CHAPTER

SOLUTION OF DIFFERENTIAL EQUATIONS 5

Abstract

Many practical problems involve the study of how rates of change in two or more variables are interrelated. Often the independent variable is time. These problems give rise naturally to differential equations which enable us to understand how the real-world works and how it changes dynamically. In this chapter, we describe a range of widely used algorithms to solve differential equations.

5.1 INTRODUCTION

To illustrate how a differential equation can model a physical situation we will examine a relatively simple problem. Consider the way a hot object cools, – for example, a saucepan of milk, the water in a bath or molten iron. Each of these will cool in a different way dependent on the environment but we shall abstract only the most important features that are easy to model. To model this process by a simple differential equation we use Newton's law of cooling which states that the *rate* at which these objects lose heat as time passes is dependent on the difference between the current temperature of the object and the temperature of its surroundings. This leads to the differential equation

$$dy/dt = K(y - s) \tag{5.1}$$

where y is the current temperature at time t, s is the temperature of the surroundings, and K is a negative constant for the cooling process. In addition we require the initial temperature, y_0, to be specified at time $t = 0$ when the observations begin. This fully specifies our model of the cooling process. We only need values for y_0, K, and s to begin our study. This type of first-order differential equation is called an *initial value problem* because we have an initial value given for the dependent variable y at time $t = 0$.

The solution of (5.1) is easily obtained analytically and will be a function of t and the constants of the problem. However, there are many differential equations which have no analytic solution or the analytic solution does not provide an explicit relation between y and t. In this situation we use numerical methods to solve the differential equation. This means that we approximate the continuous solution with an approximate discrete solution giving the values of y at specified time steps between the initial value of time and some final time value. Thus we compute values of y, which we denote by y_i, for values of t denoted by t_i where $t_i = t_0 + ih$ for $i = 0, 1, ..., n$. Fig. 5.1 illustrates the exact solution and an approximate solution of (5.1) where $K = -0.1$, $s = 10$, and $y_0 = 100$. This figure is generated using the standard MATLAB function for solving differential equations, `ode23`, from time 0 to 60 and plotting the values of y using the symbol "+". The values of the exact solution are plotted on the same graph using the symbol "o".

Numerical Methods. https://doi.org/10.1016/B978-0-12-812256-3.00014-2

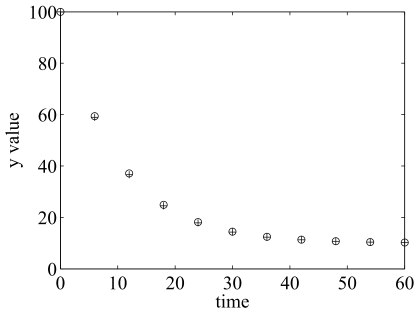

FIGURE 5.1

Exact o and approximate + solution for $dy/dt = -0.1(y-10)$.

To use `ode23` to solve (5.1) we begin by writing a function `yprime` which defines the right-hand side of (5.1). Then `ode23` is called in the following script and requires the initial and final values of t, 0, and 60 which must be placed in a row vector; a starting value for y of 100; and a low relative tolerance of 0.5. This tolerance is set using the `odeset` function which allows tolerances and other parameters to be set as required.

```
% e4s501.m
yprime = @(t,y) -0.1*(y-10); %RH of diff equn.
options = odeset('RelTol',0.5);
[t y] = ode23(yprime,[0 60],100,options);
plot(t,y,'+')
xlabel('Time'), ylabel('y value'),
hold on
plot(t,90*exp(-0.1.*t)+10,'o'), % Exact solution.
hold off
```

This type of step by step solution is based on computing the current y_i value from a single or combination of functions of previous y values. If the value of y is calculated from a combination of more than one previous value, it is called a *multi-step* method. If only one previous value is used it is called a *single-step* method. We shall now describe a simple *single-step* method known as Euler's method.

5.2 EULER'S METHOD

The dependent variable y and the independent variable t, which we used in the preceding section, can be replaced by any variable names. For example, many textbooks use y as the dependent variable and x as the independent variable. However, for some consistency with MATLAB notation we generally use y to represent the dependent variable and t to represent the independent variable. Clearly initial value problems are not restricted to the time domain although, in most practical situations, they are.

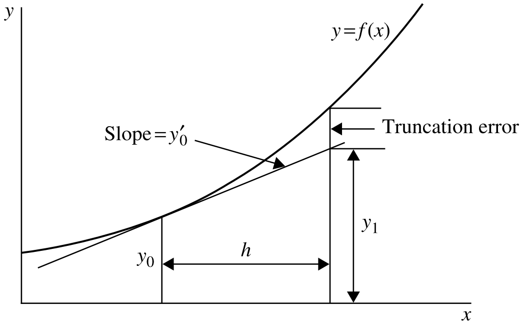

FIGURE 5.2

Geometric interpretation of Euler's method.

Consider the differential equation

$$dy/dt = y \tag{5.2}$$

One of the simplest approaches for obtaining the numerical solution of a differential equation is the method of Euler. This employs Taylor's series but uses only the first two terms of the expansion. Consider the following form of Taylor's series in which the third term is called the remainder term and represents the contribution of all the terms not included in the series.

$$y(t_0 + h) = y(t_0) + y'(t_0)h + y''(\theta)h^2/2 \tag{5.3}$$

where θ lies in the interval (t_0, t_1). For small values of h we may neglect the terms in h^2, and setting $t_1 = t_0 + h$ in (5.3) leads to the formula

$$y_1 = y_0 + hy'_0$$

where the prime denotes differentiation with respect to t and $y'_i = y'(t_i)$. In general,

$$y_{n+1} = y_n + hy'_n \text{ for } n = 0,\ 1,\ 2, \ldots$$

By virtue of (5.2) this may be written

$$y_{n+1} = y_n + hf(t_n, y_n) \text{ for } n = 0,\ 1,\ 2,\ \ldots \tag{5.4}$$

This is known as Euler's method and it is illustrated geometrically in Fig. 5.2. This is an example of the use of a single function value to determine the next step. From (5.3) we can see that the local truncation error, i.e., the error for individual steps is of order h^2.

The method is simple to script and is implemented in the MATLAB function `feuler` as follows.

```
function [tvals, yvals] = feuler(f,tspan, startval,step)
% Euler's method for solving
% first order differential equation dy/dt = f(t,y).
```

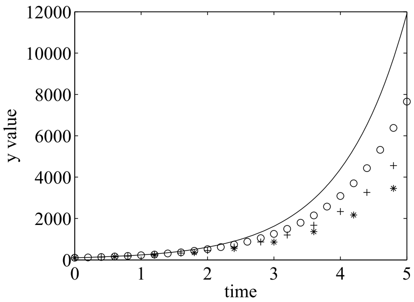

FIGURE 5.3

Points from the Euler solution of $dy/dt = y - 20$ given that $y = 100$ when $t = 0$. Approximate solutions for $h = 0.2$, 0.4, and 0.6 are plotted using o, +, and $*$ respectively. The exact solution is given by the solid line.

```
% Example call: [tvals, yvals]=feuler(f,tspan,startval,step)
% Initial and final value of t are given by tspan = [start finish].
% Initial value of y is given by startval, step size is given by step.
% The function f(t,y) must be defined by the user.
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1);
yvals = startval; tvals = tspan(1);
for i = 2:steps
    y1 = y+step*feval(f,t,y); t1 = t+step;
    %collect values together for output
    tvals = [tvals, t1]; yvals = [yvals, y1];
    t = t1; y = y1;
end
```

Applying this function to the differential equation (5.1) with $K = 1$, $s = 20$, and an initial value of $y = 100$, gives Fig. 5.3, which illustrates how the approximate solution varies for different values of h. The exact value computed from the analytical solution is given for comparison purposes by the solid line. Clearly, in view of the very large errors shown in Fig. 5.3, the Euler method, although simple, requires a very small step h to provide reasonable levels of accuracy. If the differential equation must be solved for a wide range of values of t, the method becomes very expensive in terms of computer time because of the very large number of small steps required to span the interval of interest. In addition, the errors made at each step may accumulate in an unpredictable way. This is a crucial issue, and we discuss this in the next section.

5.3 THE PROBLEM OF STABILITY

To ensure that errors do not accumulate we require that the method for solving the differential equation be stable. We have seen that the error at each step in Euler's method is of order h^2. This error is known as the local truncation error since it tells only how accurate the individual step is, not what the error is for a sequence of steps. The error for a sequence of steps is difficult to find since the error from one step affects the accuracy of the next in a way that is often complex. This leads us to the issue of absolute and relative stability. We now discuss these concepts and examine their effects in relation to a simple equation and explain how the results for this equation may be extended to differential equations in general.

Consider the differential equation

$$dy/dt = Ky \tag{5.5}$$

Since $f(t, y) = Ky$, Euler's method will have the form

$$y_{n+1} = y_n + hKy_n \tag{5.6}$$

Thus, using this recursion repeatedly and assuming that there are no errors in the computation from stage to stage we obtain

$$y_{n+1} = (1 + hK)^{n+1} y_0 \tag{5.7}$$

For small enough h, it is easily shown that this value will approach the exact value e^{Kt}.

To obtain some understanding of how errors propagate when using Euler's method, let us assume that y_0 is perturbed. This perturbed value of y_0 may be denoted by y_0^a where $y_0^a = (y_0 - e_0)$ and e_0 is the error. Thus (5.7) becomes, on using this approximate value instead of y_0,

$$y_{n+1}^a = (1 + hK)^{n+1} y_0^a = (1 + hK)^{n+1}(y_0 - e_0) = y_{n+1} - (1 + hK)^{n+1} e_0$$

Consequently, the initial error will be magnified if $|1 + hK| \geq 1$. After many steps this initial error will grow and may dominate the solution. This is the characteristic of instability and in these circumstances Euler's method is said to be unstable. If, however, $|1 + hK| < 1$, then the error dies away and the method is said to be absolutely stable. Rewriting this inequality leads to the condition for absolute stability:

$$-2 < hK < 0 \tag{5.8}$$

This condition may be too demanding and we may be content if the error does not increase as a proportion of the y values. This is called relative stability. Notice that Euler's method is not absolutely stable for any positive value of K.

The condition for absolute stability can be generalized to an ordinary differential equation of the form of (5.2). It can be shown that the condition becomes

$$-2 < h\partial f/\partial y < 0 \tag{5.9}$$

This inequality implies that, since $h > 0$, $\partial f/\partial y$ must be negative for absolute stability. Fig. 5.4 and Fig. 5.5 give a comparison of the absolute and relative error for $h = 0.1$, for the differential equation $dy/dt = y$ where $y = 1$ when $t = 0$. Fig. 5.4 shows that the error is increasing rapidly and the errors are large for even relatively small step sizes. Fig. 5.5 shows that the error is becoming an increasing proportion of the solution values. Thus the relative error is increasing linearly and so the method is neither relatively stable nor absolutely stable for this problem.

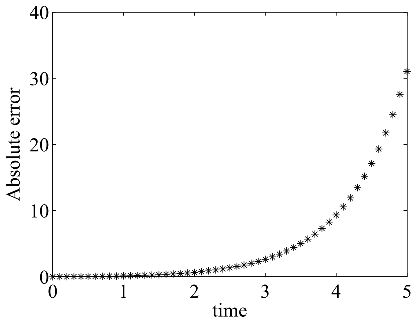

FIGURE 5.4

Absolute errors in the solution of $dy/dt = y$ where $y = 1$ when $t = 0$, using Euler's method with $h = 0.1$.

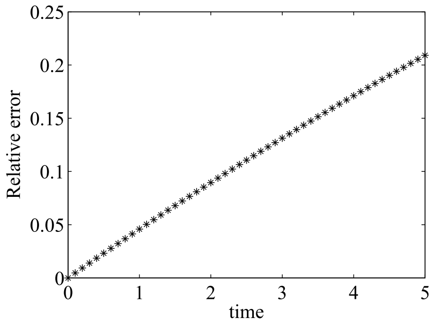

FIGURE 5.5

Relative errors in the solution of $dy/dt = y$ where $y = 1$ when $t = 0$, using Euler's method with $h = 0.1$.

We have seen that Euler's method may be unstable for some values of h. For example, if $K = -100$, then Euler's method is only absolutely stable for $0 < h < 0.02$. Clearly, if we are required to solve the differential equation between 0 and 10, we would require 500 steps. We now consider an improvement to this method called the trapezoidal method which has improved stability features although it is similar in principle to Euler's method.

5.4 THE TRAPEZOIDAL METHOD

The trapezoidal method has the form

$$y_{n+1} = y_n + h\{f(t_n, y_n) + f(t_{n+1}, y_{n+1})\}/2 \text{ for } n = 0, 1, 2, \dots \tag{5.10}$$

Applying the error analysis of Section 5.3 to this problem gives us, from (5.5), that

$$y_{n+1} = y_n + h(Ky_n + Ky_{n+1})/2 \text{ for } n = 0, 1, 2, \dots \tag{5.11}$$

Thus expressing y_{n+1} in terms of y_n gives

$$y_{n+1} = (1 + hK/2)/(1 - hK/2)y_n \text{ for } n = 0, 1, 2, \dots \tag{5.12}$$

Using this result recursively for $n = 0, 1, 2, \dots$ leads to the result

$$y_{n+1} = \{(1 + hK/2)/(1 - hK/2)\}^{n+1} y_0 \tag{5.13}$$

Now, as in Section 5.3, we can obtain some understanding of how error propagates by assuming that y_0 is perturbed by the error e_0 so that it is replaced by $y_0^a = (y_0 - e_0)$. Hence using the same procedure (5.13) becomes

$$y_{n+1} = \{(1 + hK/2)/(1 - hK/2)\}^{n+1}(y_0 - e_0)$$

This leads directly to the result

$$y_{n=1}^{a} = y_{n+1} - \{(1 + hK/2)/(1 - hK/2)\}^{n+1} e_0$$

Thus, we conclude from this that the influence of the error term which involves e_0 will die away if its multiplier is less than unity in magnitude, i.e.,

$$|(1 + hK/2)/(1 - hK/2)| < 1$$

If K is negative, then for all h the method is absolutely stable. For positive K it is not absolutely stable for any h.

This completes the error analysis of this method. However, we note that the method requires a value for y_{n+1} before we can start. An estimate for this value can be obtained by using Euler's method, that is

$$y_{n+1} = y_n + hf(t_n, y_n) \text{ for } n = 0, 1, 2, \ldots$$

This value can now be used in the right-hand side of (5.10) as an estimate for y_{n+1}. This combined method is often known as the Euler–trapezoidal method. The method can be written formally as:

1. Start with n set at zero where n indicates the number of steps taken.
2. Calculate $y_{n+1}^{(1)} = y_n + hf(t_n, y_n)$.
3. Calculate $f(t_{n+1}, y_{n+1}^{(1)})$ where $t_{n+1} = t_n + h$.
4. For $k = 1, 2, \ldots$ calculate

$$y_{n+1}^{(k+1)} = y_n + h\{f(t_n, y_n) + f(t_{n+1}, y_{n+1}^{(k)})\}/2 \tag{5.14}$$

At step 4, when the difference between successive values of y_{n+1} is sufficiently small, increment n by 1 and repeat steps 2, 3, and 4. This method is implemented in MATLAB function `eulertp` thus:

```
function [tvals, yvals] = eulertp(f,tspan,startval,step)
% Euler trapezoidal method for solving
% first order differential equation dy/dt = f(t,y).
% Example call: [tvals, yvals] = eulertp(f,tspan,startval,step)
% Initial and final value of t are given by tspan = [start finish].
% Initial value of y is given by startval, step size is given by step.
% The function f(t,y) must be defined by the user.
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1);
yvals = startval; tvals = tspan(1);
for i = 2:steps
    y1 = y+step*feval(f,t,y);
    t1 = t+step;
    loopcount = 0; diff = 1;
    while abs(diff)>0.05
        loopcount = loopcount+1;
        y2 = y+step*(feval(f,t,y)+feval(f,t1,y1))/2;
```

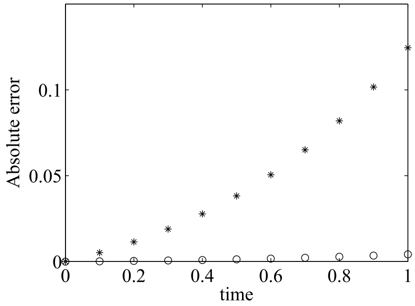

FIGURE 5.6

Solution of $dy/dt = y$ using Euler ($*$) and trapezoidal method, o. Step $h = 0.1$ and $y_0 = 1$ at $t = 0$.

```
        diff = y1-y2; y1 = y2;
    end
    %collect values together for output
    tvals = [tvals, t1]; yvals = [yvals, y1];
    t = t1; y = y1;
end
```

We use `eulertp` to study the performance of this method compared with Euler's method for solving $dy/dt = y$. The results are given in Fig. 5.6 which shows graphs of the absolute errors of the two methods. The difference is clear but although the Euler–trapezoidal method gives much greater accuracy for this problem, in other cases the difference may be less marked. In addition, the Euler–trapezoidal method takes longer.

An important feature of this method is the number of iterations that are required to obtain convergence in step 4. If this is high, the method is likely to be inefficient. However, for the example we just solved, a maximum of two iterations at step 4 was required. This algorithm may be modified to use only one iteration at step 4 in (5.14). This is called Heun's method.

Finally, we examine theoretically how the error in Heun's method compares with Euler's method. By considering the Taylor series expansion of y_{n+1} we can obtain the order of the error in terms of the step size h thus:

$$y_{n+1} = y_n + hy'_n + h^2y''_n/2! + h^3y'_n(\theta)/3! \tag{5.15}$$

where θ lies in the interval (t_n, t_{n+1}). It can be shown that y''_n may be approximated by

$$y''_n = (y'_{n+1} - y'_n)/h + O(h) \tag{5.16}$$

Substituting this expression for y''_n in (5.15) gives

$$\begin{aligned} y_{n+1} &= y_n + hy'_n + h(y'_{n+1} - y'_n)/2! + O(h^3) \\ &= y_n + h(y'_{n+1} + y'_n)/2! + O(h^3) \end{aligned}$$

This shows that the local truncation error is of order h^3 so there is a significant improvement in accuracy over the basic Euler method which has a truncation error of order h^2.

We now describe a range of methods which will be considered under the collective title of Runge–Kutta methods.

5.5 RUNGE–KUTTA METHODS

The Runge–Kutta methods comprise a large family of methods having a common structure. Heun's method, described by (5.14) but with only one iteration of the corrector, can be recast in the form of a simple Runge–Kutta method. We set

$$k_1 = hf(t_n, y_n) \text{ and } k_2 = hf(t_{n+1}, y_{n+1})$$

since

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

We have

$$k_2 = hf(t_{n+1}, y_n + hf(t_n, y_n))$$

Hence from (5.10) we have Heun's method in the form: for $n = 0, 1, 2, \ldots$

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_{n+1}, y_n + k_1) \end{aligned}$$

and

$$y_{n+1} = y_n + (k_1 + k_2)/2$$

This is a simple form of a Runge–Kutta method.

