Chapter 7. Solution of Ordinary Differential Equations

7.1. Introduction

The dynamic behavior of many relevant systems and materials can be described with ordinary differential equations (ODEs). In this chapter, we provide an introduction to the techniques for numerical solution of ODEs. We begin with a single, first-order ODE initial value problem. We then extend the process to high-order ODEs, systems of ODEs and boundary value problems. The techniques described in this chapter work for both linear and nonlinear ODEs. However, we point out that for linear ODEs and some non-linear ODEs there may exist more elegant analytical solutions. In this book, we do not investigate the analytical solutions but limit ourselves to numerical methods that provide valid solutions.

7.2. Initial Value Problems

Regardless of whether one intends to solve an ordinary differential equation (ODE) with analytical or numerical techniques, the problem must first be properly posed. Let us initially limit ourselves to a single, first-order ODE, involving a single independent variable x and a single dependent variable $y(x)$. Differential equations that involve more than one independent variable are called partial differential equations (PDEs) and are not considered in this book. Differential equations that involve more than one dependent variable constitute systems of ODEs and addressed later in this chapter. The solution to an ODE is a function, $y(x)$. The most general formulation of this ODE is

$$f\left(x, y(x), y'(x)\right) = 0 \tag{7.1}$$

where we have invoked the shorthand notation, $y'(x) \equiv dy/dx$, for the derivative of y with respect to x. Often, it is possible through algebraic manipulation of equation (7.1) to isolate the derivative on the LHS,

$$y'(x) = f\left(x, y(x)\right) \tag{7.2}$$

This is perhaps the most familiar form of an ODE, but it is not the most general.

ODEs are frequently categorized as linear and nonlinear ODEs. It is important to remember that the linearity of the ODE is defined by the linearity of *y* only (and specifically not of *x*). The unknown *y* must be operated on exclusively by linear operators in order for the ODE to be considered linear. (Recall that the differential operator is a linear operator.) The general form of a linear first order ODE is

$$a(x)y'(x) + b(x)y(x) + c(x) = 0 \tag{7.3}$$

The coefficients, $a(x)$, $b(x)$ and $c(x)$ can be nonlinear functions of *x*.

Regardless of whether one is solving an ODE of the form given in (7.1), (7.2) or (7.3), the problem is not properly posed and does not have a unique solution until an initial condition is supplied. The initial condition provides a value of the function at one point.

$$y(x = x_0) = y_0 \tag{7.4}$$

The initial value problem (IVP) is the combination of the ODE and the initial condition.

7.3. Euler Method

It is instructive to begin our study of numerical techniques for solving ODEs with a first order method. In practice, we will never use a first-order method to solve an ODE, due to its low accuracy. Nevertheless, the first-order method plays an important role as an instructional tool.

The Euler method relies on a Taylor series truncated after the linear term.

$$f(x_{i+1}) = f(x_i) + \left.\frac{df}{dx}\right|_{x_i} h + O(h^2) \tag{3.2}$$

which can be rewritten in the nomenclature of ODEs as

$$y(x_{i+1}) = y(x_i) + \Delta x y'(x_i) \tag{7.5}$$

where the discretization is given by $\Delta x = x_{i+1} - x_i$. Equation (7.5) is the Euler method. One begins at the initial condition $(x, y) = (x_0, y_0)$ as given in the problem statement. One choose a discretization, Δx, which is a crucial decision. One then evaluates the derivative at x_0 using the ODE (equation (7.1), (7.2) or (7.3)), which also was given in the problem statement. The Euler method in equation (7.5) can then be used directly to estimate the value of the unknown function at

$x_1 = x_0 + \Delta x$. This process can be repeated moving further down the x-axis for as long as interest remains.

Before we introduce specific examples, two points of discussion are necessary. First, the choice of discretization or step-size is crucial. It is true that the smaller the step-size the more accurate the solution. However, we must balance the desire for accuracy with the need for computational efficiency. If our choice of step-size is so small that we need an intractable number of steps, then that doesn't help us. If the step size is too big, the method will fail. In fact, for the simple methods presented in this book, **the only reason a numerical routine for solving an ODE will fail is because the step-size was too large**. The reason can be seen in Figure 7.1. In the top figure, it is clear that as the step-size increases, the error increases. The error is quantified in the middle figure. For some functions it is possible that the error leads to values of the function that are not permitted, such as negative values of $y(x)$ in functions that require a square root.

