyce and DiPrima).

5.1 Logistic Growth Model for a Single Species

```
%%%%%%%%%%%%%%%%%%%%%%  vfLog.m  %%%%%%%%%%%%%%%%%%%%%%
function [f] = vfLog(t, N)

%%  vector field for logistic equation (single species).
%%
%%         N'(t) = f(t,N) = (r - a*N)*N

r = 1.0;                %% intrinsic growth rate
a = 0.1;                %% intraspecific competition

f = (r - a*N)*N;

%%%%%%%%%%%%%%%%%%%%%%  End of vfLog.m  %%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%  solLog.m  %%%%%%%%%%%%%%%%%%%%%%
function [N] = solLog(t, N0)

%%  Solution to the logistic equation (single species).
```

```
%%
%%            N' = (r - a*N)*N        N(t0) = N0

r = 1.0;              %% intrinsic growth rate
a = 0.1;              %% intraspecific competition
K = r/a;

N = (N0*K) ./ (N0 + (K - N0)*exp(-r*t));
N = N(:);

%%%%%%%%%%%%%%%%%%%%  End of solLog.m  %%%%%%%%%%%%%%%%%%%%
```

5.2 Forced Linear Oscillator With Damping

```
%%%%%%%%%%%%%%%%%%%%  vfPend.m  %%%%%%%%%%%%%%%%%%%%
function [f] = vfPend(t, X)

%%  vector field for damped linear oscillator with external forcing.
%%
%%             m*X'' + b*X' + k*X = F(t)
%%
%%  Must transform to a system of two first order equations.
%%
%%             X' =  Y                          X(t0) = X0
%%             Y' = (-k*X - b*Y + F(t))/m       Y(t0) = Y0
%%
%%  Inputs:  t is a scalar,  X is a two element vector (row or column)

m  = 1.0;       %% mass
k  = 1.0;       %% spring constant (restoring force)
b  = 0.0;       %% damping coefficient
F0 = 0.0;       %% amplitude of external forcing
w  = 1.0;       %% angular frequency of periodic forcing (rads/sec)

F  = F0*cos(w*t);         %% external forcing

f(1) =  X(2);
f(2) = (-k*X(1) - b*X(2) + F) / m;

f = f(:);       %% return as a column vector

%%%%%%%%%%%%%%%%%%%%  End of vfPend.m  %%%%%%%%%%%%%%%%%%%%
```

6 Fixed-Step ODE Solvers

This section provides Matlab code for solving ODEs with the Euler method and the fourth order Runge-Kutta method. The functions are named **eul** and **rk4**, respectively. These are fixed-step integrators where the step size remains fixed throughout the integration. The two files are very similar with the only difference being the calculation of `slope` within the main integration loop. The functions use the same syntax as the Matlab functions **ode23** and **ode45** except that the user can specify the stepsize for **eul** and **rk4**. If the stepsize is omitted from the argument list the default step size is chosen by dividing the total time interval into one hundred equal steps. Note that these ODE routines are designed to solve systems of first order equations as well as scalar first order equations. Included here are comments describing the required input and output arguments for the functions **eul** and **rk4**.

```
%% EUL  Numerical integration of ODEs using Euler method.
%% RK4  Numerical integration of ODEs using fourth order Runge-Kutta.
%%
%%      EUL('xprime', t0, tf, x0, h)
%%      RK4('xprime', t0, tf, x0, h)
%%
%% Inputs:
%%      xprime   a string variable with the name of the M-file which
%%               defines the vector field on the right hand side.
%%      t0       initial time for the integration
%%      tf       final time for the integration
%%      x0       a vector (column or row) containing initial conditions
%%      h        stepsize to be used in the integration
%%
%% Outputs:
%%      tout     column vector containing integration times
%%      xout     numerical solution at times stored in tout.
%%               Each row of xout is a vector of length N, where
%%               N is the dimension of the phase space.
%%
%%       Input format is the same as ODE23() and ODE45() except for the
%%       additional parameter specifying the stepsize.  Note that EUL()
%%       and RK4() work for systems of first order equations as well as
%%       scalar equations.
```

6.1 Euler Method

```
%%%%%%%%%%%%%%%%%%%%  eul.m  %%%%%%%%%%%%%%%%%%%%
function [tout, xout] = eul(vf, t0, tf, x0, h)

%% EUL  Numerical integration of ODEs using Euler method.
%%
%%      EUL('xprime', t0, tf, x0, h)

if (nargin < 5)               %% set default stepsize
  h = (tf - t0)/100.;
end

h = abs(h);
if tf < t0                    %% h < 0 when integrating backwards
  h = -h;
end

t = t0;
tout = t0;
x = x0(:);                    %% force x to be a column vector
xout = x.';                   %% transpose into row vector

while t ~= tf
  if abs(tf - t) < abs(h)
    tnext = tf;               %% adjust the stepsize at end of
    h = tf - t;               %% interval to land exactly on tf.
  else
    tnext = t + h;
  end;

  slope = feval(vf, t, x);
  slope = slope(:);           %% must be a column vector

  x = x + h*slope;            %% the next iterate
  t = tnext;                  %% advance the independent variable

  xout = [xout; x.'];         %% append solution matrix
  tout = [tout; t];           %% append time vector
end;

%%%%%%%%%%%%%%%%%%%%  End of eul.m  %%%%%%%%%%%%%%%%%%%%
```

6.2 Fourth Order Runge-Kutta Method

```
%%%%%%%%%%%%%%%%%%%%  rk4.m  %%%%%%%%%%%%%%%%%%%%
function [tout, xout] = rk4(vf, t0, tf, x0, h)

%% RK4  Numerical integration of ODEs using fourth order Runge-Kutta.
%%
%%      RK4('xprime', t0, tf, x0, h)

if (nargin < 5)               %% set default stepsize
  h = (tf - t0)/100.;
end

h = abs(h);
if tf < t0                    %% h < 0 when integrating backwards
  h = -h;
end

t = t0;
tout = t0;
x = x0(:);                    %% force x to be a column vector
xout = x.';                   %% transpose into row vector

while t ~= tf
  if abs(tf - t) < abs(h)
    tnext = tf;               %% adjust the stepsize at end of
    h = tf - t;               %% interval to land exactly on tf.
  else
    tnext = t + h;
  end;

  s1 = feval(vf, t, x);                         s1 = s1(:);
  s2 = feval(vf, t + h/2.0, x + s1*h/2.0);     s2 = s2(:);
  s3 = feval(vf, t + h/2.0, x + s2*h/2.0);     s3 = s3(:);
  s4 = feval(vf, t + h, x + s3*h);             s4 = s4(:);

  slope = (s1 + 2.0*s2 + s3*2.0 + s4)/6.0;     %% "average" slope

  x = x + h*slope;            %% the next iterate
  t = tnext;                  %% advance the independent variable

  xout = [xout; x.'];         %% append solution matrix
  tout = [tout; t];           %% append time vector
end;

%%%%%%%%%%%%%%%%%%%%  End of rk4.m  %%%%%%%%%%%%%%%%%%%%
```