The most commonly used Runge–Kutta method is the classical one; it has the form for each step $n = 0, 1, 2, \ldots$

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h/2, y_n + k_1/2) \\ k_3 &= hf(t_n + h/2, y_n + k_2/2) \\ k_4 &= hf(t_n + h, y_n + k_3) \end{aligned} \tag{5.17}$$

and

$$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6$$

It has a global error of order h^4. The next Runge–Kutta method we consider is a variation on the formula (5.17). It is due to Gill (1951) and takes the form for each step $n = 0, 1, 2, \ldots$

$$\begin{aligned} k_1 &= hf(t_n, y_n)\\ k_2 &= hf(t_n + h/2, y_n + k_1/2)\\ k_3 &= hf(t_n + h/2, y_n + (\sqrt{2}-1)k_1/2 + \left(2-\sqrt{2}\right)k_2/2)\\ k_4 &= hf(t_n + h, y_n - \sqrt{2}k_2/2 + (1+\sqrt{2}/2)k_3) \end{aligned} \tag{5.18}$$

and

$$y_{n+1} = y_n + \left\{k_1 + (2-\sqrt{2})k_2 + (2+\sqrt{2})k_3 + k_4\right\}/6$$

Again this method is fourth-order and has a local truncation error of order h^5 and a global error of order h^4.

A number of other forms of the Runge–Kutta method have been derived which have particularly advantageous properties. The equations for these methods will not be given but their important features are as follows:

1. *Merson–Runge–Kutta method* (Merson, 1957). This method has an error term of order h^5 and in addition allows an estimate of the local truncation error to be obtained at each step in terms of known values.
2. *The Ralston–Runge–Kutta method* (Ralston, 1962). We have some degree of freedom in assigning the coefficients for a particular Runge–Kutta method. In this formula the values of the coefficients are chosen so as to minimize the truncation error.
3. *The Butcher–Runge–Kutta method* (Butcher, 1964). This method provides higher accuracy at each step, the error being of order h^6.

Runge–Kutta methods have the general form for each step $n = 0, 1, 2, \ldots$

$$\begin{aligned} k_1 &= hf(t_n, y_n)\\ k_i &= hf(t_n + hd_i, y_n + \sum_{j=1}^{i-1} c_{ij}k_j), \quad i = 2, 3, \ldots, p \end{aligned} \tag{5.19}$$

$$y_{n+1} = y_n + \sum_{j=1}^{p} b_j k_j \tag{5.20}$$

The order of this general method is p.

The derivation of the various Runge–Kutta methods is based on the expansion of both sides of (5.20) as a Taylor's series and equating coefficients. This is a relatively straightforward idea but involves lengthy algebraic manipulation.

We now discuss the stability of the Runge–Kutta methods. Since the instability which may arise in the Runge–Kutta methods can usually be reduced by a step size reduction, it is known as partial instability. To avoid repeated reduction of the value of h and re-running the method, an estimate of the value of h which will provide stability for the fourth-order Runge–Kutta methods is given by the inequality

$$-2.78 < h\partial f/\partial y < 0$$

In practice $\partial f/\partial y$ may be approximated using the difference of successive values of f and y.

Finally, it is interesting to see how we can provide an elegant MATLAB function for the general Runge–Kutta method given by (5.20) and (5.19). We define two vectors **d** and **b**, where **d** contains the coefficients d_i in (5.19) and **b** contains the coefficients b_j in (5.20), and a matrix **c** which contains the coefficients c_{ij} in (5.19). If the computed values of the k_j are assigned to a vector **k**, then the MATLAB statements that generate the values of the function and the new value of y are relatively simple; they will have the form

```
k(1) = step*feval(f,t,y);
for i = 2:p
    k(i)=step*feval(f,t+step*d(i),y+c(i,1:i-1)*k(1:i-1)’);
end
y1 = y+b*k’;
```

This is of course repeated for each step. A MATLAB function, `rkgen`, based on this is given below. Since `c` and `d` are easily changed in the script, any form of Runge–Kutta method can be implemented using this function and it is useful for experimenting with different techniques.

```
function[tvals,yvals] = rkgen(f,tspan,startval,step,method)
% Runge Kutta methods for solving
% first order differential equation dy/dt = f(t,y).
% Example call:[tvals,yvals]=rkgen(f,tspan,startval,step,method)
% The initial and final values of t are given by tspan = [start finish].
% Initial y is given by startval and step size is given by step.
% The function f(t,y) must be defined by the user.
% The parameter method (1, 2 or 3) selects
% Classical, Butcher or Merson RK respectively.
b = [ ]; c = [ ]; d = [ ];
switch method
    case 1
        order = 4;
        b = [ 1/6 1/3 1/3 1/6]; d = [0 .5 .5 1];
        c=[0 0 0 0;0.5 0 0 0;0 .5 0 0;0 0 1 0];
        disp(’Classical method selected’)
    case 2
        order = 6;
        b = [0.07777777778 0 0.355555556 0.13333333 ...
             0.355555556 0.0777777778];
        d = [0 .25 .25 .5 .75 1];
        c(1:4,:) = [0 0 0 0 0 0;0.25 0 0 0 0 0;0.125 0.125 0 0 0 0; ...
                    0 -0.5 1 0 0 0];
        c(5,:) = [.1875 0 0 0.5625 0 0];
        c(6,:) = [-.4285714 0.2857143 1.714286 -1.714286 1.1428571 0];
        disp('Butcher method selected')
    case 3
        order = 5;
        b = [1/6 0 0 2/3 1/6];
        d = [0 1/3 1/3 1/2 1];
        c = [0 0 0 0 0;1/3 0 0 0 0;1/6 1/6 0 0 0;1/8 0 3/8 0 0; ...
            1/2 0 -3/2 2 0];
        disp('Merson method selected')
    otherwise
        disp('Invalid selection')
end
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1);
yvals = startval; tvals = tspan(1);
for j = 2:steps
    k(1) = step*feval(f,t,y);
    for i = 2:order
        k(i) = step*feval(f,t+step*d(i),y+c(i,1:i-1)*k(1:i-1)');
    end
    y1 = y+b*k'; t1 = t+step;
    %collect values together for output
    tvals = [tvals, t1]; yvals = [yvals, y1];
    t = t1; y = y1;
end
```

A further issue that needs to be considered is that of the adaptive step size adjustment. Where a function is relatively smooth in the area of interest, a large step may be used throughout the region. If the region is such that rapid changes in y occur for small changes in t, then a small step size is required. However, for functions where both these regions exist, then rather than use a small step in the whole region, adaptive step size adjustment would be more efficient. The details of producing this step adjustment are not provided here; however, for an elegant discussion see Press et al. (1990). This type of procedure is implemented for Runge–Kutta methods in the MATLAB functions `ode23` and `ode45`. The basic algorithm for these functions is the Runge–Kutta–Fehlberg algorithm with adaptive step size. However the difference between the two functions is that `ode23` uses second and third order formulae while `ode45` uses fourth and fifth order formulae.

Fig. 5.7 plots the relative errors in the solution of the specific differential equation $dy/dt = -y$ by the classical, Merson and Butcher–Runge–Kutta methods using the MATLAB script: `e4s502.m`:

```
% e4s502.m
yprime = @(t,y) -y;
char = 'o*+';
for meth = 1:3
    [t, y] = rkgen(yprime,[0 3],1,0.25,meth);
```

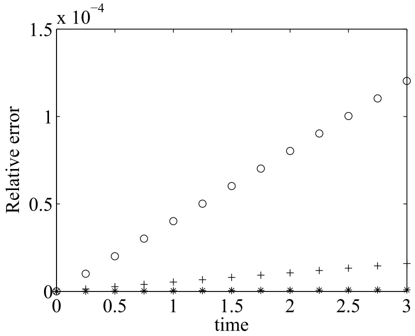

FIGURE 5.7

Solution of $dy/dt = -y$. The $*$ represents Butcher's method, + Merson's method, and o the classical method.

```
    re = (y-exp(-t))./exp(-t);
    plot(t,re,char(meth))
    hold on
end
hold off, axis([0 3 0 1.5e-4])
xlabel('Time'), ylabel('Relative error')
```

It is clear from the graphs that Butcher's method is the best and both Butcher's and Merson's methods are significantly more accurate than the classical method.

5.6 PREDICTOR–CORRECTOR METHODS

The trapezoidal method, which has already been described, is a simple example of both a Runge–Kutta method and a predictor–corrector method with a truncation error of order h^3. The predictor–corrector methods we consider now have much smaller truncation errors. As an initial example we consider the Adams–Bashforth–Moulton method. This method is based on the following equations:

$$\begin{aligned} y_{n+1} &= y_n + h(55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3})/24 \quad &(P) \\ y'_{n+1} &= f(t_{n+1}, y_{n+1}) \quad &(E) \end{aligned} \tag{5.21}$$

and

$$\begin{aligned} y_{n+1} &= y_n + h\left(9y'_{n+1} + 19y'_n - 5y'_{n-1} + y'_{n-2}\right)/24 \quad &(C) \\ y'_{n+1} &= f(t_{n+1}, y_{n+1}) \quad &(E) \end{aligned} \tag{5.22}$$

where $t_{n+1} = t_n + h$. In (5.21) we use the predictor equation (P), followed by a function evaluation (E). Then in (5.22) we use the corrector equation (C), followed by a function evaluation (E). The truncation error for both the predictor and corrector is $O(h^5)$. The first equation in the system (5.21) requires a number of initial values to be known before y can be calculated.

After each application of (5.21) and (5.22), i.e., a complete *PECE* step, the independent variable t_n is incremented by h, n is incremented by one and the process repeated until the differential equation has been solved in the range of interest. The method is started with $n = 3$ and consequently the values of y_3, y_2, y_1, and y_0 must be known before the method can be applied. For this reason, it is called a *multi-point method*. In practice y_3, y_2, y_1, and y_0 must be obtained using a self-starting procedure such as one of the Runge–Kutta methods described in Section 5.5. The self-starting procedure chosen should have the same order truncation error as the predictor–corrector method.

The Adams–Bashforth–Moulton method is often used since its stability is relatively good. Its range of absolute stability in *PECE* mode is

$$-1.25 < h\partial f/\partial y < 0$$

Apart from the need for initial starting values, the Adams–Bashforth–Moulton method in the *PECE* mode requires less computation at each step than the fourth-order Runge–Kutta method. For a true comparison of these methods, however, it is necessary to consider how they behave over a range of problems since applying any method to some differential equations results, at each step, in a growth of error that ultimately swamps the calculation since the step is outside the range of absolute stability.

The Adams–Bashforth–Moulton method is implemented by the function `abm`. It should be noted that errors arise from the choice of starting procedure, in this case the classical Runge–Kutta method. It is, however, easy to amend this function to include the option of entering highly accurate initial values.

```
function [tvals, yvals] = abm(f,tspan,startval,step)
% Adams Bashforth Moulton method for solving
% first order differential equation dy/dt = f(t,y).
% Example call: [tvals, yvals]=abm(f,tspan,startval,step)
% The initial and final values of t are given by tspan = [start finish].
% Initial y is given by startval and step size is given by step.
% The function f(t,y) must be defined by the user.
% 3 steps of Runge-Kutta are required so that ABM method can start.
% Set up matrices for Runge-Kutta methods
b = [ ]; c = [ ]; d = [ ]; order = 4;
b = [1/6 1/3 1/3 1/6]; d = [0 .5 .5 1];
c = [0 0 0 0;0.5 0 0 0;0 .5 0 0;0 0 1 0];
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1); fval(1) = feval(f,t,y);
ys(1) = startval; yvals = startval; tvals = tspan(1);
for j = 2:4
    k(1) = step*feval(f,t,y);
    for i = 2:order
        k(i) = step*feval(f,t+step*d(i),y+c(i,1:i-1)*k(1:i-1)');
    end
    y1 = y+b*k'; ys(j) = y1; t1 = t+step;
    fval(j) = feval(f,t1,y1);
    %collect values together for output
```

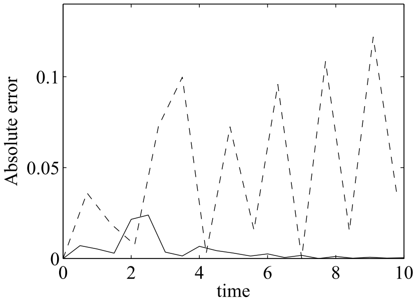

FIGURE 5.8

Absolute error in solution of $dy/dt = -2y$ using the Adams–Bashforth–Moulton method. The *solid line* plots the errors with a step size of 0.5. The *dot-dashed line* plots the errors with step size 0.7.

```
    tvals = [tvals,t1]; yvals = [yvals,y1];
    t = t1; y = y1;
end
%ABM now applied
for i = 5:steps
    y1 = ys(4)+step*(55*fval(4)-59*fval(3)+37*fval(2)-9*fval(1))/24;
    t1 = t+step; fval(5) = feval(f,t1,y1);
    yc = ys(4)+step*(9*fval(5)+19*fval(4)-5*fval(3)+fval(2))/24;
    fval(5) = feval(f,t1,yc);
    fval(1:4) = fval(2:5);
    ys(4) = yc;
    tvals = [tvals,t1]; yvals = [yvals,yc];
    t = t1; y = y1;
end
```

Fig. 5.8 illustrates the behavior of the Adams–Bashforth–Moulton method when applied to the specific problem $dy/dt = -2y$ where $y = 1$ when $t = 0$, using a step size equal to 0.5 and 0.7 in the interval 0 to 10. It is interesting to note that for this problem, since $\partial f/\partial y = -2$, the range of steps for absolute stability is $0 \le h \le 0.625$. For $h = 0.5$, a value inside the range of absolute stability, the plot shows that the absolute error does die away. However, for $h = 0.7$, a value outside the range of absolute stability, the plot shows that the absolute error increases.

5.7 HAMMING'S METHOD AND THE USE OF ERROR ESTIMATES

The method of Hamming (1959) is based on the following pair of predictor–corrector equations:

$$\begin{aligned} y_{n+1} &= y_{n-3} + 4h\left(2y'_n - y'_{n-1} + 2y'_{n-2}\right)/3 \quad (P) \\ y'_{n+1} &= f\left(t_{n+1}, y_{n+1}\right) \qquad\qquad\qquad (E) \end{aligned} \tag{5.23}$$

$$y_{n+1} = \left\{9y_n - y_{n-2} + 3h\left(y'_{n+1} + 2y'_n - y'_{n-1}\right)\right\}/8 \quad (C)$$

$$y'_{n+1} = f\left(t_{n+1}, y_{n+1}\right) \quad (E)$$

where $t_{n+1} = t_n + h$.

The first equation (P) is used as the predictor and the third as the corrector (C). To obtain a further improvement in accuracy at each step in the predictor and corrector we modify these equations using expressions for the local truncation errors. Approximations for these local truncation errors can be obtained using the predicted and corrected values of the current approximation to y. This leads to the equations

$$y_{n+1} = y_{n-3} + 4h\left(y'_n - y'_{n-1} + 2y'_{n-2}\right)/3 \quad (P) \tag{5.24}$$

$$(y^M)_{n+1} = y_{n+1} - 112(Y_P - Y_C)/121 \tag{5.25}$$

In this equation Y_P and Y_C represent the predicted and corrected value of y at the nth step.

$$y^*_{n+1} = \left\{9y_n - y_{n-2} + 3h((y^M)'_{n+1} + 2y'_n - y'_{n-1})\right\}/8 \quad (C) \tag{5.26}$$

In this equation $(y^M)'_{n+1}$ is the value of y'_{n+1} calculated using the modified value of y_{n+1} which is $(y^M)_{n+1}$.

$$y_{n+1} = y^*_{n+1} + 9(y_{n+1} - y^*_{n+1}) \tag{5.27}$$

Eq. (5.24) is the predictor and (5.25) modifies the predicted value by using an estimate of the truncation error. Eq. (5.26) is the corrector which is modified by (5.27) using an estimate of the truncation error. The equations in this form are each used only once before n is incremented and the steps repeated again. This method is implemented as MATLAB function `fhamming` thus:

```
function [tvals, yvals] = fhamming(f,tspan,startval,step)
% Hamming's method for solving
% first order differential equation dy/dt = f(t,y).
% Example call: [tvals, yvals]=fhamming(f,tspan,startval,step)
% The initial and final values of t are given by tspan = [start finish].
% Initial y is given by startval and step size is given by step.
% The function f(t,y) must be defined by the user.
% 3 steps of Runge-Kutta are required so that hamming can start.
% Set up matrices for Runge-Kutta methods
b = [ ]; c = [ ]; d = [ ]; order = 4;
b = [1/6 1/3 1/3 1/6]; d = [0 0.5 0.5 1];
c = [0 0 0 0;0.5 0 0 0;0 0.5 0 0;0 0 1 0];
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1);
fval(1) = feval(f,t,y);
ys(1) = startval;
yvals = startval; tvals = tspan(1);
for j = 2:4
```

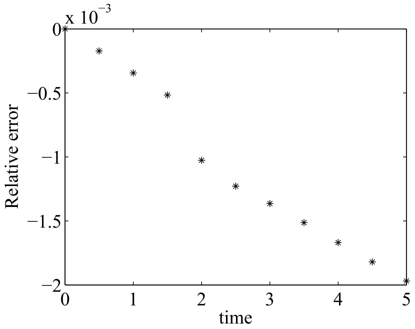

FIGURE 5.9

Relative error in the solution of $dy/dt = y$ where $y = 1$ when $t = 0$, using Hamming's method with a step size of 0.5.

```
    k(1) = step*feval(f,t,y);
    for i = 2:order
        k(i) = step*feval(f,t+step*d(i),y+c(i,1:i-1)*k(1:i-1)');
    end
    y1 = y+b*k'; ys(j) = y1; t1 = t+step; fval(j) = feval(f,t1,y1);
    %collect values together for output
    tvals = [tvals, t1]; yvals = [yvals, y1]; t = t1; y = y1;
end
%Hamming now applied
for i = 5:steps
    y1 = ys(1)+4*step*(2*fval(4)-fval(3)+2*fval(2))/3;
    t1 = t+step; y1m = y1;
    if i>5, y1m = y1+112*(c-p)/121; end
    fval(5) = feval(f,t1,y1m);
    yc = (9*ys(4)-ys(2)+3*step*(2*fval(4)+fval(5)-fval(3)))/8;
    ycm = yc+9*(y1-yc)/121;
    p = y1; c = yc;
    fval(5) = feval(f,t1,ycm); fval(2:4) = fval(3:5);
    ys(1:3) = ys(2:4); ys(4) = ycm;
    tvals = [tvals, t1]; yvals = [yvals, ycm];
    t = t1;
end
```

The choice of h must be made carefully so that the error does not increase without bound. Fig. 5.9 shows Hamming's method used to solve the equation $dy/dt = y$. This is the problem used in Section 5.6.