The second point of interest is the determination of when to stop the Euler iterations. The final value of x is simply chosen by the user. Frequently, the stopping point is determined when one has reached a steady state. For example in the bottom part of Figure 7.1, the function clearly goes to zero. Continuing to solve the ODE out to large values of x typically serves no useful purpose. If one doesn't know how long it will take to reach a steady state, then one may have to guess the final value. If it is too short, then one can simply extend the Euler procedure using the final point of the previous process as the initial condition for the

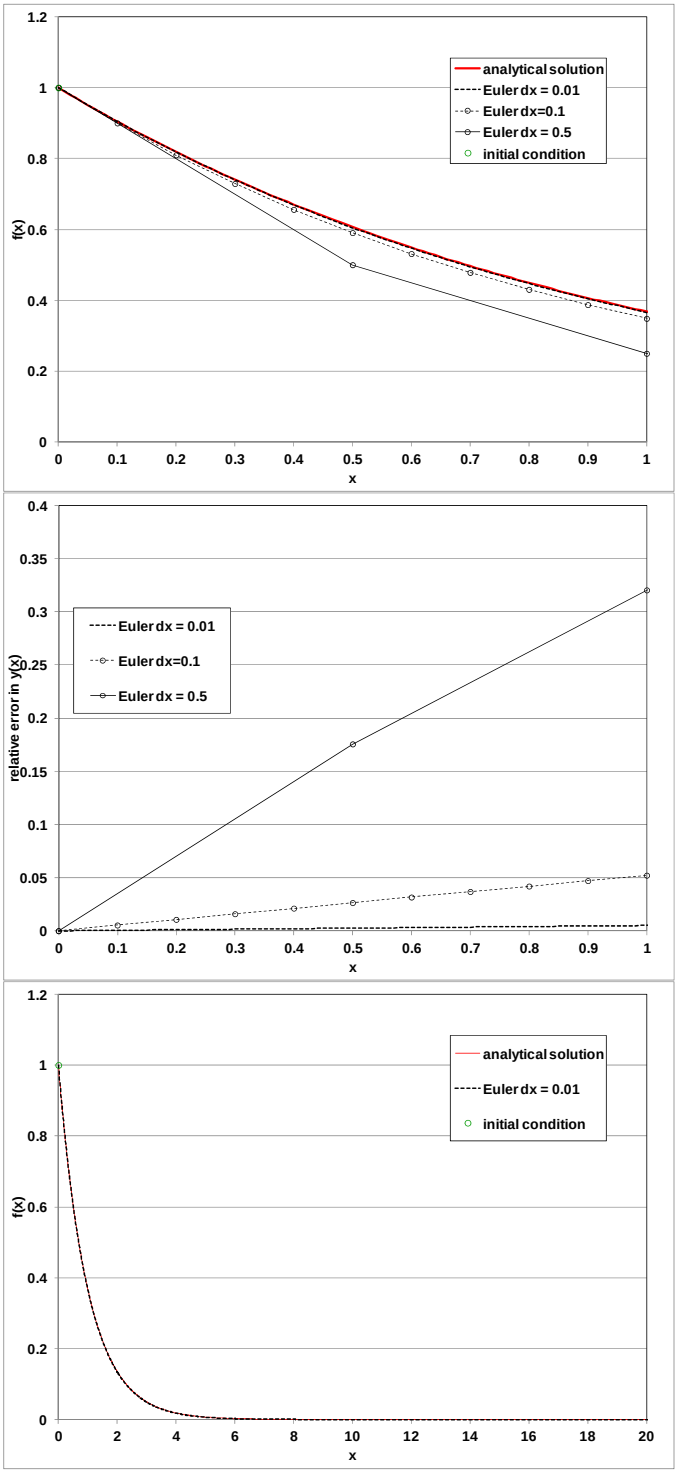

Figure 7.1 Application of Euler method to $y'(x)=-y$ with initial condition $y(x=0)=1$. Top: solution up to x=1. Middle: error up to x=1. Bottom: solution up to x=20.

continuation.

A MATLAB code which implements the Euler method for a single first-order ODE is provided later in this chapter.

