CHAPTER 13

ODE: Systems, Stiffness, Stability

In this chapter, we extend the techniques presented in the previous chapter to higher-order ODE and systems of first-order ODE. We begin by showing how a higher-order ODE can be converted into a system of first-order ODE. In the second section, we treat the Euler and midpoint methods for second-order ODE and systems of two first-order ODE in some detail. This serves to emphasize the direct relationship between each of the methods for a single ODE and the corresponding method for a system of ODE.

For larger first-order systems, we can update all components of the solution very easily by utilizing MATLAB's vector capabilities. The function for each of the methods presented in the previous chapter can be applied to systems of arbitrary size with only minor modifications. Although the methods presented in this chapter are direct extensions of those seen in the last chapter, the variety of applications that can be solved is greatly expanded. We illustrate the methods using simple examples and problems, including the nonlinear equation of motion of a simple pendulum, a spring–mass system, and a two-link robot arm. Sample problems describing chemical reactions are also solved.

Ordinary differential equations can be used to describe a wide variety of processes. Population growth models, predator-prey models, radioactive carbon dating, combat models, traffic flow models, and mechanical and electrical vibrations are a few of the most common applications; the list of possibilities is almost endless.

The solution of ODE initial-value problems forms the basis for the shooting method, one of the approaches to solving boundary-value problems for ordinary differential equations. The study of numerical methods for ODE-BVP is the subject of the next chapter.

Application 13-A Motion of a Pendulum

The motion of a pendulum of length L subject to damping can be described by the angular displacement of the pendulum from vertical, u, as a function of time. (See Figure 13.1.) If we let m be the mass of the pendulum, g the gravitational constant, and c the damping coefficient (i.e., the damping force is $F = -cu'$), then the ODE initial-value problem describing this motion is

$$u'' + \frac{c}{mL}u' + \frac{g}{L}\sin(u) = 0$$

The initial conditions give the angular displacement and velocity at time zero; for example, if $u(0) = a$ and $u'(0) = 0$, the pendulum has an initial displacement but is released with 0 initial velocity.

Analytic (closed-form) solutions rely on approximating $\sin(u)$; Figure 13.2 illustrates the difference in motion for the first 15 sec. using the linearized equation

$$u'' + \frac{c}{mL}u' + \frac{g}{L}(u) = 0$$

versus the nonlinear equation, for an initial displacement of $u(0) = \pi/2$. (See Greenspan, 1974, for further discussion.)

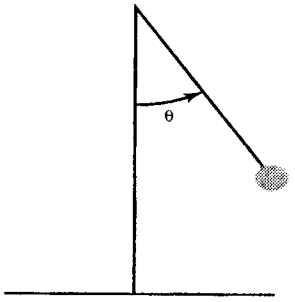

FIGURE 13.1: Simple pendulum.

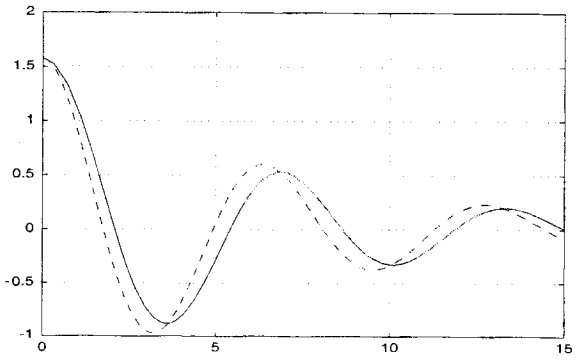

FIGURE 13.2: Motion of a simple pendulum, nonlinear ODE (solid), and linearized ODE (dashed).

Application 13-B Chemical Flow

A circular reaction involving three chemical reactions can be described as

```
                    k1
A + A' ----------------------> B

                    k2
B + B' ----------------------> C

                    k3
C + C' ----------------------> A
```

We assume that compounds A', B' and C' are present in excess, so that changes in their quantities can be neglected, and we simplify the notation by defining $r_1 = k_1A', r_2 = k_2B'$, and $r_3 = k_3C'$; the differential equations can be written as

$$\frac{dA}{dt} = r_3C - r_1A$$

$$\frac{dB}{dt} = r_1A - r_2B$$

$$\frac{dC}{dt} = r_2B - r_3C$$

If the reaction rates are constants, the solution can be found from the eigenvalues and eigenvectors of the coefficient matrix $\mathbf{R}$ when the differential equation is written in matrix-vector form $\mathbf{x}' = \mathbf{R}\mathbf{x}$ with $\mathbf{x} = [A, B, C]^T$ and

$$\mathbf{R} = \begin{bmatrix} -r_1 & 0 & r_3 \\ r_1 & -r_2 & 0 \\ 0 & r_2 & -r_3 \end{bmatrix}$$

This is discussed briefly in Problems C5.1–C5.14.