5.8 ERROR PROPAGATION IN DIFFERENTIAL EQUATIONS

In the preceding sections, we described various techniques for solving differential equations and the order, or a specific expression, for the truncation error at each step was given. As we discussed in Section 5.3 for the Euler and trapezoidal methods, it is important to examine not only the magnitude of the error at each step but also how that error accumulates as the number of steps taken increases.

For the predictor–corrector method described before, it can be shown that the predictor–corrector formulae introduce additional spurious solutions. As the iterative process proceeds, for some problems the effect of these spurious solutions may be to overwhelm the true solution. In these circumstances the method is said to be unstable. Clearly, we seek stable methods where the error does not develop in an unpredictable and unbounded way.

It is important to examine each numerical method to see if it is stable. In addition, if it is not stable for all differential equations we should provide tests to determine when it can be used with confidence. The theoretical study of stability for differential equations is a major undertaking and it is not intended to include a detailed analysis here. In Section 5.9 we summarize the stability characteristics of specific methods and compare the performance of the major methods considered on a number of example differential equations.

5.9 THE STABILITY OF PARTICULAR NUMERICAL METHODS

A good discussion of the stability of many of the numerical methods for solving first-order differential equations is given by Ralston and Rabinowitz (1978) and Lambert (1973). Some of the more significant features, assuming all variables are real, are as follows:

1. Euler and trapezoidal methods. For a detailed discussion see Sections 5.3 and 5.4.
2. Runge–Kutta methods: Runge–Kutta methods do not introduce spurious solutions but instability may arise for some values of h. This may be removed by reducing h to a sufficiently small value. We have already described how the Runge–Kutta methods are less efficient than the predictor–corrector methods because of the greater number of function evaluations that may be required at each step. If h is reduced too far, the number of function evaluations required may make the method uneconomic. The restriction on the size of the interval required to maintain stability may be estimated from the inequality $M < h\partial f/\partial y < 0$ where M is dependent on the particular Runge–Kutta method being used and may be estimated. Clearly this emphasizes the need for careful step size adjustment during the solution process. This is efficiently implemented in the functions `ode23` and `ode45` so that this question does not present a problem when applying these MATLAB functions.
3. Adams–Bashforth–Moulton method: In *PECE* mode the range of absolute stability is given by $-1.25 < h\partial f/\partial y < 0$, implying that $\partial f/\partial y$ must be negative for absolute stability.
4. Hamming's method: In *PECE* mode the range of absolute stability is given by $-0.5 < h\partial f/\partial y < 0$, again implying that $\partial f/\partial y$ must be negative for absolute stability.

Notice that the formulae given for estimating the step size can be difficult to use if f is a general function of y and t. However, in some cases the derivative of f is easily calculated, for example, when $f = Cy$ where C is a constant.

We now give some results of applying the methods discussed in previous sections to solve more general problems. The script `e4s503.m` solves the three examples that follow by setting `example` equal to 1, 2, or 3 in the first line of the script.

```
% e4s503.m
example = 1;
switch example
    case 1
        yprime = @(t,y) 2*t*y;
        sol = @(t) 2*exp(t^2);
        disp('Solution of dy/dt = 2yt')
        t0 = 0; y0 = 2;
    case 2
        yprime = @(t,y) (cos(t)-2*y*t)/(1+t^2);
        sol = @(t) sin(t)/(1+t^2);
        disp('Solution of (1+t^2)dy/dt = cos(t)-2yt')
        t0 = 0; y0 = 0;
    case 3
        yprime = @(t,y) 3*y/t;
        disp('Solution of dy/dt = 3y/t')
        sol = @(t) t^3;
        t0 = 1; y0 = 1;
end
tf = 2; tinc = 0.25; steps = floor((tf-t0)/tinc+1);
[t,x1] = abm(yprime,[t0 tf],y0,tinc);
[t,x2] = fhamming(yprime,[t0 tf],y0,tinc);
[t,x3] = rkgen(yprime,[t0 tf],y0,tinc,1);
disp('t        abm        Hamming    Classical     Exact')
for i = 1:steps
    fprintf('%4.2f%12.7f%12.7f',t(i),x1(i),x2(i))
    fprintf('%12.7f%12.7f\n',x3(i),sol(t(i)))
end
```

Example 5.1. Solve

$$dy/dt = 2yt \text{ where } y = 2 \text{ when } t = 0.$$

Exact solution: $y = 2\exp(t^2)$. Running the script `e4s503` with `example = 1` produces the following output:

```
Solution of dy/dt = 2yt
t        abm        Hamming    Classical     Exact
0.00   2.0000000   2.0000000   2.0000000   2.0000000
0.25   2.1289876   2.1289876   2.1289876   2.1289889
0.50   2.5680329   2.5680329   2.5680329   2.5680508
0.75   3.5099767   3.5099767   3.5099767   3.5101093
```

```
1.00   5.4340314   5.4294215   5.4357436   5.4365637
1.25   9.5206761   9.5152921   9.5369365   9.5414664
1.50  18.8575896  18.8690552  18.9519740  18.9754717
1.75  42.1631012  42.2832017  42.6424234  42.7618855
2.00 106.2068597 106.9045567 108.5814979 109.1963001
```

Example 5.2. Solve

$$(1+t^2)dy/dt + 2ty = \cos t \text{ where } y = 0 \text{ when } t = 0$$

Exact solution: $y = (\sin t)/(1+t^2)$. Running script `e4s503` with `example = 2` produces the following output:

```
Solution of (1+t^2)dy/dt = cos(t)-2yt
t         abm        Hamming    Classical     Exact
0.00   0.0000000   0.0000000   0.0000000   0.0000000
0.25   0.2328491   0.2328491   0.2328491   0.2328508
0.50   0.3835216   0.3835216   0.3835216   0.3835404
0.75   0.4362151   0.4362151   0.4362151   0.4362488
1.00   0.4181300   0.4196303   0.4206992   0.4207355
1.25   0.3671577   0.3705252   0.3703035   0.3703355
1.50   0.3044513   0.3078591   0.3068955   0.3069215
1.75   0.2404465   0.2432427   0.2421911   0.2422119
2.00   0.1805739   0.1827267   0.1818429   0.1818595
```

Example 5.3. Solve

$$dy/dt = 3y/t \text{ where } y = 1 \text{ when } t = 1$$

Exact solution: $y = t^3$. Running script `e4s503` with `example = 3` produces the following output:

```
Solution of dy/dt = 3y/t
t         abm        Hamming    Classical     Exact
1.00   1.0000000   1.0000000   1.0000000   1.0000000
1.25   1.9518519   1.9518519   1.9518519   1.9531250
1.50   3.3719182   3.3719182   3.3719182   3.3750000
1.75   5.3538346   5.3538346   5.3538346   5.3593750
2.00   7.9916917   7.9919728   7.9912355   8.0000000
```

Examples 5.2 and 5.3 appear to show that there is little difference between the three methods considered and they are all fairly successful for the step size $h = 0.25$ in this range. Example 5.1 is a relatively difficult problem in which the classical Runge–Kutta method performs well.

For a further comparison, we now use the MATLAB function `ode113`. This employs a predictor–corrector method based on the PECE approach described in Section 5.6 for the Adams–Bashforth–Moulton method. However the method implemented in the `ode113` is of variable order. The standard call of the function takes the form

```
[t,y] = ode113(f,tspan,y0,options);
```

where `f` is the name of the function providing the right-hand sides of the system of differential equations; `tspan` is the range of solution for the differential equation, given as a vector `[to tfinal]`; `y0` is the vector of initial values for the differential equation at time $t = 0$; `options` is an optional parameter providing additional settings for the differential equation such as accuracy. To illustrate the use of this function we consider the example

$$dy/dt = 2yt \text{ with initial condition } y = 2 \text{ when } t = 0$$

The call to solve this differential equation is

```
>> options = odeset('RelTol', 1e-5,'AbsTol',1e-6);
>> [t,yy] = ode113(@(t,x) 2*t*x,[0,2],[2],options); y = yy', time = t'
```

The result of executing these statements is

```
y =
 Columns 1 through 7
   2.0000   2.0000   2.0000   2.0002   2.0006   2.0026   2.0103
 Columns 8 through 14
   2.0232   2.0414   2.0650   2.0943   2.1707   2.2731   2.4048
 Columns 15 through 21
   2.5703   2.7755   3.0279   3.3373   3.7161   4.1805   4.7513
 Columns 22 through 28
   5.4557   6.3290   7.4177   8.7831  10.5069  15.5048  22.7912
 Columns 29 through 32
  34.6321  54.3997  88.3328 109.1944

time =
 Columns 1 through 7
        0   0.0022   0.0045   0.0089   0.0179   0.0358   0.0716
 Columns 8 through 14
   0.1073   0.1431   0.1789   0.2147   0.2862   0.3578   0.4293
 Columns 15 through 21
   0.5009   0.5724   0.6440   0.7155   0.7871   0.8587   0.9302
 Columns 22 through 28
   1.0018   1.0733   1.1449   1.2164   1.2880   1.4311   1.5599
 Columns 29 through 32
   1.6887   1.8175   1.9463   2.0000
```

Although a direct comparison between each step is not possible, because `ode113` uses a variable step size, we can compare the result for $t = 2$ with the results given for Example 5.1 above. This shows that the final y value given by `ode113` is better than those given by the other methods.

5.10 SYSTEMS OF SIMULTANEOUS DIFFERENTIAL EQUATIONS

The numerical techniques we have described for solving a single first-order differential equation can be applied, after simple modification, to solve systems of first-order differential equations. Systems of differential equations arise naturally from mathematical models of the physical world. In this section we consider systems of differential equations. The first is the Zeeman model and the second is the predator–prey problem.

5.10.1 ZEEMAN MODEL

Zeeman introduced a simplified model of the heart and incorporates ideas from catastrophe theory. The model is described briefly here but more detail is given in the excellent text of Beltrami (1987). The resulting system of differential equations will be solved using the MATLAB function `ode23` and the graphical facilities of MATLAB will help to clarify the interpretation of the results.

The starting point for this model of the heart is Van der Pol's equation which may be written in the form

$$dx/dt = u - \mu(x^3/3 - x)$$
$$du/dt = -x$$

This is a system of two simultaneous equations. The choice of this differential equation reflects our wish to imitate the beat of the heart. The fluctuation in the length of the heart fiber, as the heart contracts and dilates subject to an electrical stimulus, thus pumping blood through the system, may be represented by this pair of differential equations. The fluctuation has certain subtleties which our model should allow for. Starting from the relaxed state, the contraction begins with the application of the stimulus slowly at first and then becomes faster, so giving a sufficient final impetus to the blood. When the stimulus is removed, the heart dilates slowly at first and then more rapidly until the relaxed state is again reached and the cycle can begin again.

To follow this behavior, the Van der Pol equation requires some modification so that the x variable represents the length of heart fiber and the u variable can be replaced by one which represents the stimulus applied to the heart. This is achieved by making the substitution $s = -u/\mu$, where s represents the stimulus and μ is a constant. Since ds/dt is equal to $(-du/dt)/\mu$, it follows that $du/dt = -\mu ds/dt$. Hence we obtain

$$dx/dt = \mu(-s - x^3/3 + x)$$
$$ds/dt = x/\mu$$

If these simultaneous differential equations are solved for s and x for a range of time values, we find that s and x oscillate in a manner representing the fluctuations in the heart fiber length and stimulus. However, Zeeman proposed the introduction into this model of a tension factor p, where $p > 0$, in an attempt to account for the effects of increased blood pressure in terms of increased tension on the heart fiber. The model he suggested has the form

$$dx/dt = \mu(-s - x^3/3 + px)$$
$$du/dt = x/\mu$$

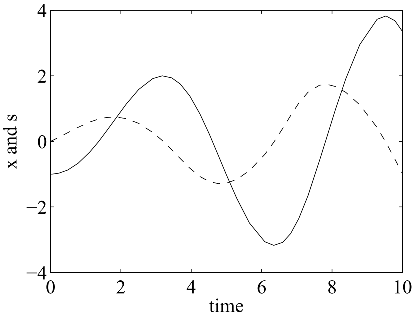

FIGURE 5.10

Solution of Zeeman's model with $p = 1$ and accuracy 0.005. The solid line represents s and the dashed line represents x.

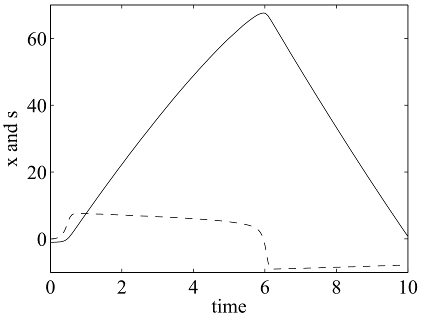

FIGURE 5.11

Solution of Zeeman's model with $p = 20$ and accuracy 0.005. The *solid line* represents s and the *dashed line* represents x.

Although the motivation for such a modification is plausible, the effects of these changes are by no means obvious.

This problem provides an interesting opportunity to apply MATLAB to simulate the heart beat in an experimental environment that allows us to monitor its changes under the effects of differing tension values. The following script solves the differential equations and draws various graphs.

```
% e4s504.m Solving Zeeman's Catastrophe model of the heart
clear all
p = input('enter tension value ');
simtime = input('enter runtime ');
acc = input('enter accuracy value ');
xprime = @(t,x) [0.5*(-x(2)-x(1)^3/3+p*x(1)); 2*x(1)];
options = odeset('RelTol',acc);
initx = [0 -1]';
[t x] = ode23(xprime,[0 simtime],initx,options);
% Plot results against time
plot(t,x(:,1),'--',t,x(:,2),'-')
xlabel('Time'), ylabel('x and s')
```

In the preceding function definition, $\mu = 0.5$. Fig. 5.10 shows graphs of the fiber length x and the stimulus s against time for a relatively small tension factor set at 1. The graphs show that a steady periodic oscillation of fiber length for this tension value is achieved for small stimulus values. However, Fig. 5.11 plots x and s against time with the tension set at 20. This shows that the behavior of the oscillation is clearly more labored and that much larger values of stimulus are required to produce the fluctuations in fiber length for the much higher tension value. Thus, the graphs show the deterioration in the beat with increasing tension. The results parallel the expected physical effects and also give some degree of experimental support to the validity of this simple model.

A further interesting study can be made. The interrelation of the three parameters x, s, and p can be represented by a three-dimensional surface called the cusp catastrophe surface. This surface can be

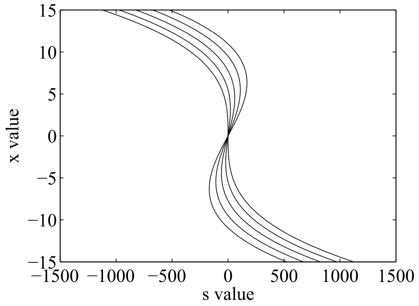

FIGURE 5.12

Sections of the cusp catastrophe curve in Zeeman's model for $p = 0 : 10 : 40$.

shown to have the form

$$-s - x^3/3 + px = 0$$

See Beltrami (1987) for a more detailed explanation. Fig. 5.12 shows a series of sections of the cusp catastrophe curve for $p = 0 : 10 : 40$. The curve has a pleat which becomes increasingly pronounced in the direction of increasing p. High tension or high p value consequently corresponds to movement on the sharply pleated part of this surface and consequently provides smaller changes in the heart fiber length relative to the stimulus.

5.10.2 THE PREDATOR–PREY PROBLEM

A system of differential equations which models the interaction of competing or predator–prey populations is based on the Volterra equations and may be written in the form

$$\begin{aligned} dP/dt &= K_1P - CPQ \\ dQ/dt &= -K_2Q + DPQ \end{aligned} \tag{5.28}$$

together with the initial conditions

$$Q = Q_0 \text{ and } P = P_0 \text{ at time } t = 0$$

The variables P and Q give the size of the prey and predator populations, respectively, at time t. These two populations interact and compete. K_1, K_2, C, and D are positive constants. K_1 relates to the rate of growth of the prey population P, and K_2 relates to the rate of decay of the predator population Q. It seems reasonable to assume that the number of encounters of predator and prey is proportional to P multiplied by Q and that a proportion C of these encounters will be fatal to members of the prey population. Thus the term CPQ gives a measure of the decrease in the prey population and the unrestricted growth in this population, which could occur assuming ample food, must be modified by the subtraction of this term. Similarly the decrease in the population of the predator must be modified by the addition of the term DPQ since the predator population gains food from its encounters with its prey and therefore more of the predators survive.

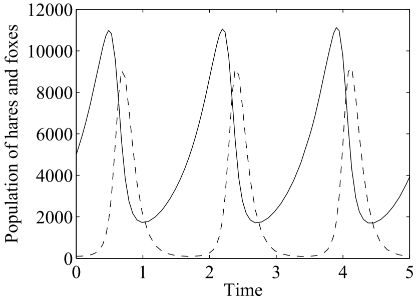

FIGURE 5.13

Variation in the population of lynxes (*dashed line*) and hares (*solid line*) against time, beginning with 5000 hares and 100 lynxes. Accuracy 0.005.

The solution of the differential equation depends on the specific values of the constants and will often result in nature in a stable cyclic variation of the populations. This is because as the predators continue to eat the prey, the prey population will fall and become insufficient to support the predator population which itself then falls. However, as the predator population falls, more of the prey survive and consequently the prey population will then increase. This in turn leads to an increase in the predator population since it has more food and the cycle begins again. This cycle maintains the predator and prey populations between certain upper and lower limits. The Volterra differential equations can be solved directly but this solution does not provide a simple relation between the size of the predator and prey populations; therefore, numerical methods of solution should be applied. An interesting description of this problem is given by Simmons (1972).

We now use MATLAB to study the behavior of a system of equations of the form (5.28) applied to the interaction of the lynx and its prey, the hare. The choice of the constants K_1, K_2, C, and D is not a simple matter if we wish to obtain a stable situation where the populations of the predator and prey never die out completely but oscillate between upper and lower limits. The MATLAB script below uses $K_1 = 2$, $K_2 = 10$, $C = 0.001$, and $D = 0.002$, and considers the interaction of a population of lynxes and hares where it is assumed that this interaction is the crucial feature in determining the size of the two populations. With an initial population of 5000 hares and 100 lynxes, the script `e4s505.m` uses these values to produce the graph in Fig. 5.13.

```
% e4s505.m
% x(1) and x(2) are hare and lynx populations.
simtime = input('enter runtime ');
acc = input('enter accuracy value ');
fv = @(t,x) [2*x(1)-0.001*x(1)*x(2); -10*x(2)+0.002*x(1)*x(2)];
initx = [5000 100]';
options = odeset('RelTol',acc);
[t x] = ode23(fv,[0 simtime],initx,options);
plot(t,x(:,1),'k',t,x(:,2),'k--')
xlabel('Time'), ylabel('Population of hares and lynxes')
```

For these parameters, there is a remarkably wide variation in the populations of hares and lynxes. The lynx population, although periodically small, still recovers following a recovery of the hare population.