7.4. Classical Fourth-Order Runge-Kutta Method

The obvious solution to inaccuracy of the Euler method is to increase the order of the method. There are numerous flavors of ODE solvers of all orders. Here we simply present the Classical Fourth-Order Runge-Kutta (RK4) Method, which is a tried and true solution technique. The RK4 method proceeds as does the Euler method by starting at the initial condition and following an estimate of the average slope over the interval, $\langle y' \rangle$, out to the next discretization point.

$$y(x_{i+1}) = y(x_i) + \Delta x \langle y' \rangle \tag{7.6}$$

In the Euler method, the estimate of the average slope was $\langle y' \rangle = y'(x_i)$, simply the value of the slope at the beginning of the interval. The RK4 method uses a higher order approximation for the average slope over the interval, namely

$$\langle y' \rangle = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{7.7}$$

where

$$\begin{aligned} k_1 &= y'(x_i, y(x_i)) \\ k_2 &= y'\left(x_i + \frac{\Delta x}{2}, y(x_i) + \frac{\Delta x}{2} k_1 \right) \\ k_3 &= y'\left(x_i + \frac{\Delta x}{2}, y(x_i) + \frac{\Delta x}{2} k_2 \right) \\ k_4 &= y'(x_i + \Delta x, y(x_i) + \Delta x k_3) \end{aligned} \tag{7.8}$$

Without a rigorous derivation, we can see that the Runge-Kutta method improves the quality of the estimate of the slope over the interval by evaluating the slope at the beginning, middle (twice) and end of the interval. The RK4 method is a fourth-order method, so the error decreases as the step-size to the fourth power.

In Figure 7.2, we present an application of the RK4 method. In the left figure we show that even for coarse discretization, the RK4 method can be very accurate. In the right figure we quantify the relative error. Note that the error is now shown on a log axis

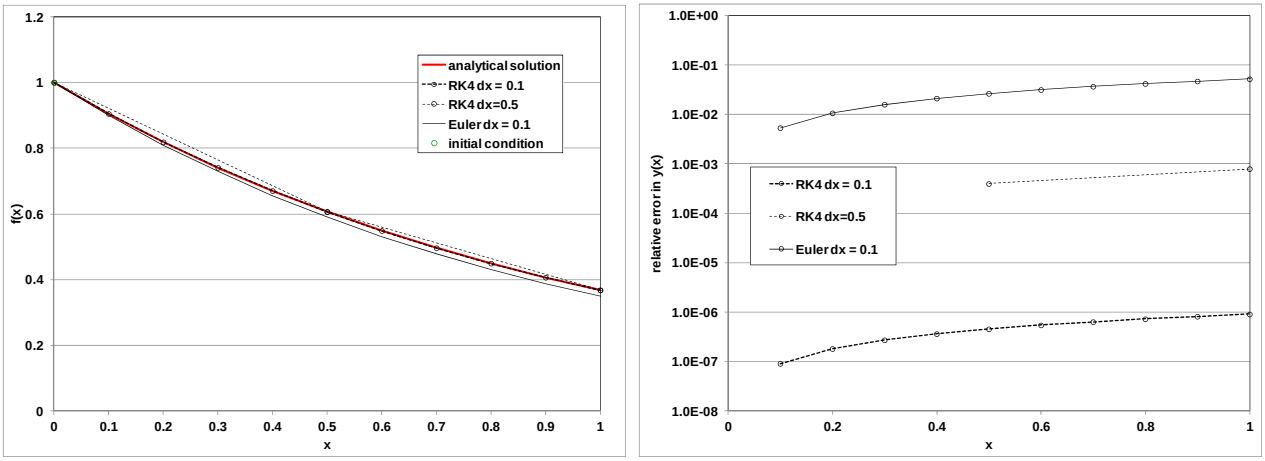

Figure 7.2. Application of RK4 method to $y'(x)=-y$ with initial condition $y(x=0)=1$. Left: solution up to x=1. Right: error up to x=1.

A MATLAB code which implements the classical fourth-order Runge-Kutta method for a single first-order ODE is provided later in this chapter.