On the other hand, if the reaction rates are not constant, numerical methods may be especially useful. For example, we can take $r_2 = 2, r_3 = 1$, and r_1 changing from 0.1 to 10. (The units for these parameters are sec^{-1}.) The appropriate initial values of the unknown functions A, B, and C depend on the total amount of the three chemicals that is present ($Q = A + B + C$) and the rate constants; the initial values are chosen to be consistent with the equilibrium values, which are

$$A = \frac{Q}{1 + r_1/r_2 + r_1/r_3}, \quad B = \frac{r_1}{r_2}A, \quad C = \frac{r_1}{r_3}A$$

(For further discussion see Simon, 1986, p. 118.)

13.1 SYSTEMS

Systems of first-order ODE may arise in applications, or may result from conversion of a higher-order ODE to a system of first-order ODE. We illustrate this conversion process, first for a second-order ODE and then for a more general higher-order ODE.

Transforming Higher-Order ODE

A second-order ODE of the form

$$y'' = g(x, y, y')$$

can be converted to a system of two first-order ODE by a simple change of variables:

$$u = y$$
$$v = y'$$

The differential equations relating these variables (functions) are

$$u' = v = f(x, u, v)$$
$$v' = g(x, u, v)$$

The initial conditions for the original ODE,

$$y(0) = a_0, \quad y'(0) = a_1$$

become the initial conditions for the system, i.e.,

$$u(0) = a_0, \quad v(0) = a_1$$

EXAMPLE 13.1 Motion of a Pendulum

Consider the motion of a pendulum described at the beginning of the chapter, with angular displacement $y(x)$ given by

$$y'' + \frac{c}{mL}y' + \frac{g}{L}\sin(y) = 0; \quad y(0) = a, \quad y'(0) = b$$

Choosing $g/L = 1$ and $c/(mL) = 0.3, a = \pi/2$, and $b = 0$, we get the second-order ODE-IVP

$$y'' = -0.3y' - \sin(y)$$

which can be converted to a system of first-order ODE by means of the change of variables

$$u = y$$
$$v = y'$$

The differential equations relating these variables are

$$u' = f(x, u, v) = v$$
$$v' = g(x, u, v) = -0.3v - \sin(u)$$

with initial conditions $u(0) = \pi/2, v(0) = 0$.

We investigate the application of Euler's method and the midpoint method to this and other systems of two first-order ODE in the next section.

The n^{th}-order ODE

$$y^{(n)} = f(x, y, y', y', \ldots, y^{(n-1)})$$

$$y(0) = a_0, \quad y'(0) = a_1, \quad y''(0) = a_2, \quad \ldots, \quad y^{(n-1)}(0) = a_{n-1}$$

becomes a system of first-order ODE by the following change of variables:

$$\begin{aligned} u_1 &= y \\ u_2 &= y' \\ u_3 &= y'' \\ &\vdots \\ u_n &= y^{(n-1)} \end{aligned}$$

The differential equations relating these variables are

$$\begin{aligned} u_1' &= g_1(x, u_1, u_2, u_3, \ldots, u_n) = u_2 \\ u_2' &= g_2(x, u_1, u_2, u_3, \ldots, u_n) = u_3 \\ u_3' &= g_3(x, u_1, u_2, u_3, \ldots, u_n) = u_4 \\ &\vdots \\ u_n' &= g_n(x, u_1, u_2, u_3, \ldots, u_n) = f(x, u_1, u_2, u_3, \ldots, u_n) \end{aligned}$$

with the initial conditions

$$u_1(0) = a_0, \quad u_2(0) = a_1, \quad u_3(0) = a_2, \quad \ldots, \quad u_n(0) = a_{n-1}$$

EXAMPLE 13.2 A Higher-Order ODE

Consider the equation

$$y''' = f(x, y, y', y'') = x + 2y - 3y' + 4y''$$

with initial conditions

$$y(0) = 4, \quad y'(0) = 3, \quad y''(0) = 2$$

The system of ODE is

$$\begin{aligned} u_1' &= f_1(x, u_1, u_2, u_3) = u_2 \\ u_2' &= f_2(x, u_1, u_2, u_3) = u_3 \\ u_3' &= f_3(x, u_1, u_2, u_3) = x + 2u_1 - 3u_2 + 4u_3 \end{aligned}$$

We investigate the application of several of the methods from Chapter 12 to general systems of ODE in Sec. 13.2. By utilizing MATLAB's vector capabilities, only minor changes are required to the functions presented in Chapter 12.