5.11 HIGHER-ORDER DIFFERENTIAL EQUATIONS

Higher-order differential equations contain differential terms of second order or above, for example, the third-order differential term d^3y/dx^3. Higher-order differential equations can be solved by converting them to a system of first-order differential equations, or in some case, solved by direct methods. In the following subsections, we describe the process of transforming from a higher-order equation to set of first-order differential equations, and also describe one direct method for solving second-order differential equations.

5.11.1 CONVERSION TO A SET OF SIMULTANEOUS FIRST-ORDER DIFFERENTIAL EQUATIONS

The process of converting a higher-order differential equation to a set of first-order simultaneous equations is best described by an example. To illustrate this, consider the third-order differential equation

$$d^3u/dx^3 - cd^2u/dx^2 + (df/du)(du/dx) - (b/c)u = 0 \tag{5.29}$$

where $f = u(1-u)(u-a)$, $0 < a < 1$, and $b > 0$.

This equation describes a simplified model of the behavior of pulses in a squid axon (Iqbal, 1999). We require the derivative of f. Thus

$$df/du = (1-u)(u-a) + u(-1)(u-1) + u(1-u) = -3u^2 + 2(a+1)u - a$$

If we substitute $v = du/dx$, $w = dv/dx = d^2u/dx^2$ then (5.29) becomes

$$\begin{aligned} du/dx &= v \\ dv/dx &= w \\ dw/dx &= cw + 3u^2 - 2(a+1) + av + (b/c)u \end{aligned} \tag{5.30}$$

Thus, the third-order differential equation has been replaced by a system of three first-order differential equations.

If we have an nth-order differential equation of the form

$$a_n d^n y/dt^n + a_{n-1} d^{n-1} y/dt^{n-1} + \ldots + a_0 y = f(t, y) \tag{5.31}$$

by making the substitutions

$$P_0 = y \text{ and } dP_{i-1}/dt = P_i \text{ for } i = 1, 2, \ldots, n-1 \tag{5.32}$$

(5.31) becomes

$$a_n dP_{n-1}/dt = f(t, y) - a_{n-1}P_{n-1} - a_{n-2}P_{n-2} - \ldots - a_0 P_0 \tag{5.33}$$

Now (5.32) and (5.33) together constitute a system of n first-order differential equations. Initial values will be given for (5.31) in terms of the various order derivatives P_i for $i = 1, 2, ..., n - 1$ at some initial value t_0 and these can easily be translated into initial conditions for the system of Eqs. (5.32) and (5.33). In general, the solutions of the original nth order differential equation and the system of first-order differential equations, (5.32) and (5.33), are the same. In particular, the numerical solution will provide the values of y for a specified range of t. An excellent discussion of the equivalence of the solutions of the two problems is given in Simmons (1972). We can see from this description that any order differential equation of the form (5.31) with given initial values can be reduced to solving a system of first-order differential equations. This argument is easily extended to the more general nth-order differential equation by making exactly the same substitutions as above in

$$d^n y/dt^n = f(t, y, y', \dots y^{(n-1)})$$

where $y^{(n-1)}$ denotes the $(n-1)$th-order derivative of y.

5.11.2 NEWMARK'S METHOD

Newmark's method, (Newmark, 1959), allows the direct solution of a second-order differential equation or a system of second-order differential equations without the need for the transformation to a pair of simultaneous first-order differential equations. The method may be applied in various fields of engineering, in particular to dynamic response systems. We begin the derivation of Newmark's method by considering a system of p second-order differential equations:

$$\mathbf{M}\frac{d^2\mathbf{u}}{dt^2} + \mathbf{C}\frac{d\mathbf{u}}{dt} + \mathbf{K}\mathbf{u} = \mathbf{f}(t)$$

or using dashed notation:

$$\mathbf{M}\mathbf{u}'' + \mathbf{C}\mathbf{u}' + \mathbf{K}\mathbf{u} = \mathbf{f}(t) \tag{5.34}$$

together with the initial conditions that $\mathbf{u} = \mathbf{u}(0)$ and $\mathbf{u}' = \mathbf{u}'(0)$ at time $t = 0$. Here, if $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are arrays of constant coefficients and the function $\mathbf{f}(t)$ is a function of time only, then the equations are linear differential equations, otherwise they are non-linear differential equations. We will deal with linear differential equations case only. The initial development of Newmark's method uses the mean value theorem for u, one element of $\mathbf{u}$, and which may be stated as follows. There exists a scalar value β such that:

$$u(t + \Delta t) - u(t) = u'(t + \beta\Delta t)\Delta t$$

where the dash denotes the derivative of u and Δt is the time increment.

Newmark used the standard Taylor's series expansion of the functions $u(t + \Delta t)$ and $u'(t + \Delta t)$ as far as the third-order derivative of u. Denoting $u(t + \Delta t)$ by u_{n+1} and $u(t)$ by u_n, these are given by:

$$u_{n+1} = u_n + u_n'\Delta t + u_n''\frac{(\Delta t)^2}{2!} + u_n'''\frac{(\Delta t)^3}{3!} + ... \tag{5.35}$$

$$u_{n+1}' = u_n' + u_n''\Delta t + u_n'''\frac{(\Delta t)^2}{2!}... \tag{5.36}$$

Applying the generalized mean value theorem we may write these equations as:

$$u_{n+1} = u_n + u'_n \Delta t + u''_n \frac{(\Delta t)^2}{2!} + \beta u'''_n (\Delta t)^3 + \ldots \tag{5.37}$$

$$u'_{n+1} = u'_n + u''_n \Delta t + \gamma u'''_n (\Delta t)^2 + \ldots \tag{5.38}$$

where there exit $0 \le \gamma \le 1$ and $0 \le \beta \le 1$ such that these equations are true.

The third derivative u''', if linear, may be written:

$$u'''_{n+1} = (u''_{n+1} - u''_n)/\Delta t$$

So far we have considered the third derivative of a single variables u. However, if we are considering a system of p differential equations, then u must be generalized to a column vector of p variable, $\mathbf{u}$. Thus the previous equation becomes

$$\mathbf{u}'''_{n+1} = (\mathbf{u}''_{n+1} - \mathbf{u}''_n)/\Delta t$$

Substituting for each element of $\mathbf{u}'''$ in (5.38) in turn, and collecting the values of $\mathbf{u}'_{n+1}$ into a column vector, gives:

$$\mathbf{u}'_{n+1} = \mathbf{u}'_n + \Delta t(1-\gamma)\mathbf{u}''_n + \Delta t \gamma \mathbf{u}''_{n+1} \tag{5.39}$$

and $0 \le \gamma \le 1$.

Now, substituting for each element of $\mathbf{u}'''_{n+1}$ in (5.37) and collecting the values of $\mathbf{u}_{n+1}$ into a column vector gives

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{u}'_n + (1-2\beta)\frac{(\Delta t)^2}{2}\mathbf{u}''_n + \beta(\Delta t)^2 \mathbf{u}''_{n+1} \tag{5.40}$$

As above, we place a constraint on the values of β, $0 \le 2\beta \le 1$ and so $0 \le \beta \le 1/2$

We now have formulae for velocities, (5.39) and displacements (5.40). We can obtain a formula for accelerations from the original differential equation, (5.34), thus:

$$\mathbf{u}''_{n+1} = \mathbf{M}^{-1}(\mathbf{f}(t_{n+1}) - \mathbf{C}\mathbf{u}'_{n+1} - \mathbf{K}\mathbf{u}_{n+1}) \tag{5.41}$$

Thus the equations give formulae for displacement, velocity and acceleration. However since these are implicit they must be reformulated so that we obtain expressions for the updated values in terms of the previously known values at step n. This can be achieved by algebraic manipulation. Rearranging Eq. (5.40) to provide an expression for $\mathbf{u}''_{n+1}$ gives:

$$\beta(\Delta t)^2 \mathbf{u}''_{n+1} = \mathbf{u}_{n+1} - \mathbf{u}_n - \Delta t \mathbf{u}'_n - (1-2\beta)((\Delta t)^2/2)\mathbf{u}''_n$$

Now dividing this equation by $\beta(\Delta t)^2$ gives:

$$\mathbf{u}''_{n+1} = (\mathbf{u}_{n+1} - \mathbf{u}_n)/(\beta(\Delta t)^2) - \mathbf{u}'_n/(\beta \Delta t) - (1/(2\beta) - 1)\mathbf{u}''_n \tag{5.42}$$

Inserting the expression for $\mathbf{u}''_{n+1}$ from (5.42) into (5.39) gives:

$$\mathbf{u}'_{n+1} = \gamma(\mathbf{u}_{n+1} - \mathbf{u}_n)/(\beta \Delta t) + \mathbf{u}'_n(1-\gamma/\beta) + \Delta t \mathbf{u}''_n(1-\gamma/(2\beta)) \tag{5.43}$$

We can now insert the explicit expressions for $\mathbf{u}''_{n+1}$ and $\mathbf{u}'_{n+1}$ from (5.42) and (5.43) into the original second-order differential equation (5.34) to obtain:

$$\begin{aligned}&\mathbf{M}\left[(\mathbf{u}_{n+1}-\mathbf{u}_n)/(\beta(\Delta t)^2)-\mathbf{u}'_n/(\beta\Delta t)-(1/(2\beta)-1)\mathbf{u}''_n\right]\ldots\\&+\mathbf{C}\left[(\mathbf{u}_{n+1}-\mathbf{u}_n)\gamma/(\beta\Delta t)+\mathbf{u}'_n(1-\gamma/\beta)+\Delta t\mathbf{u}''_n(1-\gamma/(2\beta))\right]\ldots\\&+\mathbf{K}\mathbf{u}_{n+1}=\mathbf{f}(t_{n+1})\end{aligned} \tag{5.44}$$

Now rearranging (5.44) as an equation for u_{n+1} we obtain the equation:

$$\mathbf{A}\mathbf{u}_{n+1}=\mathbf{B}_n \tag{5.45}$$

and thus $\mathbf{u}_{n+1}=\mathbf{A}^{-1}\mathbf{B}_n$.

This is an *explicit* expression for $\mathbf{u}_{n+1}$ in terms of values at an earlier cycle. Where $\mathbf{A}$ and $\mathbf{B}_n$ are defined as follows:

$$\mathbf{A}=\frac{\mathbf{M}}{\beta(\Delta t)^2}+\frac{\gamma\mathbf{C}}{\beta\Delta t}+\mathbf{K} \tag{5.46}$$

$$\begin{aligned}\mathbf{B}_n=\mathbf{f}(t_{n+1})+\mathbf{M}&\left[\frac{\mathbf{u}_n}{\beta(\Delta t)^2}+\frac{\mathbf{u}'_n}{\beta\Delta t}+\left(\frac{1}{2\beta}-1\right)\mathbf{u}''_n\right]\ldots\\+\mathbf{C}&\left[\frac{\gamma\mathbf{u}_n}{\beta\Delta t}-\mathbf{u}'_n\left(1-\frac{\gamma}{\beta}\right)-\Delta t\mathbf{u}''_n\left(1-\frac{\gamma}{2\beta}\right)\right]\end{aligned} \tag{5.47}$$

Note that once β and Δt are given specific numerical values, $\mathbf{A}$ is an array of constant values for a particular set of differential equations. This, of course, is not the case for $\mathbf{B}_n$ which changes with each iteration. Having determined $\mathbf{u}_{n+1}$ we use Eqs. (5.42) and (5.43) to determine $\mathbf{u}''_{n+1}$ and $\mathbf{u}'_{n+1}$.

Knowing $\mathbf{u}''_{n+1}$ we can determine $\mathbf{u}'_{n+1}$ the iterative process can then be continued. We now proceed in this way for the number of iterations required.

It remains to choose values for γ and β. An undamped system (that is $\mathbf{C}=0$) is conditionally stable if $\gamma\geq 1/2$, and unconditionally stable if $\gamma\geq 1/2$ and $\beta\geq\frac{1}{4}\left(\gamma+\frac{1}{2}\right)^2$. However, $\gamma>1/2$ introduces artificial damping into the solution, and so Newmark chose $\gamma=1/2$ and $\beta=1/4$. This is sometimes called the average acceleration method and it is unconditionally stable, meaning that the method will converge for all time increments. If β is chosen to be $1/6$ the method is called the linear acceleration method, but it is only conditionally stable.

It is important to consider the stability of Newmark's method, i.e. the conditions under which the algorithm will converge to the correct solution. In certain circumstances Newmark's method is unconditionally stable, this means that the method will converge for all time increments.

This is the case if the following condition is satisfied:

$$2\beta\geq\gamma\geq 1/2 \tag{5.48}$$

For example $\beta=1/2$ and $\gamma=1/2$ satisfy this condition. But it is important to note that certain values of γ do introduce significant errors in the computation process. The method is called conditionally convergent when convergence is guaranteed only for a limited range of the step size. This condition is given by:

$$\Delta t \leq \frac{1}{\omega_{max}\sqrt{\frac{\gamma}{2} - \beta}} \tag{5.49}$$

where $\beta \leq 1/2$ and $\gamma \geq 1/2$.

The value ω_{max} can be calculated from the characteristic equation or the eigenvalue problem of the differential equations.

The following MATLAB function, `newmark`, implements this procedure.

```
function [tp,x] = newmark(M,C,K,F,tspan,n,x0,xd0)
% M, C, K are matrices multiplying xddot, xdot and x repectively
% F is column vector of exciations. x0, xd0 are initial x0 and xd vectors
% tspan = [t_initial t_final]; n is tspan/t_increment
dt = (tspan(2)-tspan(1))/n;
tp(1) = tspan(1);
x(:,1) = x0';
xd(:,1) = xd0';
gamma = 1/2;   beta = 1/4;
A = (1/(beta*dt^2))*M+(gamma/(beta*dt))*C+K; invA = inv(A);
xdd(:,1) = inv(M)*(F(:,1)-C*xd(:,1)-K*x(:,1));
for i = 1:n
    B = (F(:,i+1)+...
        M*((1/(beta*dt^2))*x(:,i)+(1/(beta*dt))*xd(:,i)+ ...
(1/(2*beta)-1)*xdd(:,i))+C*((gamma/(beta*dt))*x(:,i)+...
(gamma/beta-1)*xd(:,i)+(gamma/beta-2)*(dt/2)*xdd(:,i)));
    x(:,i+1) = invA*B;
    xdd(:,i+1) = (1/(beta*dt^2))*(x(:,i+1)-x(:,i))...
-(1/(beta*dt))*xd(:,i)-((1/(2*beta))-1)*xdd(:,i);
    xd(:,i+1) = xd(:,i)+(1-gamma)*dt*xdd(:,i)+gamma*dt*xdd(:,i+1);
    tp(i+1) = tp(i)+dt;
end
x = x';
```

We now illustrate the application of Newmark's method to solving the system shown in Fig. 2.8. The equations of motion for this system are given in (2.35) and these equations can be written in matrix form as

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{0}$$

For the purposes of this illustration, we will add an external force to the system given by $\mathbf{f}(t)$ and a damping matrix, $\mathbf{C}$, so that the above equation becomes

$$\mathbf{M}\ddot{\mathbf{x}} + \mathbf{C}\dot{\mathbf{x}} + \mathbf{K}\mathbf{x} = \mathbf{f}(t)$$

M and **K** are given by (2.38) and repeated here for convenience.

$$\mathbf{M} = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 30 \end{bmatrix} \text{ kg and } \mathbf{K} = \begin{bmatrix} 45 & -20 & -15 \\ -20 & 45 & -25 \\ -15 & -25 & 40 \end{bmatrix} \text{ kN/m}$$

x lists the displacements of the three coordinates. We will make $\mathbf{C} = 3 \times 10^{-2}\mathbf{K}$ and apply a half sine pulse of 0.29 s duration with a peak value of 50 N at coordinate x_3. The MATLAB script `e4s506.m` calls both `newmark` and `ode45` to solve this problem.

```
%e4s506.m
clear all
t_init = 0;   t_final = 3;
t_incr = 0.005;
n = (t_final-t_init)/t_incr;
tt = t_init:t_incr:t_final;
trange = [t_init t_final];
M = [10 0 0;0 20 0;0 0 30];
K = 1e3*[45 -20 -15;-20 45 -25;-15 -25 40];
C = 3e-2*K;
F(1,:) = 0*tt;
F(2,:) = 0*tt;
omega = pi/.29;
F(3,:) = 50*sin(omega*tt);
for j = 1:n
    pulse(j) = sin(omega*tt(j));
    if tt(j) > pi/omega
        pulse(j) = 0;
    end
end
F(3,1:n) = 50*pulse;
x0 = [0 0 0];
xd0= [0 0 0];
[tp,x1] = newmark(M,C,K,F,trange,n,x0,xd0);
x1 = 1000*x1;
figure(1), plot(tp,x1(:,1),'k',tp,x1(:,2),'k',tp,x1(:,3),'k',tp,0*tp,'k')
xlabel('Time, s')
ylabel('x, mm')
axis([0 3 -5 10])
hold on
figure(1), plot(tt(1:60),5*pulse(1:60),'.k')
hold off
[t,x2] = ode45('f10',[t_init:t_incr:t_final],[0 0 0 0 0 0]);
x2 = 1000*x2;
d(:,1) = (x1(:,1)-x2(:,1));
```

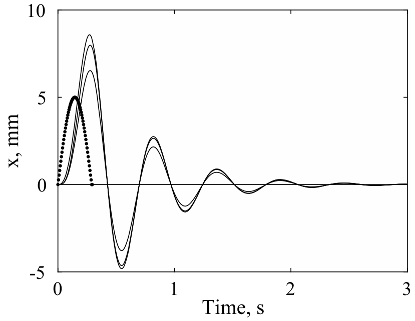

FIGURE 5.14

Graph showing the three coordinate responses of a mass-spring-damper system, shown by full lines, when excited by a half sine pulse, shown by a dotted line.

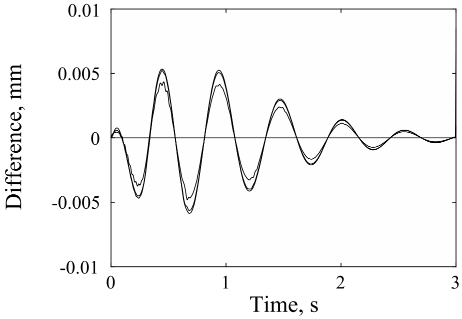

FIGURE 5.15

Plot showing the difference between the Newmark and 4th-order Runge–Kutta method solutions for the three coordinates.

```
d(:,2) = (x1(:,2)-x2(:,2));
d(:,3) = (x1(:,3)-x2(:,3));
figure(2), plot(tp,d(:,1),'k',tp,d(:,2),'k',tp,d(:,3),'k',tp,0*tp,'k')
xlabel('Time, s')
ylabel('Difference, mm')
axis([0 3 -1e-2 1e-2])
```

The function `f10`, required by function `ode45`, is as follows:

```
function yprime = f10(t,y)
m = [10 0 0;0 20 0;0 0 30];
k = 1e3*[45 -20 -15;-20 45 -25;-15 -25 40];
c = 3e-2*k;  f =[0; 0; 50];
omega = pi/.29;
pulse = sin(omega*t);
if t > pi/omega
    pulse = 0;
end
A = [zeros(3,3) eye(3,3); -m\k -m\c];
b = [zeros(3,1); m\f*pulse];
yprime = A*y+b;
```

Running script `e4s506.m` generates Figs. 5.14 and 5.15. Fig. 5.14 shows the response of the three coordinates and the half sine pulse. The pulse is scaled a factor of 10 in order to display on the graph. Fig. 5.15 shows the difference between the output of `newmark` and `ode45`. It is seen that the maximum difference is 0.006 mm.