7.5. Application to Systems of Ordinary Differential Equations

The extension of the Euler or Runge-Kutta method to systems of ODEs is very straightforward for an initial value problem. Let's suppose that we have *n* ODEs, each of which can be written in the form of equation (7.2) as

$$\begin{aligned}
\frac{dy_1(x)}{dx} &= f_1(x, y_1, y_2, y_3, \ldots y_{n-1}, y_n) \\
\frac{dy_2(x)}{dx} &= f_2(x, y_1, y_2, y_3, \ldots y_{n-1}, y_n) \\
&\ldots \\
\frac{dy_n(x)}{dx} &= f_n(x, y_1, y_2, y_3, \ldots y_{n-1}, y_n)
\end{aligned} \tag{7.9}$$

with *n* initial conditions:

$$y_j(x = x_0) = y_{j,0} \quad \text{for j} = 1 \text{ to n} \tag{7.10}$$

The fact that all of the conditions are given at the same value of *x* is what makes this problem an IVP. Note carefully, that there are *n* ODE's, thus *n* functions, f_j, and *n* unknown functions, y_j,

but there is only one independent variable *x*. This is what makes this system a system of ODEs rather than PDEs. The techniques in this chapter will not solve PDEs.

The Euler method for a system of of ODEs can be written as

$$y_j(x_{i+1}) = y_j(x_i) + \Delta x * f_j\left(x_i, y_1(x_i), y_2(x_i), y_3(x_i), \ldots y_{n-1}(x_i), y_n(x_i)\right) \quad \text{for } j = 1 \text{ to } n \quad (7.11)$$

There is no difference between this equation and the equation for the single system ODE using Euler's method (equation (7.5)). In this case, remember that the subscript *j* attached to the *y* and the *f* denotes different functions. The subscript *i* attached to the *x* variable denotes steps (or iterations). The Euler algorithm for a system of ODEs simply requires one to evaluate all f_j at iteration *i* from the ODEs in the problem statement (equation 7.9), then compute all y_j at iteration *i*+1 from equation (7.11) and repeat the process.

The extension of RK4 to systems of ODE's is just as simple as the extension of Euler's method. However, because RK4 has a little more sophistication, the extension looks more complicated, when it is really not. The RK4 equation for a system of equations is given by

$$y_j(x_{i+1}) = y_j(x_i) + \left[\frac{1}{6}\left(k_{1,j} + 2k_{2,j} + 2k_{3,j} + k_{4,j}\right)\right]h \qquad \text{for } j = 1 \text{ to } n \qquad (7.12)$$

where

$$\begin{aligned}
k_{1,j} &= f_j\left(x_i, \{y_m(x_i)\}\right) \\
k_{2,j} &= f_j\left(x_i + \frac{\Delta x}{2}, \left\{y_m(x_i) + \frac{\Delta x}{2}k_{1,m}\right\}\right) \\
k_{3,j} &= f_j\left(x_i + \frac{\Delta x}{2}, \left\{y_m(x_i) + \frac{\Delta x}{2}k_{2,m}\right\}\right) \\
k_{4,j} &= f_j\left(x_i + \Delta x, \{y_m(x_i) + \Delta x k_{3,m}\}\right)
\end{aligned} \qquad \text{for } j = 1 \text{ to } n \qquad (7.13)$$

In equation (7.13), the braces in $\{y_m(x_i)\}$ represent the set of all y_m from *m* = 1 to *n*., all of which are evaluated at x_i.

The multi-ODE RK4 algorithm requires that one first evaluate all $k_{1,j}$. With $k_{1,j}$ in hand, one can compute the arguments required for $k_{2,j}$. Once all of the $k_{2,j}$ are known, one can next compute the arguments required for $k_{3,j}$. Once all of the $k_{3,j}$ are known, one can next compute the arguments required for $k_{4,j}$. Once all the $k_{1,j}$, $k_{2,j}$, $k_{3,j}$ and $k_{4,j}$ are known, one can use the RK4 method (equation (7.12) to evaluate the value of the unknown functions at the next step, $y_j(x_{i+1})$.

MATLAB codes which implement the Euler method and the classical fourth-order Runge-Kutta method for a system of first-order ODEs are provided later in this chapter.