5.12 CHAOTIC SYSTEMS

Chaos arises in several disciplines, including meteorology, physics, environmental science, engineering, economics, biology and ecology. Because of this it is perhaps surprising that chaotic motion was not properly recognized until the 1960s.

Chaotic systems are difficult to define. Essentially they are dynamic systems that are highly sensitive to small differences in their initial conditions and also to rounding errors in numerical computation. These differences produce widely diverging outcomes for such systems. In 1972 Edward Lorenz presented a talk to the American Association for the Advancement of Science entitled “Predictability: Does the Flap of a Butterfly’s Wings in Brazil Set a Tornado in Texas?”. The theme of this presentation was that small changes in the state of the atmosphere can result in large differences in its later states. The term *butterfly effect* has entered popular culture as a symbol of chaos.

Chaos occurs even though such systems are deterministic, meaning that their future behavior is fully determined by their initial conditions, with no random elements involved. Given *exactly* the same initial conditions, the same result is obtained. Thus, despite its “random” appearance, chaos is a deterministic outcome.

A simple example of a chaotic system quoted by Lorenz (1993) is a pin ball machine. At the start of the game, the ball is given an initial velocity. The ball then rolls through an array of pins. Striking a pin changes the ball’s direction and the ball subsequently strikes other pins in the array. The slightest change in the initial velocity of the ball will cause it to strike different pins and take a totally different route through the array of pins.

5.12.1 THE LORENZ EQUATIONS

As an example of a system of three simultaneous equations, we consider the Lorenz system. This system has a number of important applications including weather forecasting. The system has the form

$$dx/dt = s\,(y - x)$$

$$dy/dt = rx - y - xz$$

$$dz/dt = xy - bz$$

subject to appropriate initial conditions. As the parameters s, r, and b are varied through various ranges of values, the solutions of this system of differential equations vary in form. In particular, for certain values of the parameters the system exhibits chaotic behavior. To provide more accuracy in the computation process we use the MATLAB function `ode45` rather than `ode23`. The MATLAB script `e4s507.m` for solving this problem is given below.

```
% e4s507.m  Solution of the Lorenz equations
r = input('enter a value for the constant r ');
simtime = input('enter runtime ');
acc = input('enter accuracy value ');
xprime = @(t,x) [10*(x(2)-x(1)); r*x(1)-x(2)-x(1)*x(3); ...
             x(1)*x(2)-8*x(3)/3];
```

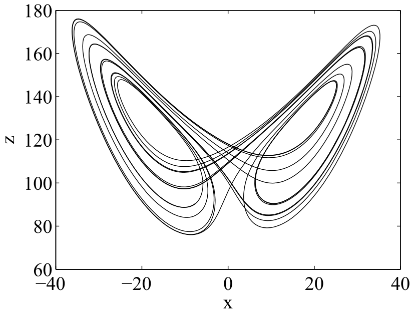

FIGURE 5.16

Solution of Lorenz equations for $r = 126.52$, $s = 10$, and $b = 8/3$ using an accuracy of 0.000005 and terminating at $t = 8$.

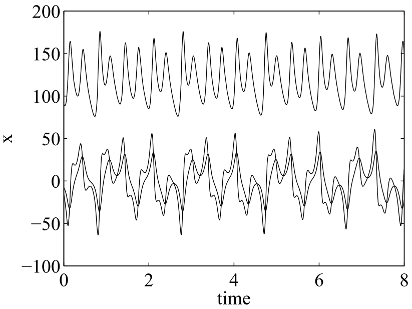

FIGURE 5.17

Solution of Lorenz equations where each variable is plotted against time. Conditions are the same as those used to generate Fig. 5.16. Note the unpredictable nature of the solutions.

```
initx = [-7.69 -15.61 90.39]';
tspan = [0 simtime];
options = odeset('RelTol',acc);
[t x] = ode45(xprime,tspan,initx,options);
% Plot results against time
figure(1), plot(t,x,'k')
xlabel('Time'), ylabel('x')
figure(2), plot(x(:,1),x(:,3),'k')
xlabel('x'), ylabel('z')
```

The results of running script `e4s507.m` are given in Figs. 5.16 and 5.17. Fig. 5.16 is characteristic of the Lorenz equations and shows the complexity of the relationship between x and z. Fig. 5.17 shows how x, y, and z change with time.

For $r = 126.52$ and for other large values of r the behavior of this system is chaotic. In fact for $r > 24.7$ most orbits exhibit chaotic wandering. The trajectory passes around two points of attraction, called *strange attractors*, switching from one to another in an apparently unpredictable fashion. This appearance of apparently random behavior is remarkable considering the clearly deterministic nature of the problem. However, for other values of r the behavior of the trajectories is simple and stable.

We now consider the case of the Lorenz equations with $s = 10$, $b = 8/3$, and $r = 28$. Fig. 5.18 shows the solution of the Lorenz equations with initial conditions of $\mathbf{x} = [5\ 5\ 5]$ and $\mathbf{x} = [5.0091\ 4.9997\ 5.0060]$. Fig. 5.19 shows the solution of Lorenz equations with the initial conditions $\mathbf{x} = [5\ 5\ 5]$ but with two different accuracies; the default accuracy of the MATLAB function `ode45`; namely an absolute tolerance of 10^{-6} and a relative tolerance of 10^{-3}, and an increased accuracy with an absolute tolerance of 10^{-10} and a relative tolerance of 10^{-8}.

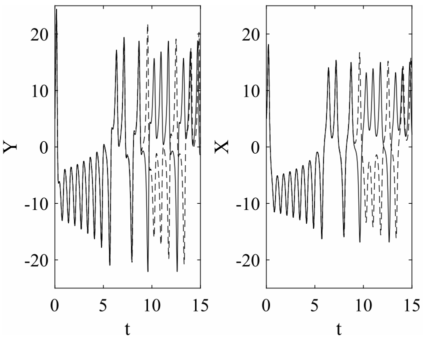

FIGURE 5.18

Solution of Lorenz equations for $r = 28$, $s = 10$, and $b = 8/3$. Initial conditions $\mathbf{x} = [5\ 5\ 5]$ shown by the full line, and $\mathbf{x} = [5.0091\ 4.9997\ 5.0060]$ shown by the dashed line. Note the sudden divergence of the two solutions from each other and unpredictable nature of the solutions.

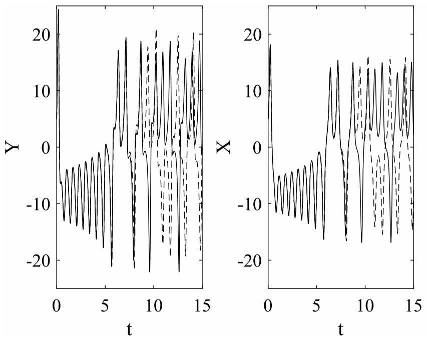

FIGURE 5.19

Solution of Lorenz equations for $r = 28$, $s = 10$, and $b = 8/3$. The full line shows the solution using the default accuracy of the MATLAB Runge–Kutta 4/5 function. The dashed line shows a higher accuracy solution. Note the sudden divergence of the two solutions from each other and unpredictable nature of the solutions.

5.12.2 DUFFING'S EQUATION

The Duffing oscillator is described by Duffing's equation: a second-order differential equation of the form

$$m\ddot{x} + c\dot{x} + kx + hx^3 = f(t)$$

This equation might represent, for example, an electric circuit or a mechanical oscillator. In a mechanical oscillator, m is the system mass, c is the viscous damping coefficient, k and h are stiffness coefficients and $f(t)$ is the force applied to the system. This force is a function of time, t. Duffing's equation is, of course, a non-linear differential equation since it contains a term in x^3. The equation can be solved approximately by seeking a series solution and considering only a small number of terms in the series. These approximate solutions can be very revealing and provide information about the free vibration of the oscillator by setting $f(t)$ to zero, or the system response to a non-zero force $f(t)$. Here we seek a numerical solution using the MATLAB function `ode45`. In order to use `ode45` we must transform the second-order differential equation into a pair of first-order differential equations thus

$$m\dot{u} = f(t) - cu - kx - hx^3 \text{ and } \dot{x} = u$$

We consider two cases. In each case we assume $m = 1$ kg, $c = 5$ Ns/m, $k = -5000$ N/m, $h = 50 \times 10^6$ N/m^3, and $f = 50\sin(\omega t)$ N. In each case only the frequency of the applied force and the initial conditions are changed. The following script `e4s508.m` solves Duffing's equation and generates Figs. 5.20–5.25.

```
% e4s508.m
clear all
close all
% Solution of Duffing's equation
% x_ddot+c*x_dot+k*x+h*x^3 = f*cos(omega*t)
c = 10; k = -5000; h = 50e6; f = 30;
% Case 1 omega = 100
omega = 100;
omega_Hz = omega/2/pi;
qprime = @(t,q)[q(2); -c*q(2)-k*q(1)-h*q(1)^3+f*cos(omega*t)];
dt = 0.001; nt = 8192; T = nt*dt;
t = 0:dt:(nt-1)*dt;
[t q] = ode45(qprime,[0:dt:(nt-1)*dt],[0 0]);
figure(1), plot(t,1000*q(:,1),'k',t,5*cos(omega*t),'k--')
axis([2 3 -25 25])
xlabel('Time, s')
ylabel('Displacement, mm')
grid
% Case 2 omega = 120
omega = 120;
omega_Hz = omega/2/pi;
qprime = @(t,q)[q(2); -c*q(2)-k*q(1)-h*q(1)^3+f*cos(omega*t)];
[t q11] = ode45(qprime,[0:0.001:1],[0 0]);
[t q12] = ode45(qprime,[0:0.001:1],[0.001 1]);
figure(2), plot(t,1000*q11(:,1),'k',t,1000*q12(:,1),'k--')
grid
xlabel('Time, s')
ylabel('Displacement, mm')
[t4 q4] = ode45(qprime,[0:0.001:2],[0.001000 0]);
[t5 q5] = ode45(qprime,[0:0.001:2],[0.001001 0]);
[t6 q6] = ode45(qprime,[0:0.001:2],[0.001002 0]);
figure(3), plot(t4,1000*q4(:,1),'k',t5,1000*q5(:,1),'k--',t6,1000*q6(:,1),'k-.')
grid
axis([0 2 -20 20])
xlabel('Time, s')
ylabel('Displacement, mm')
[t q] = ode45(qprime,[15:0.001:20],[0 0]);
x = 1000*q(:,1); v = q(:,2); theta = omega*t;
figure(4); plot(x,v,'k')
grid
xlabel('Displacement, mm')
ylabel('Velocity, m/s')
[t1 q1] = ode45(qprime,[0:2*pi/omega:400],[0 0]);
figure(5), plot(1000*q1(300:end,1),q1(300:end,2),'k.')
```

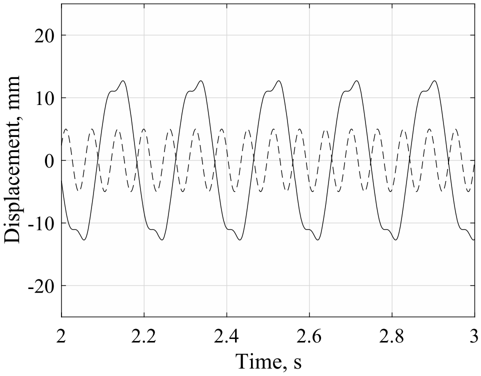

FIGURE 5.20

Case 1: The full line is the output from Duffing oscillator. $\omega = 100$ rad/s (15.92 Hz). Zero initial conditions. The dashed line is the input force, arbitrarily scaled in amplitude.

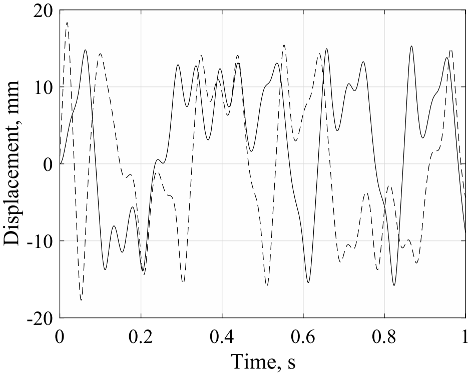

FIGURE 5.21

Output from Duffing oscillator. $\omega = 120$ rad/s. Full line gives output with zero initial conditions. Dashed line give output with an initial displacement of 1 mm and an initial velocity of 1 m/s.

```
axis([-20 20 -1.1 1.1])
xlabel('Displacement, mm')
ylabel('Velocity, m/s')
[t1 q1] = ode45(qprime,[0:2*pi/omega:400],[0 0]);
[t2 q2] = ode45(qprime,[0:2*pi/omega:400],[0.001 1]);
figure(6)
plot(1000*q2(775:25:2000,1),q2(775:25:2000,2),'ko',1000*q1(775:25:2000,1),...
q1(775:25:2000,2),'k+')
axis([-20 20 -1.1 1.1])
xlabel('Displacement, mm')
ylabel('Velocity, m/s')
```

Case 1. Input frequency $\omega = 100$ rad/s (15.92 Hz). With the values of the two initial conditions equal to zero, the oscillator output is shown in Fig. 5.20. Also shown in the figure is the input force, its amplitude arbitrarily scaled to fit the graph. It can be seen that the output frequency is about one third of the input frequency. This can be confirmed by using the discrete Fourier transform (see Chapter 8). The Fourier transform shows the major frequency component in the output is $\omega/3$ and there are also smaller frequency components at ω and $5\omega/3$. However, introducing an initial velocity of 1 m/s causes the $\omega/3$ and $5\omega/3$ frequency components in the output to vanish so that the ω frequency component is dominant. There is then also a small 2ω frequency component present.

Case 2. Input frequency $\omega = 120$ rad/s (19.10 Hz). At this excitation frequency the character of the response has changed completely, as shown in Fig. 5.21. The output appears to be unpredictable. Furthermore, the output with initial conditions of velocity and displacement equal to zero, is strikingly different from the output with an initial displacement of 1 mm and an initial velocity of 1 m/s except, perhaps, in the overall amplitude. Even more remarkable are the three plots shown in Fig. 5.22. In each case the initial velocity is zero but the three initial displacements are 1 mm, 1.001 mm and

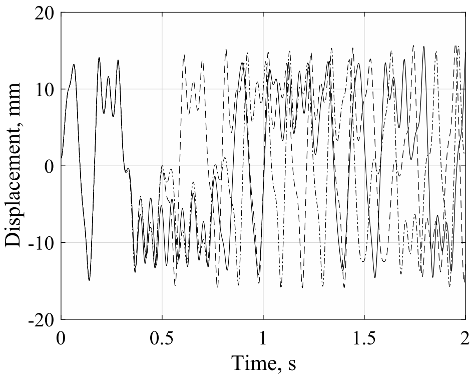

FIGURE 5.22

Output from Duffing oscillator. $\omega = 120$ rad/s. Solution with zero initial velocity and initial displacements of 1, 1.001, and 1.002 mm. (Shown by full, dashed and dot-dashed lines respectively.)

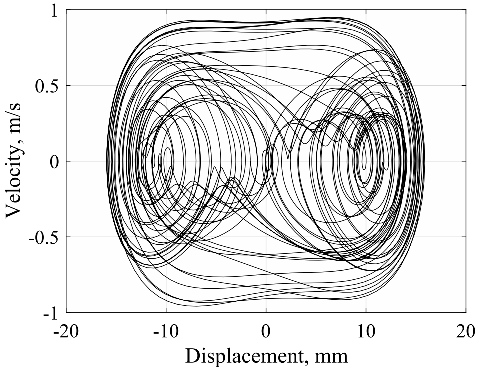

FIGURE 5.23

Output from Duffing oscillator. Phase plane plot. $\omega = 120$ rad/s.

1.002 mm. The outputs differ from each other even though the difference in the initial displacements are only 1 and 2 μm. Clearly, this is an example of chaotic motion. If we plot the output displacement against the output velocity for a short period, Fig. 5.22 is obtained. It is difficult to draw any useful conclusions from this plot, and so we plot the so-called Poincaré map or section to clarify the situation. The output displacement and velocity are sampled once per cycle of the excitation or input signal frequency, for a large number of cycles and the resulting large number of sample points is plotted in the displacement–velocity plane. Thus if $T = 2\pi/\omega$ then the samples are taken at multiples of T, that is $T_s = 0, T, 2T, ..., nT$. The Poincaré map for this system is shown in Fig. 5.23. We see that there are areas where points cluster and other areas devoid of values. If we change the initial displacement from zero to 1 mm and the initial velocity from zero to 1 m/s the Poincaré map appears unchanged. How can this be since the output is so influenced by the initial condition, as shown in Fig. 5.21? The explanation is that if we plot a small number of the output points at the same instances of time but with the two different sets of initial conditions, the points on the Poincaré map are in different locations but, none the less, they are confined in the same regions. This is shown in Fig. 5.24. A large number of points from either output will create the same Poincaré map but the points generated from the two different sets of initial conditions will not be coincident with each other in the Poincaré map. The Poincaré map helps to clarify the relationship between displacement and velocity.

5.13 DIFFERENTIAL EQUATIONS APPLIED TO NEURAL NETWORKS

Different types of neural networks have been used to solve a wide range of problems. Neural networks often consist of several layers of “neurons” that are “trained” by fixing a set of weights. These weights are found by minimizing the sum of squares of the difference between actual and required outputs. Once trained, the networks can be used to classify a range of inputs. However, here we consider

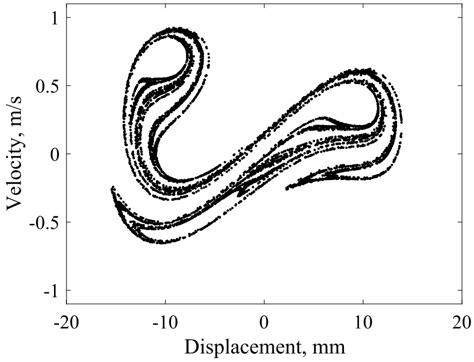

FIGURE 5.24

Poincaré map showing output from Duffing oscillator. $\omega = 120$ rad/s.

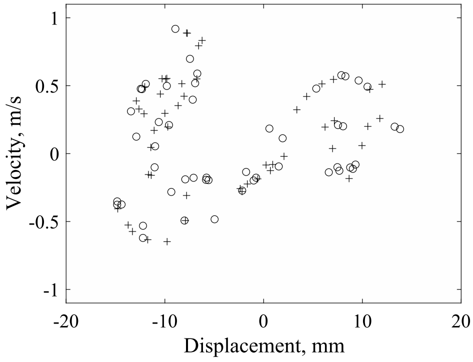

FIGURE 5.25

Output from Duffing oscillator showing where points from two solutions lie on a Poincaré map. + and o indicate points generated from two different initial conditions with $\omega = 120$ rad/s.

a different approach that uses a neural network that may be based directly on considering a system of differential equations. This approach is described by Hopfield and Tank (1985, 1986), who demonstrated the application of neural networks to solving specific numerical problems. It is not our intention to provide the full details or proofs of this process here.

Hopfield and Tank, in their 1985 and 1986 papers, utilized a system of differential equations which take the form

$$\frac{du_i}{dt} = \frac{-u_i}{\tau} + \sum_{j=0}^{n-1} T_{ij} V_j + I_i \quad \text{for } i = 0, 1, \ldots n - 1 \tag{5.50}$$

where τ a constant usually taken as 1. This system of differential equations represents the interaction of a system of n neurons, and each differential equation is a simple model of a single biological neuron. (This is only one of a number of possible models of a neural network.) Clearly, to establish a network of such neurons, they must be able to interact with each other and this interaction must be represented in the differential equations. The T_{ij} provide the strengths of the interconnections between the ith and jth neuron and the I_i provide the externally applied current to the ith neuron. These I_i may be viewed as inputs to the system. The V_j values provide the outputs from the system and are directly related to the u_j so that we may write $V_j = g(u_j)$. The function g, called a sigmoidal function, may be specified, for example, by

$$V_j = (1 + \tanh u_j)/2 \text{ for all } j = 0, 1, \ldots, n - 1$$

A plot of this function is given in Fig. 5.26.

Having provided such a model of a neural network, the question still remains: how can we show that it can be used to solve specific problems? This is the key issue and a significant problem in itself. Before we can solve a given problem using a neural network we must first reformulate our problem so that it can be solved by this approach.

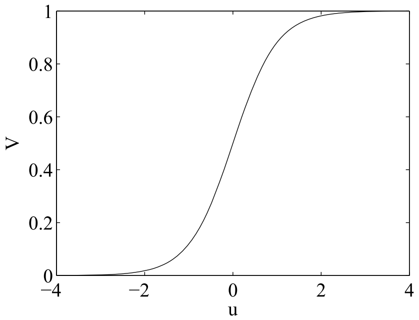

FIGURE 5.26

Plot of sigmoid function $V = (1 + \tanh u)/2$.

To illustrate this process, Hopfield and Tank chose as an example the simple problem of binary conversion. That is, to find the binary equivalent of a given decimal number. Since there is no obvious and direct relationship between this problem and the system of differential equations (5.50) which model the neural network, a more direct link has to be established.

Hopfield and Tank have shown that the stable state solution of (5.50), in terms of the V_j, is given by the minima of the energy function:

$$E = -\frac{1}{2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} T_{ij} V_i V_j - \sum_{j=0}^{n-1} I_j V_j \tag{5.51}$$

It is an easy matter to link the solution of the binary conversion problem to the minimization of the function (5.51).

Hopfield and Tank consider the energy function

$$E = \frac{1}{2}\left\{x - \sum_{j=0}^{n-1} V_j 2^j\right\}^2 + \sum_{j=0}^{n-1} 2^{2j-1} V_j \left(1 - V_j\right) \tag{5.52}$$

Now the minimum of (5.52) will be attained when $x = \Sigma V_j 2^j$ and $V_j = 0$ or 1. Clearly the first term ensures that the required binary representation is achieved while the second term provides that the V_j take either 0 or 1 values when the value of E is minimized. On expanding this energy function (5.52) and comparing it with the general energy function (5.51) we find that if we make

$$T_{ij} = -2^{i+j} \text{ for } i \neq j \text{ and } T_{ij} = 0 \text{ when } i = j$$
$$I_j = -2^{2j-1} + 2^j x$$

then the two energy functions are equivalent, apart from a constant. Thus the minimum of one gives the minimum of the other. Solving the binary conversion problem expressed in this way is thus equivalent to solving the system of differential equations (5.50) with this special choice of values for T_{ij} and I_i.

In fact, using an appropriate choice of T_{ij} and I_i, a range of problems can be represented by a neural network in the form of the system of differential equations (5.50). Hopfield and Tank have extended this process from the simple example considered above to attempting to solve the very challenging traveling salesman problem. The details of this are given in Hopfield and Tank (1985, 1986).

In MATLAB we may use `ode23` or `ode45` to solve this problem. The crucial part of this exercise is to define the function which gives the right-hand sides of the differential equation system for the neural network. This can be done very simply using the function `hopbin` below. This function gives the right-hand side for the differential equations which solve the binary conversion problem. In the definition of function `hopbin`, `sc` is the decimal value we wish to convert.

```
function neurf = hopbin(t,x)
global n sc
% Calculate synaptic current
I = 2.^[0:n-1]*sc-0.5*2.^(2.*[0:n-1]);
% Perform sigmoid transformation
V = (tanh(x/0.02)+1)/2;
% Compute interconnection values
p = 2.^[0:n-1].*V';
% Calculate change for each neuron
neurf = -x-2.^[0:n-1]'*sum(p)+I'+2.^(2.*[0:n-1])'.*V;
```

This function `hopbin` is called by the script `e4s509.m` to solve the system of differential equations which define the neural network and hence simulate its operation.

```
% e4s509.m
% Hopfield and Tank neuron model for binary conversion problem
global n sc
n = input('enter number of neurons ');
sc = input('enter number to be converted to binary form ');
simtime = 0.2; acc = 0.005;
initx = zeros(1,n)';
options = odeset('RelTol',acc);
%Call ode45 to solve equation
[t x] = ode45('hopbin',[0 simtime],initx,options);
V = (tanh(x/0.02)+1)/2;
bin = V(end,n:-1:1);
for i = 1:n
    fprintf('%8.4f', bin(i))
end
fprintf('\n\n')
plot(t,V,'k')
xlabel('Time'), ylabel('Binary values')
```

Running this script to convert the decimal number 5 gives

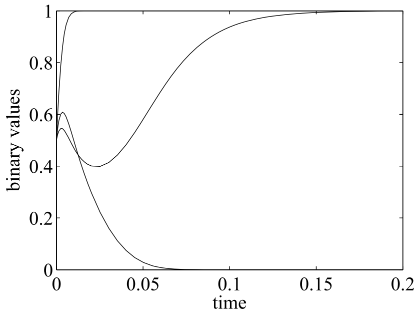

FIGURE 5.27

Neural network finds the binary equivalent of 5 using 3 neurons and an accuracy of 0.005. The three curves show the convergence to the binary digits 1, 0, and 1.

```
enter number of neurons 3
enter number to be converted to binary form 5
  1.0000  0.0000  0.9993
```

together with Fig. 5.27. This plot shows how the neural network model converges to the required results, that is, `V(1) = 1`, `V(2) = 0` and `V(3) = 1` or binary number 101. As a further example, we convert the decimal number 59 using 7 neurons as follows:

```
enter number of neurons 7
enter number to be converted to binary form 59
  0.0000  1.0000  1.0000  1.0000  0.0000  1.0000  0.9999
```

Again, a correct result.

This is an application of neural networks to a trivial problem. A real test for neural computing is the traveling salesman problem. The MATLAB neural network toolbox provides a range of functions to solve neural network problems.

5.14 STIFF EQUATIONS

When the solution of a system of differential equations contains components which change at significantly different rates for given changes in the independent variable, the equation system is said to be “stiff”. When this phenomenon is present, a particularly careful choice of the step size must be made if stability is to be achieved.

We will now consider how the stiffness phenomenon arises in an apparently simple system of differential equations. Consider the system given below.

$$\begin{aligned} dy_1/dt &= -by_1 - cy_2 \\ dy_2/dt &= y_1 \end{aligned} \tag{5.53}$$

This system may be written in matrix form as

$$d\mathbf{y}/dt = \mathbf{A}\mathbf{y} \tag{5.54}$$

The solution of (5.54) is

$$\begin{aligned} y_1 &= A\exp(r_1t) + B\exp(r_2t) \\ y_2 &= C\exp(r_1t) + D\exp(r_2t) \end{aligned} \tag{5.55}$$

where A, B, C, and D are constants set by the initial conditions. It can easily be verified that r_1 and r_2 are the eigenvalues of the matrix $\mathbf{A}$.

If a numerical procedure is applied to solve these systems of differential equations, the success of the method will depend crucially on the eigenvalues of the matrix $\mathbf{A}$ and in particular the ratio of the smallest and largest eigenvalues.

As an example of a matrix with widely spaced eigenvalues we can take the 8 × 8 Rosser matrix; this is available in MATLAB as `rosser`. The sequence of statements

```
>> a = rosser; lambda = eig(a);
>> eigratio = max(abs(lambda))/min(abs(lambda))

eigratio =
  3.2285e+15
```

produces a matrix with eigenvalue ratios of order 10^{16}. Thus a system of ordinary first-order differential equations involving this matrix would be pathologically difficult to solve. The significance of the eigenvalue ratio in relation to the required step size can be generalized to systems of many equations. Consider the system of n equations

$$d\mathbf{y}/dt = \mathbf{A}\mathbf{y} + \mathbf{P}(t) \tag{5.56}$$

where $\mathbf{y}$ is an n component column vector, $\mathbf{P}(t)$ is an n component column vector of functions of t and $\mathbf{A}$ is an $n \times n$ matrix of constants. It can be shown that the solution of this system takes the form

$$\mathbf{y}(t) = \sum_{i=1}^{n} v_i \, \mathbf{d}_i \, \exp(r_i t) + \mathbf{s}(t) \tag{5.57}$$

Here r_1, r_2, ... are the eigenvalues and $\mathbf{d}_1$, $\mathbf{d}_2$, ... the eigenvectors of $\mathbf{A}$. The vector function $\mathbf{s}(t)$ is the particular integral of the system, sometimes called the steady-state solution since for negative eigenvalues the exponential terms should die away with increasing t. If it is assumed that the $r_k < 0$ for $k = 1, 2, 3, ...$ and we require the steady-state solution of system (5.56), then any numerical method applied to solve this problem may face significant difficulties, as we have seen. We must continue the integration until the exponential components have been reduced to negligible levels and yet we must take sufficiently small steps to ensure stability, thus requiring many steps over a large interval. This is the most significant effect of stiffness.

The definition of stiffness can be extended to any system of the form (5.56). The stiffness ratio is defined as the ratio of the largest and smallest eigenvalues of $\mathbf{A}$ and gives a measure of the stiffness of the system.

The methods used to solve stiff problems must be based on stable techniques. The MATLAB function `ode23s` uses continuous step size adjustment and therefore is able to deal with such problems, although the solution process may be slow. If we use a predictor–corrector method, not only must this method be stable but the corrector must also be iterated to convergence. An interesting discussion of this topic is given by Ralston and Rabinowitz (1978). Specialized methods have been developed for solving stiff problems, and Gear (1971) has provided a number of techniques which have been reported to be successful. Note that although explicit methods are commonly used for stiff equations, implicit methods can be a good choice for this type of problem.

5.15 SPECIAL TECHNIQUES

A further set of predictor–corrector equations may be generated by making use of an interpolation formula due to Hermite. An unusual feature of these equations is that they contain second-order derivatives. It is usually the case that the calculation of second-order derivatives is not particularly difficult and consequently this feature does not add a significant amount of work to the solution of the problem. However, it should be noted that in using a computer program for this technique the user has to supply not only the function on the right-hand side of the differential equation but its derivative as well. To the general user this may be unacceptable.

The equations for Hermite's method take the form

$$\begin{aligned} y_{n+1}^{(1)} &= y_n + h(y_n' - 3y_{n-1}')/2 + h^2(17y_n'' + 7y_{n-1}'')/12 \\ y_{n+1}^{*\,(1)} &= y_{n+1}^{(1)} + 31(y_n - y_n^{(1)})/30 \\ y_{n+1}^{\prime\,(1)} &= f(t_{n+1}, y_{n+1}^{*\,(1)}) \end{aligned} \tag{5.58}$$

For $k = 1, 2, 3, \ldots$

$$y_{n+1}^{(k+1)} = y_n + h(y_{n+1}^{\prime\,(k)} + y_n')/2 + h^2(-y_{n+1}^{\prime\prime\,(k)} + y_n'')/12$$

This method is stable and has a smaller truncation error at each step than Hamming's method. Thus it may be worthwhile accepting the additional effort required by the user. We note that since we have

$$dy/dt = f(t, y)$$

then

$$d^2y/dt^2 = df/dt$$

and thus y_n'', etc., are easily calculated as the first derivative of f. The MATLAB function `fhermite` implements this method, and the script is given below. Note that in this function, the function `f` must provide both the first and second derivatives of y.

```
function [tvals, yvals] = fhermite(f,tspan,startval,step)
% Hermite's method for solving
% first order differential equation dy/dt = f(t,y).
% Example call: [tvals, yvals] = fhermite(f,tspan,startval,step)
% The initial and final values of t are given by tspan = [start finish].
% Initial value of y is given by startval, step size is given by step.
% The function f(t,y) and its derivative must be defined by the user.
% 3 steps of Runge-Kutta are required so that hermite can start.
% Set up matrices for Runge-Kutta methods
b = [ ]; c = [ ]; d = [ ];
order = 4;
b = [1/6 1/3 1/3 1/6]; d = [0 0.5 0.5 1];
c = [0 0 0 0;0.5 0 0 0;0 0.5 0 0;0 0 1 0];
steps = (tspan(2)-tspan(1))/step+1;
y = startval; t = tspan(1);
ys(1) = startval; w = feval(f,t,y); fval(1) = w(1); df(1) = w(2);
yvals = startval; tvals = tspan(1);
for j = 2:2
    k(1) = step*fval(1);
    for i = 2:order
        w = feval(f,t+step*d(i),y+c(i,1:i-1)*k(1:i-1)');
        k(i) = step*w(1);
    end
    y1 = y+b*k'; ys(j) = y1; t1 = t+step;
    w = feval(f,t1,y1); fval(j) = w(1); df(j) = w(2);
    %collect values together for output
    tvals = [tvals, t1]; yvals = [yvals, y1];
    t = t1; y = y1;
end
%hermite now applied
h2 = step*step/12; er = 1;
for i = 3:steps
    y1 = ys(2)+step*(3*fval(1)-fval(2))/2+h2*(17*df(2)+7*df(1));
    t1 = t+step; y1m = y1; y10 = y1;
    if i>3, y1m = y1+31*(ys(2)-y10)/30; end
    w = feval(f,t1,y1m); fval(3) = w(1); df(3)=w(2);
    yc = 0; er = 1;
    while abs(er)>0.0000001
        yp = ys(2)+step*(fval(2)+fval(3))/2+h2*(df(2)-df(3));
        w = feval(f,t1,yp); fval(3) = w(1); df(3) = w(2);
        er = yp-yc; yc = yp;
    end
    fval(1:2) = fval(2:3); df(1:2) = df(2:3);
    ys(2) = yp;
    tvals = [tvals, t1]; yvals = [yvals, yp];
    t = t1;
end
```

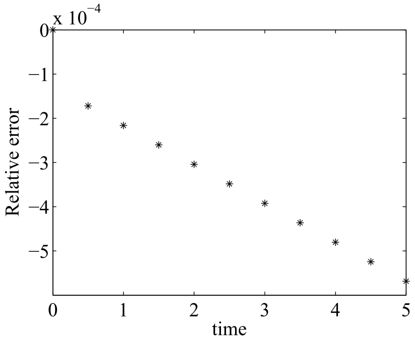

FIGURE 5.28

Relative error in the solution of $dy/dt = y$ using Hermite's method. Initial condition $y = 1$ when $t = 0$ and a step of 0.5.

Fig. 5.28 gives the error when solving the specific equation $dy/dt = y$ using the same step size and starting point as for Hamming's method – see Fig. 5.9. For this particular problem Hermite's method performs better than Hamming's method.

Finally, we compare the Hermite, Hamming, and Adams–Bashforth–Moulton methods for the difficult problem

$$dy/dt = -10y \text{ given } y = 1 \text{ when } t = 0$$

The script `e4s510.m` implements these comparisons.

```
% e4s510.m
vg = @(t,x) [-10*x 100*x];
v = @(t,x) -10*x;
disp('Solution of dx/dt = -10x')
t0 = 0; y0 = 1;
tf = 1; tinc = 0.1; steps = floor((tf-t0)/tinc+1);
[t,x1] = abm(v,[t0 tf],y0,tinc);
[t,x2] = fhamming(v,[t0 tf],y0,tinc);
[t,x3] = fhermite(vg,[t0 tf],y0,tinc);
disp('t         abm         Hamming     Hermite     Exact');
for i = 1:steps
    fprintf('%4.2f%12.7f%12.7f',t(i),x1(i),x2(i))
    fprintf('%12.7f%12.7f\n',x3(i),exp(-10*(t(i))))
end
```

Note that for the function `fhermite`, we must supply both the first and second derivatives of y with respect to t. For the first derivative, we have directly $dy/dt = -10y$ but the second derivative d^2y/dt^2 is given by $-10dy/dt = -10(-10y) = 100y$. Consequently, the function takes the form

```
vg = @(t,x) [-10*x 100*x];
```

The functions `abm` and `fhamming` require only the first derivative of y with respect to t and we define function thus:

```
v = @(t,x) -10*x;
```

Running the above script provides the following results, demonstrating the superiority of the Hermite method.

```
Solution of dx/dt = -10x
t          abm          Hamming      Hermite      Exact
0.00    1.0000000    1.0000000    1.0000000    1.0000000
0.10    0.3750000    0.3750000    0.3750000    0.3678794
0.20    0.1406250    0.1406250    0.1381579    0.1353353
0.30    0.0527344    0.0527344    0.0509003    0.0497871
0.40   -0.0032654    0.0109440    0.0187528    0.0183156
0.50   -0.0171851    0.0070876    0.0069089    0.0067379
0.60   -0.0010598    0.0131483    0.0025454    0.0024788
0.70    0.0023606    0.0002607    0.0009378    0.0009119
0.80   -0.0063684    0.0006066    0.0003455    0.0003355
0.90   -0.0042478    0.0096271    0.0001273    0.0001234
1.00    0.0030171   -0.0065859    0.0000469    0.0000454
```

One feature which may be used to improve many of the methods discussed above is step size adjustment. This means that we adjust the step size h according to the progress of the iteration. One criterion for adjusting h is to monitor the size of the truncation error. If the truncation error is smaller than the accuracy requirement, we can increase h; however, if the truncation error is too large, we can reduce h. Step size adjustment can lead to considerable additional work; for example, if a predictor–corrector method is used, new initial values must be calculated. The following method is an interesting alternative to this kind of procedure.

5.16 EXTRAPOLATION TECHNIQUES

The extrapolation method described in this section is based on a similar procedure to that used in Romberg integration, introduced in Chapter 4. The procedure begins by obtaining successive initial approximations for y_{n+1} using a modified mid-point method. The interval sizes used for obtaining these approximations are calculated from

$$h_i = h_{i-1}/2 \text{ for } i = 1,\ 2,\ \ldots \tag{5.59}$$

with the initial value h_0 given.