7.6. Higher-Order ODEs

Regardless of whether one is pursuing an analytical or numerical solution to a higher than first order ODE, there is a simple trick to converting an n^{th}-order ODE to a system of n first-order ODEs. You make a substitution that transforms the nth-order differential equation into n first-order differential equations. Consider an n^{th}-order ODE of the form

$$\frac{d^n y(x)}{dx^n} = f\left(x, y(x), \frac{dy(x)}{dx}, \frac{d^2 y(x)}{dx^2} \cdots \frac{d^{n-1} y(x)}{dx^{n-1}}\right) \tag{7.14}$$

with the following *n* initial conditions:

$$y(x = x_0) = y_0, \left.\frac{dy(x)}{dx}\right|_{x=x_0} = y_0', \left.\frac{d^2 y(x)}{dx^2}\right|_{x=x_0} = y_0'', \ldots \left.\frac{d^{n-1} y(x)}{dx^{n-1}}\right|_{x=x_0} = y_0^{(n-1)} \tag{7.15}$$

The conversion is a three step process. The first step is defining *n* new variables, which always have the following form:

$$\begin{aligned} y_1(x) &= y(x) \\ y_2(x) &= \frac{dy(x)}{dx} \\ y_3(x) &= \frac{d^2 y(x)}{dx^2} \\ &\vdots \\ y_n(x) &= \frac{d^{n-1} y(x)}{dx^{n-1}} \end{aligned} \tag{7.16}$$

The second step is writing one first-order ODE for each of these *n* variables. The first n-1 ODEs have a trivial form. The last ODE is simply the original higher-order ODE, equation (7.14), with the new variables, equation (7.16) substituted in.

$$\begin{aligned}
&\frac{dy_1(x)}{dx} = y_2(x)\\
&\frac{dy_2(x)}{dx} = y_3(x)\\
&\vdots\\
&\frac{dy_{n-1}(x)}{dx} = y_n(x)\\
&\frac{dy_n(x)}{dx} = f\left(x, y_1(x), y_2(x), y_3(x)\ldots y_{n-1}(x)\right)
\end{aligned} \tag{7.16}$$

The third and final step of the conversion process is to rewrite the initial conditions, equation (7.15), in terms of the new variables,

$$y_1(x_0) = y_0,\ y_2(x_0) = y_0',\ y_3(x_0) = y_0'',\ldots y_n(x_0) = y_0^{(n-1)} \tag{7.17}$$

After the conversion equations (7.16) and (7.17) have constitute a system of *n* first-order ODEs with *n* initial conditions. This IVP can be solved using the methods of the previous section.

7.7. Boundary Value Problems

When one invokes Fick's law of diffusion in a material balance, or Fourier's law of heat conduction in an energy balance, or Newton's law of viscosity in a momentum balance, one can end up with a steady-state model that is a second order ODE, where the independent variable is position. What distinguishes these problems from other higher-order ODEs is that the conditions are not given at the same point. From a mathematical point of view, we can write this equation as

$$\frac{d^2y(x)}{dx^2} = f\left(x, y(x), \frac{dy(x)}{dx}\right) \tag{7.18}$$

with the following 2 boundary conditions:

$$y(x_0) = y_0 \qquad \text{and} \qquad y(x_f) = y_f \tag{7.19}$$

Instead of being given the value of the function and its derivative at a single point, x_0, we now have a constraint on the function at two points, x_0 and x_f. For example, we know the temperature at either side of a metal rod,where our thermocouples are attached, but we know nothing about the

derivative of the temperature. The boundary conditions in equation (7.19) define what is called the Boundary Value Problem (BVP).

As presented, the Euler or RK4 only work for IVPs, not BVPs. The solution lies in creating a numerical method that combines our ability to numerically solve nonlinear algebraic equations with ODEs. If we had been given an initial value of the first derivative, $\left.\frac{dy(x)}{dx}\right|_{x=x_0} = y_0'$, we would have an IVP and we could solve the problem easily, using the techniques in Section 7.6 (breaking up the second-order ODE into two first-order ODEs) and Section 7.5 (solving a system of first-order ODEs in an IVP). Therefore, we play to our strengths. We will guess a value of y_0'. We will solve the system of ODEs from x_0 to x_f. We will compare the calculated value of $y(x_f)$ with the boundary condition, y_f. If they match within a tolerance, we made a good guess. Probably, they don't match and we must make a new guess for y_0' and iterate again. What should guide our guess for y_0'? Why, I have a just the thing! We should use the Newton-Raphson method with numerical derivatives for solving a single nonlinear algebraic equation. The unknown is y_0'. The algebraic equation is

$$g(y_0') = y(x_f) - y_f = 0 \tag{7.20}$$

In other words, we must choose the correct initial condition on the derivative to satisfy the given boundary condition on the function. We understand that since $y(x_f)$ is arrived at via the solution of the set of ODEs, that changing the value of y_0' will change the value of $y(x_f)$. Every evaluation of the function given in equation (7.20) requires a solution of the system of ODEs.