Once these initial approximations have been obtained, we can use (5.60), the extrapolation formula, to obtain improved approximations.

$$\begin{gathered} T_{m,\,k} = (4^m T_{m-1,\,k+1} - T_{m-1,\,k})/(4^m - 1) \\ \text{for } m = 1, 2, \ldots \text{ and } k = 1, 2, \ldots\ s - m \end{gathered} \tag{5.60}$$

The calculations are set out in an array in much the same way as the calculations for Romberg's method for integration described in Chapter 4. When $m = 0$, the values of $T_{0,k}$ for $k = 0, 1, 2, ..., s$ are taken as the successive approximations to the values of y_{n+1} using the h_i values obtained from (5.59).

The formula for calculating the approximations used for the initial values $T_{0,k}$ in the above array are computed using the following equations.

$$\begin{gathered} y_1 = y_0 + hy_0' \\ y_{n+1} = y_{n-1} + 2hy_n' \text{ for } n = 1,\ 2\ \ldots,\ N_k \end{gathered} \tag{5.61}$$

Here $k = 1, 2, ...$ and N_k is the number of steps taken in the range of interest, so that $N_k = 2^k$ as the size of the interval is halved each time. The independent variable increment between y_{n+1} and y_{n-1} is $2h$. Values of this increment may lead to significant variations in the magnitude of the error. Because of this, instead of using the final value of y_{n+1} given by (5.61), Gragg (1965) has suggested that at the final step these values be smoothed using the intermediate value y_n. This leads to the values for $T_{0,k}$

$$T_{0,\ k} = (y_{N-1}^k + 2y_N^k + y_{N+1}^k)/4$$

where the superscript k denotes the value at the kth division of the interval.

Alternatives to the method of Gragg are available for finding the initial values in the function `rombergx` and various combinations of predictor–correctors may be used. It should be noted, however, that if the corrector is iterated until convergence is achieved, this will improve the accuracy of the initial values but at considerable computational expense for smaller step sizes, i.e., for larger N values. The MATLAB function `rombergx` implements the extrapolation method and is given below.

```
function [v W] = rombergx(f,tspan,intdiv,inity)
% Solves dy/dt = f(t,y) using Romberg's method.
% Example call: [v W] = rombergx(f,tspan,intdiv,inity)
% The initial and final values of t are given by tspan = [start finish].
% Initial value of y is given by inity.
% The number of interval divisions is given by intdiv.
% The function f(t,y) must be defined by the user.
W = zeros(intdiv-1,intdiv-1);
for index = 1:intdiv
    y0 = inity; t0 = tspan(1);
    intervals = 2^index;
    step = (tspan(2)-tspan(1))/intervals;
    y1 = y0+step*feval(f,t0,y0);
    t = t0+step;
    for i = 1:intervals
        y2 = y0+2*step*feval(f,t,y1);
        t = t+step;
        ye2 = y2; ye1 = y1; ye0 = y0; y0 = y1; y1 = y2;
    end
    tableval(index) = (ye0+2*ye1+ye2)/4;
end
```

```
for i = 1:intdiv-1
    for j = 1:intdiv-i
        table(j) = (tableval(j+1)*4^i-tableval(j))/(4^i-1);
        tableval(j) = table(j);
    end
    tablep = table(1:intdiv-i);
    W(i,1:intdiv-i) = tablep;
end
v = tablep;
```

We can now call this function to solve $dx/dt = -10x$ with $x = 1$ at $t = 0$. The following MATLAB statement solves this differential equation when $t = 0.5$:

```
>> [fv P] = rombergx(@(t,x) -10*x,[0 0.5],7,1)

fv =
    0.0067

P =
   -2.5677    0.2277    0.1624    0.0245    0.0080    0.0068
    0.4141    0.1580    0.0153    0.0069    0.0067         0
    0.1539    0.0131    0.0068    0.0067         0         0
    0.0125    0.0068    0.0067         0         0         0
    0.0068    0.0067         0         0         0         0
    0.0067         0         0         0         0         0
```

The final value, 0.0067, is better than any of the results achieved for this problem by other methods presented in this chapter. It must be noted that only the final value is found; other values in a given interval can be obtained if intermediate ranges are considered.

This completes our discussion of those types of differential equations known as initial value problems. In Chapter 6 we consider a different type of differential equation known as a boundary value problem.

5.17 SIMULINK

Simulink is part of MATLAB and can only be used when MATLAB is installed. Simulink is used to model, simulate, and analyze dynamic systems. The systems can be linear or non-linear, modeled in continuous or sampled time, or a mixture of both. Simulink provides a graphical user interface that allows the user to drag and drop the appropriate blocks in order to create the required block diagram to model the system. The wide range of blocks in the block library comprise sources (inputs to the system), sinks (outputs from the system) and both linear and non-linear process blocks, and connectors. Simulink also allows the user to modify or create blocks of their own. The parameters for each block are set by the user to be appropriate for their specific problem.

It is reasonable to ask why Simulink was developed. MATLAB provides a range of functions to solve differential equation and hence to simulate and analyze dynamic systems. Where Simulink proves its value is in simulating more complicated dynamic systems, perhaps combining the dynamics of two or more systems. A further advantage of Simulink is that it is possible to visualize a system in terms of process block diagrams (as control engineers often do) rather than in terms of systems of differential equations. A further advantage is the ease with which non-linear phenomena such as dead zones, limits, hysteresis, and so on can be incorporated into the model.

In this brief introduction to Simulink we provide some examples of the way block diagrams can be developed to simulate some simple systems. We do not give details of how the process blocks are selected, dragged, and dropped into the model nor how connections are made between blocks, which, in any case, is quite intuitive. The purpose of this introduction is to show how simple problems can be represented by Simulink models. We do not provide the output from the models because in this context the output data is only of secondary interest.

We now provide a series of Simulink examples.

Example 5.4. Suppose we wish to simulate the dynamic system described by the following second-order differential equation:

$$m\ddot{x} + c\dot{x} + kx = f_0 \sin(\omega t) \tag{5.62}$$

We begin by isolating the highest derivative of the equation to obtain

$$\ddot{x} = (f_0/m)\sin(\omega t) - (c/m)\dot{x} - (k/m)x \tag{5.63}$$

To create the Simulink model we begin by dragging and dropping two integration process blocks, labeled internally $\frac{1}{s}$, see Fig. 5.29. The integration process blocks perform numerical integration. The input to the process block (Integrator 1) is labeled x ddot and its output is x dot. This output becomes the input to Integrator 2 and the output of this integrator is x. A connection is made from x dot to one of the gain process block, labeled Gain c/m, set to a gain of c/m. Gain means that the signal is increased or multiplied by this quantity to give an output (c/m)x dot. Similarly x is connected to the other gain process block, labeled Gain k/m, with a gain of k/m to give an output of (k/m)x. At the left of the Simulink diagram is a process block that generates a sine wave, labeled Sine Wave. The output from this block is a sine wave of amplitude f/m, and is connected to a summing unit. Also connected to the summing unit is the negatives of (c/m)x dot and (k/m)x. From (5.63), $(f_0/m)\sin(\omega t) - (c/m)\dot{x} + (k/m)x$ is equal to $\ddot{x}$, so the output of the summing block is x ddot which is connected to the input of integrator 1. Finally, x and x dot are connected to a Scope block, labeled Scope 1, which, as the name suggests, displays x and $\dot{x}$ against the simulation time.

Before we run the model, we must open the Sine Wave block and set the frequency and amplitude of the input; we must also open the integrators to specify any initial conditions required, in this case the initial velocity at Integrator 1 and initial the displacement at Integrator 2, and open the Gain blocks to set the required gains. Finally, we open the Model Simulation Parameters and set the start and end times of the simulation, the ODE solver to be used (ode45 in this case), relative and absolute accuracy, step size, and so on.

Rearranging (5.62) to isolate the $\ddot{x}$ terms is not the only arrangement we might consider for a Simulink model. We could arrange (5.62) to isolate the x term as thus:

$$x = (f_0/k)\sin(\omega t) - (m/k)\ddot{x} - (c/k)\dot{x}$$

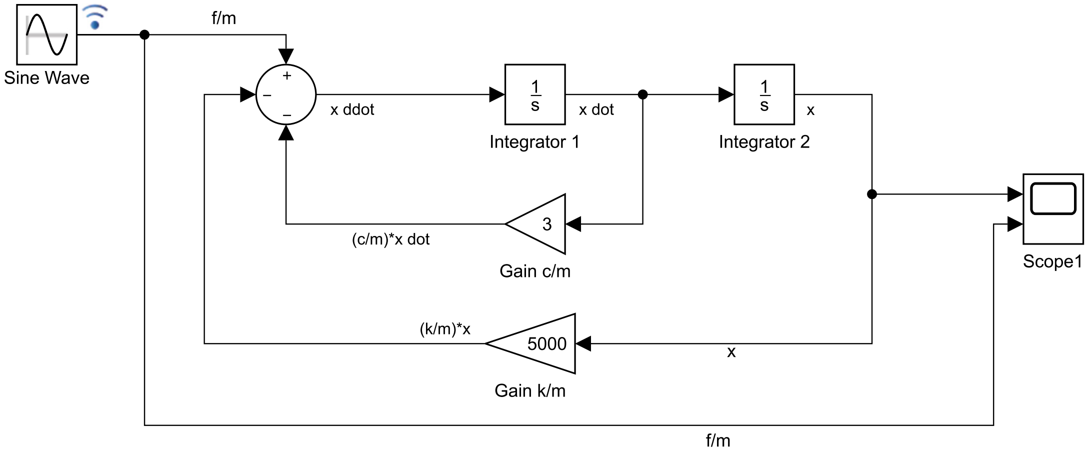

FIGURE 5.29

Model of a second-order differential equation, (5.62).

Simulating this equation would require two stages of differentiation. Although Simulink provides a differentiation process block, one should not simulate (5.62) in this way. Numerical differentiation is a noisy process that will lead to errors.

Example 5.5. In Example 5.4, we have used two integrator process blocks to determine the velocity and displacement from acceleration. Basically, we need as many stages of integration as the order of the ordinary differential equation. Because many ordinary differential equations arising from real world problems are second order, Simulink provides a second-order integrator, that is a block that combines two integrator blocks. For example, suppose we wish to simulate

$$\ddot{x} + F\text{sign}\dot{x} + kx = 0$$

Rearranging this equation in terms of $\ddot{x}$ gives

$$\ddot{x} = -F\text{sign}\dot{x} - kx$$

In this example, there is no external excitation, only the non-zero initial displacements and/or velocities cause the subsequent changes in x. Also, there is no $\dot{x}$ term. Energy dissipation is by a constant force F which opposes the motion and changes direction with the sign of $\dot{x}$. This form of energy dissipation is sometimes called Coulomb damping. The Simulink model for the system is shown in Simulink Fig. 5.30.

Note the ease with which we are able to model the non-linear Coulomb damping.

Example 5.6. In this example, we simulate the same system as simulated in Example 5.4. However, rather than a simulation based on the ODE that describes the system, we begin by converting (5.62) to an equivalent transfer function. To do this we take the Laplace Transform of (5.62). The Laplace transform is described in Section 10.12 of Chapter 10.

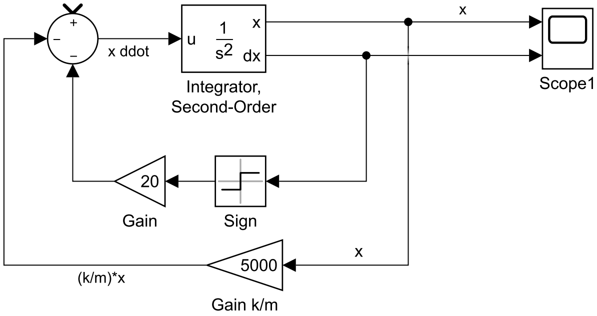

FIGURE 5.30

Model of a second-order differential equation with Coulomb damping.

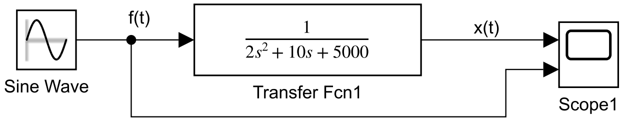

FIGURE 5.31

A second-order system modeled by a transfer function.

Denoting the Laplace transform of $x(t)$ by $X(s)$ then $\mathcal{L}(x(t)) = X(s)$ and $\mathcal{L}(\dot{x}(t)) = sX(s) + x(0)$ and $\mathcal{L}(\ddot{x}(t)) = s^2X(s) + sx(0) + v(0)$ and where $x(0)$ and $v(0)$ are the initial displacement and velocity, respectively. Applying this transform to (5.62) and assuming the initial conditions are zero gives

$$(ms^2 + cs + k)X(s) = F(s) \tag{5.64}$$

Rearranging, (5.64) becomes

$$\frac{X(s)}{F(s)} = \frac{1}{ms^2 + cs + k} \tag{5.65}$$

The expression on the right side of (5.65) is the transfer function for (5.62). Comparing Simulink Fig. 5.31 with Simulink Fig. 5.29, we see that the two integrators and two gain processes have been replaced by a Simulink transfer function block.

Example 5.7. We now give the Simulink model for Van der Pol's equation. Van der Pol's equation is

$$\ddot{x} - \mu(1 - x^2)\dot{x} + x = 0$$

The equation named after a Dutch electrical engineer who investigated the performance of an electrical circuit described by this equation. The equation has a non-linear damping term. To model it in

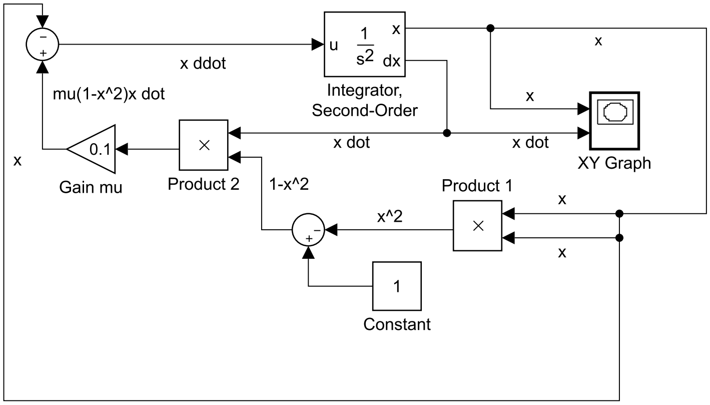

FIGURE 5.32

Model of Van der Pol's equation.

Simulink we express the equation as

$$\ddot{x} = \mu(1 - x^2)\dot{x} - x$$

The block diagram, Simulink Fig. 5.32 shows to stages of integration (using a double integration block). The x output is connected to both inputs of the product block to give the output signal x squared. In a summing block, this output is subtracted from unity. The output from the summing block is multiplied by x dot and then x is subtracted from it to generate x ddot; the input to the integrator. Note that instead of using a Scope block to view x or x dot against time, here we use an XY block and view x against x dot. If we start with a small initial condition, the oscillations grow and reach a *limit cycle* beyond which the vibration cannot grow. In contrast, if we start with a large initial condition the vibrations level decreases until a limit cycle is reached, below which the vibration level cannot fall. By examining the coefficient of $\dot{x}$ we can intuitively recognize this. When $x^2 < 1$ the $\dot{x}$ term is negative and the vibration level will increase. When $x^2 > 1$ the $\dot{x}$ term is positive and vibration levels will decrease.

Example 5.8. Consider the equations

$$m_{11}\ddot{x} + c_{11}\dot{x} + c_{12}\dot{y} + k_{11}x + k_{12}y = X(t) \tag{5.66}$$

$$m_{22}\ddot{y} + c_{21}\dot{x} + c_{22}\dot{y} + k_{21}x + k_{22}y = Y(t) \tag{5.67}$$

We will initially neglect the damping, ($\dot{x}$ and $\dot{y}$), terms. Thus, with some rearrangement (5.66) and (5.67) become

$$\ddot{x} = -(k_{11}/m_{11})x - (k_{12}/m_{11})y + X(t) \tag{5.68}$$

$$\ddot{y} = -(k_{21}/m_{22})x - (k_{22}/m_{22})y + Y(t) \tag{5.69}$$

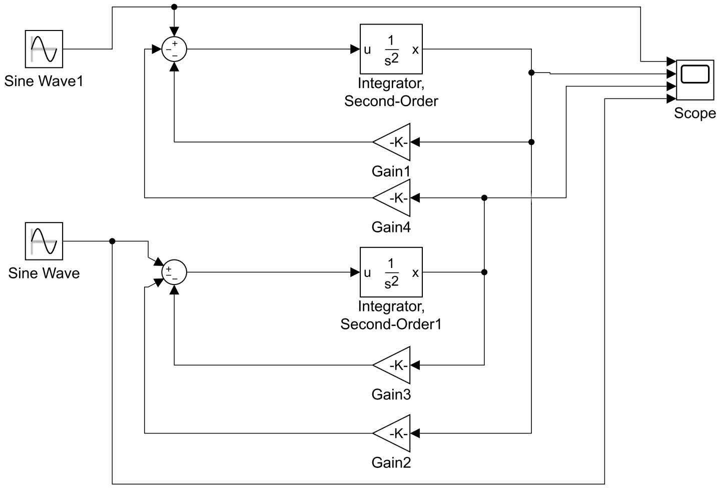

FIGURE 5.33

Model of a pair of simultaneous ordinary differential equations.

Fig. 5.33 illustrates the difficulty of simulating several simultaneous ODEs as separate ODE with interconnections. In this case we only have two equations and we have neglected $\dot{x}$ and $\dot{y}$ terms, but the flow diagram is already cluttered; many more ordinary differential equations in the system would make assembling a process diagram impossible.

However, if the two simultaneous equations, (5.66) and (5.67), are written in matrix notation we have

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{C}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f}(t) \tag{5.70}$$

where $\mathbf{q}$ is a vector of system displacements, $[x\ y]^{\mathrm{T}}$, $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are the system mass, damping, and stiffness matrices and $\mathbf{f}(\mathbf{t})$ is a vector of input forces, $[X\ Y]^{\mathrm{T}}$. The easiest way of simulating this system in Simulink is to use a space-state representation. The classical state space model for a system is described by:

$$\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}\\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}\end{aligned} \tag{5.71}$$

where $\mathbf{A}$ is called the state or system matrix, $\mathbf{B}$ is the input matrix, $\mathbf{C}$ is the output matrix and $\mathbf{D}$ is called the feed-through matrix. The vector $\mathbf{x}$ is the state vector, $\mathbf{u}$ is the input vector, and $\mathbf{y}$ is the output vector.