A MATLAB code which implements the solution of a BVP for a system of 2 ODEs using the Newton-Raphson method and the classical fourth-order Runge-Kutta method is provided later in this chapter.

7.8. Subroutine Codes

In this section, we provide routines for implementing the various numerical ODE solver methods described above. Note that these codes correspond to the theory and notation exactly as laid out in this book. These codes do not contain extensive error checking, which would complicate the coding and defeat their purpose as learning tools. That said, these codes work and can be used to solve problems.

As before, on the course website, two entirely equivalent versions of this code are provided and are titled *code.m* and *code_short.m*. The short version is presented here. The longer version, containing instructions and serving more as a learning tool, is not presented here. The numerical mechanics of the two versions of the code are identical.

Code 7.1. Euler Method – 1 ODE (euler1_short)

```
function [x,y]=euler1_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
y = zeros(n+1,1);
y(1) = yo;
for i =  1:1:n
   dydx = funkeval(x(i),y(i));
      y(i+1) = y(i) + dx*dydx;
end
close all;
iplot = 1;
if (iplot == 1)
   plot (x,y,'k-o'), xlabel( 'x' ), ylabel ( 'y' );
end
fid = fopen('euler1_out.txt','w');
fprintf(fid,'x                y \n');
fprintf(fid,'%23.15e %23.15e   \n', [x,y]');
fclose(fid);

function dydx = funkeval(x,y);
dydx = -1.0*y;
```

An example of using euler1_short is given below.

```
» [x,y]=euler1_short(10,0,2,1);
```

This program generates outputs in three forms. First, the x and y vectors are stored in memory and can be directly accessed. Second, the program generates a plot of y vs. x. Third, the program generates an output file, *euler1_out.txt*, that contains x and y vectors in tabulated form.

Code 7.2. Fourth-Order Runge-Kutta Method – 1 ODE (rk41_short)

```
function [x,y]=rk41_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
y = zeros(n+1,1);
y(1) = yo;
for i =  1:1:n
   x1 = x(i);
   y1 = y(i);
   k1 = funkeval(x1,y1);
   x2 = x(i) + 0.5*dx;
```

```
   y2 = y(i) + 0.5*dx*k1;
   k2 = funkeval(x2,y2);
   x3 = x(i) + 0.5*dx;
   y3 = y(i) + 0.5*dx*k2;
   k3 = funkeval(x3,y3);
   x4 = x(i) + dx;
   y4 = y(i) + dx*k3;
   k4 = funkeval(x4,y4);
   dydx = 1.0/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4);
   y(i+1) = y(i) + dx*dydx;
end
close all;
iplot = 1;
if (iplot == 1)
   plot (x,y,'k-o'), xlabel( 'x' ), ylabel ( 'y' );
end
fid = fopen('rk41_out.txt','w');
fprintf(fid,'x                 y \n');
fprintf(fid,'%23.15e %23.15e   \n', [x,y]');
fclose(fid);

function dydx = funkeval(x,y);
dydx = -1.0*y
```

An example of using rk41_short is given below.

```
» [x,y]=rk41_short(10,0,2,1);
```

This command generates outputs in three forms. First, the x and y vectors are stored in memory and can be directly accessed. Second, the program generates a plot of y vs. x. Third, the program generates an output file, *rk41_out.txt*, that contains x and y vectors in tabulated form.

Code 7.3. Euler Method – n ODEs (eulern_short)

```
function [x,y]=eulern(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
m=max(size(yo));
y = zeros(n+1,m);
y(1,1:m) = yo(1:m);
dydx = zeros(m,1);
for i =  1:1:n
   dydx = funkeval(x(i),y(i,1:m));
      y(i+1,1:m) = y(i,1:m) + dx*dydx(1:m);
end
close all;
iplot = 1;
if (iplot == 1)
```

```
   for i = 1:1:m
      color_index = get_plot_color(i);
      plot (x(:),y(:,i),color_index);
      hold on;
   end
   hold off;
   xlabel( 'x' );
   ylabel ( 'y' );
   legend (int2str([1:m]'));
end
fid = fopen('eulern_out.txt','w');
fprintf(fid,'x  y(1) ... y(m) \n');
for i = 1:1:n+1
   fprintf(fid,'%23.15e ', x(i));
   for j = 1:1:m
      fprintf(fid,'%23.15e ', y(i,j));
   end
   fprintf(fid,' \n');
end
fclose(fid);

function dydx = funkeval(x,y);
dydx(1) = -1.0*y(1) - 2.0*y(2) - 0.5*y(3);
dydx(2) = -0.1*y(1) - 4.0*y(2) - 0.5*y(3);
dydx(3) = -0.5*y(1) - 0.4*y(2) - 0.2*y(3);
```

An example of using eulern_short is given below.

```
» [x,y]=eulern(100,0,10,[1,0,2]);
```

Note that the fourth input argument, yo, is now a vector. This program generates outputs in three forms. First, the x vector and y matrix are stored in memory and can be directly accessed. Second, the program generates a plot of y vs. x. Third, the program generates an output file, *eulern_out.txt*, that contains x and y in tabulated form.

Also note that this code calls an ancillary function for plotting, *get_plot_color*, which is reproduced below. The function assigns a different color to each function so that the curves are distinguishable on the graph. This function can be included at the bottom of the file eulern_short.m.

```
function color_index = get_plot_color(i);
if (i == 1)
   color_index = 'k-';
elseif (i == 2)
   color_index = 'r-';
elseif (i == 3)
   color_index = 'b-';
elseif (i == 4)
   color_index = 'g-';
elseif (i == 5)
```

```
   color_index = 'm-';
elseif (i == 6)
   color_index = 'k:';
elseif (i == 7)
   color_index = 'r:';
elseif (i == 8)
   color_index = 'b:';
elseif (i == 9)
   color_index = 'g:';
elseif (i == 10)
   color_index = 'm:';
else
   color_index = 'k-';
end
```

Code 7.4. Classical Fourth-Order Runge-Kutta Method – n ODEs (rk4n_short)

```
function [x,y]=rk4n_short(n,xo,xf,yo);
dx = (xf-xo)/n;
x = zeros(n+1,1);
for i = 1:1:n+1
   x(i) = xo + (i-1)*dx;
end
m=max(size(yo));
y = zeros(n+1,m);
y(1,1:m) = yo(1:m);
dydx = zeros(1,m);
ytemp = zeros(1,m);
k1 = zeros(1,m);
k2 = zeros(1,m);
k3 = zeros(1,m);
k4 = zeros(1,m);
for i =  1:1:n
   x1 = x(i);
   ytemp(1:m) = y(i,1:m);
   k1(1:m) = funkeval(x1,ytemp);
   x2 = x(i) + 0.5*dx;
   ytemp(1:m) = y(i,1:m) + 0.5*dx*k1(1:m);
   k2(1:m) = funkeval(x2,ytemp);
   x3 = x(i) + 0.5*dx;
   ytemp(1:m) = y(i,1:m) + 0.5*dx*k2(1:m);
   k3(1:m) = funkeval(x3,ytemp);
   x4 = x(i) + dx;
   ytemp(1:m) = y(i,1:m) + dx*k3(1:m);
   k4(1:m) = funkeval(x4,ytemp);
   dydx(1:m) = 1.0/6.0*(k1(1:m) + 2.0*k2(1:m) + 2.0*k3(1:m) + k4(1:m));
   y(i+1,1:m) = y(i,1:m) + dx*dydx(1:m);
end
close all;
iplot = 1;
if (iplot == 1)
   for i = 1:1:m
```

```
      color_index = get_plot_color(i);
      plot (x(:),y(:,i),color_index);
      hold on;
   end
   hold off;
   xlabel( 'x' );
   ylabel ( 'y' );
   legend (int2str([1:m]'));
end
fid = fopen('rk4n_out.txt','w');
fprintf(fid,'x  y(1) ... y(m) \n');
for i = 1:1:n+1
   fprintf(fid,'%23.15e ', x(i));
   for j = 1:1:m
      fprintf(fid,'%23.15e ', y(i,j));
   end
   fprintf(fid,' \n');
end
fclose(fid);

function dydx = funkeval(x,y);
dydx(1) = -1.0*y(1) - 2.0*y(2) - 0.5*y(3);
dydx(2) = -0.1*y(1) - 4.0*y(2) - 0.5*y(3);
dydx(3) = -0.5*y(1) - 0.4*y(2) - 0.2*y(3);
```

An example of using rk4n_short is given below.

```
» [x,y]=rk4n(100,0,10,[1,0,2]);
```

Note that the fourth input argument, yo, is now a vector. This program generates outputs in three forms. First, the x vector and y matrix are stored in memory and can be directly accessed. Second, the program generates a plot of y vs. x. Third, the program generates an output file, *eulern_out.txt*, that contains x and y in tabulated form.