The question is, how do we rearrange (5.70) into a state space form? To obtain the state space representation we begin by multiplying (5.70) by $\mathbf{M}^{-1}$ to give

$$\ddot{\mathbf{q}} = -\mathbf{M}^{-1}\mathbf{C}\dot{\mathbf{q}} - \mathbf{M}^{-1}\mathbf{K}\mathbf{q} + \mathbf{M}^{-1}\mathbf{f}(t) \tag{5.72}$$

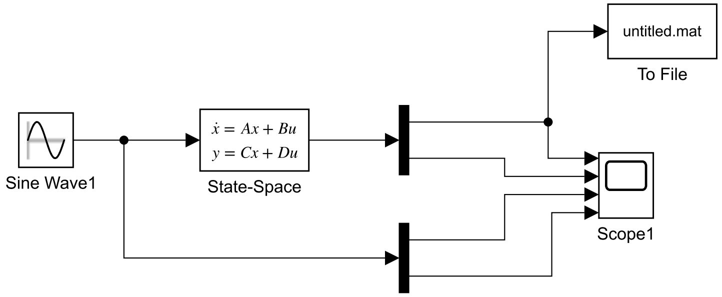

FIGURE 5.34

Two simultaneous ordinary differential equations modeled in state space form.

We now let $\mathbf{q} = \mathbf{x}_1$, $\dot{\mathbf{q}} = \mathbf{x}_2$, and $\mathbf{f}(t) = \mathbf{u}(t)$ so that (5.72) becomes

$$\dot{\mathbf{x}}_2 = -\mathbf{M}^{-1}\mathbf{C}\mathbf{x}_2 - \mathbf{M}^{-1}\mathbf{K}\mathbf{x}_1 + \mathbf{M}^{-1}\mathbf{u}(t) \tag{5.73}$$

Noting that

$$\dot{\mathbf{x}}_1 = \mathbf{x}_2 \tag{5.74}$$

we can combine (5.73) and (5.74) to give

$$\begin{bmatrix} \dot{\mathbf{x}}_1 \\ \dot{\mathbf{x}}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\mathbf{M}^{-1}\mathbf{K} & -\mathbf{M}^{-1}\mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} + \begin{bmatrix} \mathbf{0} \\ \mathbf{M}^{-1} \end{bmatrix} \mathbf{u}(t) \tag{5.75}$$

and

$$\mathbf{q} = [\mathbf{I} \quad \mathbf{0}] \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} + \mathbf{0}\mathbf{u}(t) \tag{5.76}$$

Comparing (5.75) and (5.76) with (5.71) we see that

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{I} \\ -\mathbf{M}^{-1}\mathbf{K} & -\mathbf{M}^{-1}\mathbf{C} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} \mathbf{q} \\ \dot{\mathbf{q}} \end{bmatrix} \quad \mathbf{B} = \begin{bmatrix} \mathbf{0} \\ \mathbf{M}^{-1} \end{bmatrix} \quad \mathbf{u}(t) = \begin{bmatrix} \mathbf{0} \\ \mathbf{I} \end{bmatrix} \mathbf{f}(t)$$

and

$$\mathbf{y} = \begin{bmatrix} \mathbf{q} \\ \mathbf{0} \end{bmatrix} \quad \mathbf{C} = [\mathbf{I} \quad \mathbf{0}] \quad \mathbf{D} = \mathbf{0}$$

Simulink has a state space process block. The matrices **A**, **B**, **C**, and **D** are specified and an input **u** must be provided. The output from the process, **y** can then be connected to a display. This is illustrated in Simulink Fig. 5.34.

The Simulink simulations shown in Examples 5.4 to 5.8 have one feature in common; they are all simulations of differential equations. It is possible to simulate algebraic equations, for example

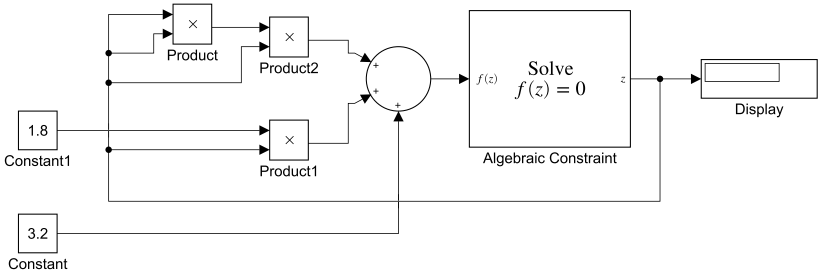

FIGURE 5.35

Model to determine the root of a cubic equation.

we could easily simulate $y = ax + b$ in Simulink. The danger when simulating algebraic equation is the accidental inclusion of an "algebraic loop". This is a loop exists in the Simulink block diagram with only direct feed-through blocks within the loop. A feed-through block output depends on its input and no integration is involved. Examples of a feed-through blocks include a gain block and math function blocks. It further illustrates an algebraic loop, consider a simulation that involves an input x to a summing block. If there are no other inputs to the summing block, the output from the block will also be x. However, suppose we take a connection from the output, pass it through a gain block, with a gain of 10, and connect it to the summing block. Then an input of $x = 1$ will produce a feedback of 10 into the summing junction which will give an output of $1 + 10 = 1$. This output is feedback and becomes $10 \times 11 = 110$ at the summing junction. Clearly, the output of the summing block rapidly tends a numeric overflow and Simulink stops. The following example solves an algebraic equation by simulation, but does contain an algebraic loop.

Example 5.9. Consider a model of a process to determine the smallest root of the cubic equation

$$x^3 + ax + b = 0$$

Note that there is no x^2 term in this equation. However, Simulink can find the lowest root of any equation.

The Simulink Fig. 5.35 shows the block diagram for the process. At the left side of the diagram we see two inputs. One input, the constant b, is connected directly to a summing block; the other input, the constant a, is multiplied by x and connected to a summing block. Also connected to the summing block is $x \times x^2 = x^3$. The output from the summing block, equal to $x^3 + ax + b$ is fed into the block labeled Solve $f(z) = 0$. This block iteratively determines a value of x to make the input to the block equal zero. At each iteration the current best estimate for x is fed back to allow an update value for $x^3 + ax + b$ to be determined. Once convergence has reached the tolerance level set in the Solve $f(z) = 0$ block, the value of x is passed to the display.

Simulink allows the user to mask a subsection of a block diagram. A mask hides the details and content of a selected subsystem so that it appears in the block diagram as a single block. (See Fig. 5.36.) If desired by the user, the masked block can have its own parameter dialog box and icon. Furthermore, it can be placed in the custom created block library.

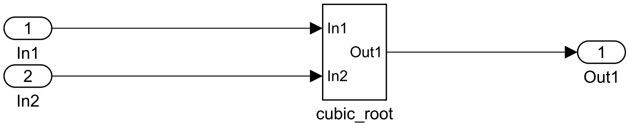

FIGURE 5.36

The Simulink model of Fig. 5.35 replaced by a single mask.

For readers with Simulink, an excellent example of a Simulink model can be found by from the MATLAB command window by entering

```
>> shower
```

This is a simulation of a temperature control system for a shower that uses fuzzy logic. There are four masked blocks. For example the hot water valve block itself contains seven lower level blocks.

5.18 SUMMARY

In this chapter, we have developed a range of MATLAB functions for solving differential equations and systems of differential equations which supplement those provided in MATLAB. We have demonstrated how these functions may be used to solve a wide variety of problems. One particular problem studied is that of a differential equation that is chaotic. We have also briefly introduced the MATLAB Simulink toolbox.

5.19 PROBLEMS

5.1. A radioactive material decays at a rate that is proportional to the amount that remains. The differential equation which models this process is

$$dy/dt = -ky \text{ where } y = y_0 \text{ when } t = t_0$$

Here y_0 represents the mass at the time t_0. Solve this equation for $t = 0$ to 10 given that $y_0 = 50$ and $k = 0.05$, using:
(**a**) the function `feuler`, with $h = 1, 0.1, 0.01$.
(**b**) the function `eulertp`, with $h = 1, 0.1$.
(**c**) the function `rkgen`, set for the classical method, with $h = 1$.
Compare your results with the exact solution, $y = 50\exp(-0.05t)$.

5.2. Solve $y' = 2xy$ with initial conditions $y_0 = 2$ when $x_0 = 0$ in the range $x = 0$ to 2. Use the classical, Merson, and Butcher variants of the Runge–Kutta method, all implemented in function `rkgen`, with step $h = 0.2$. Note that the exact solution is $y = 2\exp(x^2)$.

5.3. Repeat Problem 5.1 using the following predictor–corrector methods with $h = 2$ and for $t = 0$ to 50:

(**a**) Adams–Bashforth–Moulton's method, function `abm`.
(**b**) Hamming's method, function `fhamming`.

5.4. Express the following second-order differential equation as a pair of first-order equations:

$$xy'' - y' - 8x^3y^3 = 0$$

with the initial conditions $y = 1/2$ and $y' = -1/2$ at $x = 1$. Solve the pair of first-order equations using both `ode23` and `ode45` in the range 1 to 4. The exact solution is given by $y = 1/(1 + x^2)$.

5.5. Use function `fhermite` to solve:
(**a**) Problem 5.1 with $h = 1$.
(**b**) Problem 5.2 with $h = 0.2$.
(**c**) Problem 5.2 with $h = 0.02$.

5.6. Use the MATLAB function `rombergx` to solve the following problems. In each case use eight divisions.
(**a**) $y' = 3y/x$ with initial conditions $x = 1$, $y = 1$. Determine y when $x = 20$.
(**b**) $y' = 2xy$ with initial conditions $x = 0$, $y = 2$. Determine y when $x = 2$.

5.7. Consider the predator–prey problem described in Section 5.10.2. This problem may be extended to consider the effect of culling on the interacting populations by subtracting a term from both equations in (5.28) as follows:

$$dP/dt = K_1P - CPQ - S_1P$$
$$dQ/dt = K_2Q - DPQ - S_2Q$$

Here S_1 and S_2 are constants which provide the culling level for the populations. Use `ode45` to solve this problem with $K_1 = 2$, $K_2 = 10$, $C = 0.001$, and $D = 0.002$ and initial values of the population $P = 5000$ and $Q = 100$. Assuming that S_1 and S_2 are equal, experiment with values in the range 1 to 2. There is a wealth of experimental opportunity in this problem and the reader is encouraged to investigate different values of S_1 and S_2.

5.8. Solve the Lorenz equations given in Section 5.12.1 for $r = 1$, using `ode23`.

5.9. Use the Adams–Bashforth–Moulton method to solve $dy/dt = -5y$, with $y = 50$ when $t = 0$, in the range $t = 0$ to 6. Try step sizes, h, of 0.1, 0.2, 0.25, and 0.4. Plot the error against t for each case. What can you deduce from these results with regard to the stability of the method? The exact answer is $y = 50e^{-5t}$.

5.10. The following first-order differential equation represents the growth in a population in an environment that can support a maximum population of K:

$$dN/dt = rN\,(1 - N/K)$$

where $N(t)$ is the population at time t and r is a constant. Given $N = 100$ when $t = 0$, we use the MATLAB function `ode23`, solve this differential equation in the range 0 to 200, and plot a graph of N against time. Take $K = 10{,}000$ and $r = 0.1$.

5.11. The Leslie–Gower predator–prey problem takes the form

$$dN_1/dt = N_1(r_1 - cN_1 - b_1N_2)$$
$$dN_2/dt = N_2\,(r_2 - b_2\,(N_2/N_1))$$

where $N_1 = 15$ and $N_2 = 15$ at time $t = 0$. Use `ode45` to solve this equation given $r_1 = 1$, $r_2 = 0.3$, $c = 0.001$, $b_1 = 1.8$, and $b_2 = 0.5$. Plot N_1 and N_2 in the range $t = 0$ to 40.

5.12. By setting $u = dx/dt$, reduce the following second-order differential equation to two first-order differential equations.

$$\frac{d^2x}{dt^2} + k\left(\frac{1}{v_1} + \frac{1}{v_2}\right)\frac{dx}{dt} = 0$$

where $x = 0$ and $dx/dt = 10$ when $t = 0$. Use the MATLAB function `ode45` to solve this problem given that $v_1 = v_2 = 1$ and $k = 10$.

5.13. A model for a conflict between guerilla, g_2, and government forces, g_1, is given by the equations

$$dg_1/dt = -cg_2$$
$$dg_2/dt = -rg_2g_1$$

Given that the government forces number 2000 and the guerilla forces number 700 at time $t = 0$, use the function `ode45` to solve this system of equations, taking $c = 30$ and $r = 0.01$. You should solve the equations over the interval time interval 0 to 0.6. Plot a graph of the solution showing the changes in government and guerilla forces over time.

5.14. The following differential equation provides a simple model of a suspension system. The constant m gives the mass of moving parts, the constant k relates to stiffness of the suspension system and the constant c is a measure of the damping in the system. F is a constant force applied at $t = 0$.

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = F$$

Given that $m = 1$, $k = 4$, $F = 1$ and both $x = 0$ and $dx/dt = 0$ when $t = 0$, use `ode23` to determine the response, $x(t)$ for $t = 0$ to 8 and plot a graph of x against t in each case. Assume the following values for c:

(a) $c = 0$ **(b)** $c = 0.3\surd(mk)$ **(c)** $c = \surd(mk)$

(d) $c = 2\surd(mk)$ **(e)** $c = 4\surd(mk)$

Comment on the nature of your solutions. The exact solutions are as follows:

For cases (**a**), (**b**), and (**c**) $x(t) = \frac{F}{k}\left[1 - \frac{1}{\sqrt{1-\zeta^2}}e^{-\zeta\omega_n t}\cos(\omega_d t - \phi)\right]$

For case (**d**) $x(t) = \frac{F}{k}\left[1 + (1 + \omega_n t)\, e^{-\omega_n t}\right]$

where

$$\omega_n = \sqrt{k/m}, \quad \zeta = c/(2\sqrt{mk}), \quad \omega_d = \omega_n\sqrt{1-\zeta^2}$$

$$\phi = \tan^{-1}(\zeta/\sqrt{1-\zeta^2})$$

For case (**e**)

$$x(t) = \frac{F}{k}\left[1 + \frac{1}{2q}\left(s_2e^{s_1t} - s_1e^{s_2t}\right)\right]$$

where

$$q = \omega_n \surd \zeta^1 -! \quad s_1 = -\zeta\omega_n + q \quad s_2 = -\zeta\omega_n - q$$

Plot these solutions and compare them with your numerical solutions.

5.15. Gilpin's system for modeling the behavior of three interacting species is given by the differential equations:

$$\begin{aligned}
dx_1/dt &= x_1 - 0.001x_1^2 - 0.001kx_1x_2 - 0.01x_1x_3 \\
dx_2/dt &= x_2 - 0.001kx_1x_2 - 0.001x_2^2 - 0.001x_2x_3 \\
dx_3/dt &= -x_3 + 0.005x_1x_3 + 0.0005x_2x_3
\end{aligned}$$

given $x_1 = 1000$, $x_2 = 300$, and $x_3 = 400$ at time $t = 0$. Taking $k = 0.5$, use `ode45` to solve this system of equations in the range $t = 0$ to $t = 50$ and plot the behavior of the population of the three species against time.

5.16. A problem which arises in planet formation is where a range of objects called planetesimals coagulate to form larger objects and this coagulation continues until a stable state is reached where a number planetary size objects have been created. To simulate this situation we assume that a minimum size object exists of mass m_1 and the masses of all other objects are integral multiples of the mass of this object. Thus there are n_k objects of mass m_k where $m_k = km_1$. Then the manner in which the number of objects of specific mass changes over time t is given by the *Coagulation Equation* as follows:

$$\frac{dn_k}{dt} = \frac{1}{2}\sum_{i+j=k} A_{ij}n_in_j - n_k \sum_{i=k+1}^{\max k} A_{ki}n_i$$

The values A_{ij} are the probabilities of collisions between the objects i and j. A simple interpretation of this equation is that number of bodies of mass n_k is increased by collisions between bodies of lesser mass but the decreased by collisions with larger bodies.

As an exercise write out the equations for this system for the case where there are only three different sizes of planetesimals and assign A_{ij} equal to $n_in_j/(1000(n_i + n_j))$. Note that the division by 1000 ensures that impacts are relatively rare which seems a plausible assumption in the vast volume of space considered. The initial values for the numbers of planetesimals n_1, n_2, n_3 are as taken as 200, 25, and 1, respectively.

Solve the resulting system using the MATLAB function `ode45` using a time interval of 2 units. Study the case where the values of the collision probabilities are calculated using the varying values of the number of planetesimals as time varies. Plot graphs of your results.

5.17. The following example studies the effect of life on a planetary environment. A relatively simple way of studying these effects is to consider the concept of daisy world. This envisages a world inhabited by only two life forms; white and black daisies. This situation can be modeled a pair of differential equations where the area covered by the black daisies a_b and the area covered by the white daisies a_w changes with time t as follows:

$$\begin{aligned}
da_b/dt &= a_b(x\beta_b - \gamma) \\
da_w/dt &= a_w(x\beta_w - \gamma)
\end{aligned}$$

where $x = 1 - a_b - a_w$ represents the area not covered by either daisy assuming the total area of the planet is represented by unity. The value of γ gives the death rate for the daisies and β_b and β_w give the growth rate for the black and white daisies respectively. This is related to the energy they receive from the planetary Sun or the local temperature. Consequently, an empirical formula may be given for these values as follows:

$$\beta_b = 1 - 0.003265(295.5 - T_b)^2$$

and

$$\beta_w = 1 - 0.003265(295.5 - T_w)^2$$

where the values of T_b and T_w lie in the range 278 to 313 K, where K denotes degrees Kelvin. Outside this range growth is assumed zero. Taking $\gamma = 0.3$, $T_b = 295$ K, $T_w = 285$ K and initial values for $a_b = 0.2$, $a_w = 0.3$, solve the system of equations for $t = [0, 10]$ using the MATLAB function `ode45`. Plot graphs of the changes in a_b and a_w with respect to time. It should be noted that the extent of the areas covered by the black and white daisies will affect the overall temperature of the planet, since white and dark areas react differently in the way they absorb energy from the Sun.

5.18. In an article by Pearce and Shearer (2016), the authors describe how capillaries work in the circulatory system and how they can be modeled using a network of resistors and capacitors. In particular the manner in which the capacitor operates with an applied fluctuating current and consequently alternatively releasing and storing voltage through the resistor reflects the behavior of the circulatory system. The model of the corresponding basic electrical circuit behavior over time t, is well established and takes the form:

$$C\frac{dV}{dt} + \frac{V}{R} = I(t)$$

where V is the voltage, R the resistance, C the capacity, and I the current. In terms of blood flow this may be translated as:

$$C\frac{dP}{dt} + \frac{P}{R} = Q_0 \sin(\pi t/t_0)$$

where C is called the compliance of the aorta, which corresponds to the concept of capacity, R is the total resistance of the rest of the circulatory system and the right-hand side of the equation models the flow of the heart, t_0 is the initial time and Q_0 is the maximum flow rate. We can use this first-order differential equation to find the varying values of P.

Solve this equation using the MATLAB function for the solution of differential equations, `ode45`, taking $R = 10$, $C = 1.5$, $R = 40$, $Q_0 = 2.8$, and $t_0 = 0$.