Also note that this code calls an ancillary function for plotting, *get_plot_color*, which is reproduced above. The function assigns a different color to each function so that the curves are distinguishable on the graph. This function can be included at the bottom of the file rk4n_short.m.

Code 7.5. Newton-Raphson/Runge-Kutta Method – 2 ODEs BVP (nrnd1 & rk4n)

No new codes were used to solve the BVP problem. Instead the input files for the Newton-Raphson with numerical derivatives (nrnd1.m) and the classical fourth-order Runge-Kutta method for a system of ODEs (rk4n.m) were modified.

In the file rk4n.m, we entered the ODEs

```
function dydx = funkeval(x,y);
dydx(1) = y(2);
dydx(2) = -1.0*y(1) - 0.5*y(2);
```

In the file nrnd1.m, the input function was specified as

```
function f = funkeval(x)
xo = 0;
yo_1 = 1;
yo_2 = x;
xf = 2.0;
yf = 1.0;
n = 100;
[x,y]=rk4n(n,xo,xf,[yo_1,yo_2]);
yf_calc = y(n+1,1);
f = yf_calc-yf;
```

In this input file, we call the Runge-Kutta routine. We provide the given boundary conditions. We allow the initial condition for the second function to vary with each iteration, since it is the unknown in the Newton-Raphson procedure.

An example of using solving this BVP is given below.

```
» [x0,err] = nrnd1(0.1)
icount = 1 xold = 1.000000e-001 f = -1.012145e+000 df = 5.850002e-001 xnew =
1.830161e+000  err = 1.000000e+002

icount = 2 xold = 1.830161e+000 f = -2.167155e-013 df = 5.850002e-001 xnew =
1.830161e+000  err = 2.023704e-013

x0 =    1.8302
err =  2.0237e-013
```

The problem was solved in one iteration because the ODEs were linear. It took a second iteration to identify convergence. The initial value of the derivative required to solve the boundary condition is `1.8302`. The rk4n.m program generates a plot of y vs. x and generates an output file, *rk4n_out.txt*, that contains x and y in tabulated form. However, be warned that this plot and table of data correspond to the last call of the Runge-Kutta routine by the Newton-Raphson code, which would have been to evaluate the numerical derivative. Therefore, this plot does not correspond exactly to the solution. To generate the solution, run the Runge-Kutta code with the appropriate initial condition.

```
» [x,y]=rk4n(100,0,2,[1,1.8302]);
```

The resulting plot is shown in Figure 7.3. It demonstrates that the final value of *y* is indeed the specified value of 1.

7.9. Problems

Problems are located on course website.

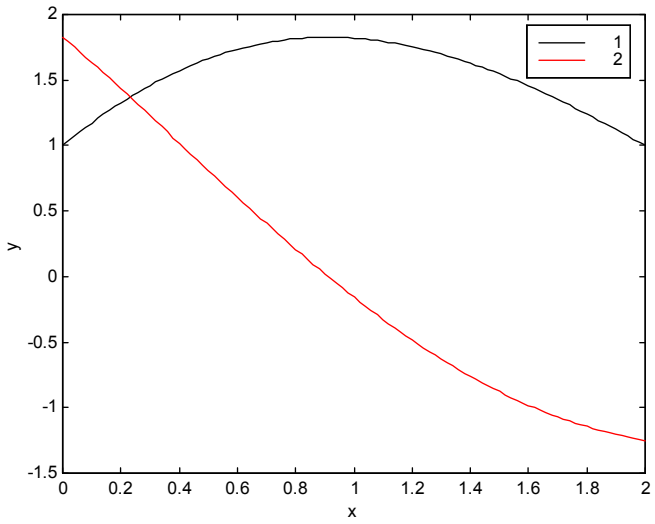

Figure 7.3. Solution to boundary value problem. $\frac{d^2y(x)}{dx^2} = -y(x) - \frac{1}{2}\frac{dy(x)}{dx}$ subject to the boundary conditions $y(0) = 1$ and $y(2) = 1$. The two curves correspond to y (black) and the first derivative (red).