13.1.1 Systems of Two ODE

Any of the methods for solving ODE-IVP discussed in Chapter 12 can be generalized to apply to systems of equations. We begin by considering systems of two first-order ODE in detail, using the Euler and Runge-Kutta midpoint methods. In these small examples, we treat the component functions separately to emphasize how each is updated. Then we treat systems of arbitrary size, using MATLAB's vector capabilities extensively.

Euler's Method

To apply the basic Euler's method

$$y_{i+1} = y_i + hF(x_i, y_i)$$

to the system of ODE

$$u' = f(x, u, v)$$
$$v' = g(x, u, v)$$

we update the function u using $f(x, u, v)$ and update v using $g(x, u, v)$.

The same step size h is used for each function (since that refers to the spacing of the independent variable x):

$$u(i+1) = u(i) + h\ f(x(i), u(i), v(i))$$

$$v(i+1) = v(i) + h\ g(x(i), u(i), v(i))$$

The following MATLAB function uses two inline functions, `f` and `g`, to define the ODE.

Euler's Method for a System of Two ODE

```
function [x, u, v] = Euler_sys2(f, g, a, b, u0, v0, n)
% solve the first-order system
% u' = f(x, u, v)
% v' = g(x, u, v)
% f and g are inline functions
h = (b-a)/n;
x = a : h : b;
u(1) = u0 ;
v(1) = v0 ;
for i = 1 : n
    u(i+1) = u(i) + h*f(x(i), u(i), v(i));
    v(i+1) = v(i) + h*g(x(i), u(i), v(i));
end
```

EXAMPLE 13.3 Motion of a Pendulum Using Euler's Method

Consider the system of ODE obtained from the nonlinear second-order ODE for the motion of the pendulum described in Application 13-A and Example 13.1.

$$u' = f(x, u, v) = v$$
$$v' = g(x, u, v) = -0.3v - \sin(u)$$

The initial conditions are $u_0 = \pi/2$, and $v_0 = 0$.

```
% example13_2
f = inline('v', 'x','u', 'v');
g = inline('-0.3*v - sin(u)', 'x','u', 'v');
[ x, u, v ] = Euler_sys2( f, g, 0, 15, pi/2, 0, 200 );
plot(x,u,x,v,'--')
grid on
```

The motion for the first 15 seconds is shown in Figure 13.3 for $n = 50, n = 100$, and $n = 200$. The corresponding step sizes are $h = 0.3$, $h = 0.15$, and $h = 0.075$. The differences in the solution shown for these three step sizes illustrates the sensitivity of the method to the choice of parameters (such as step size). The solutions for the larger step sizes ($n = 50$ or $n = 100$) are not accurate. In order to get accurate results, we must take a fairly small step size, e.g., $n = 200$. The solution using even smaller steps is essentially the same as that shown here for $n = 200$.

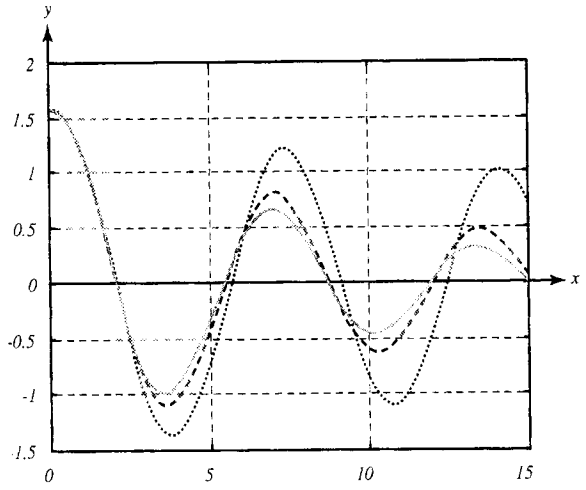

FIGURE 13.3: Oscillations of a pendulum: $n = 200$ (solid line), $n = 100$ (dashed line), $n = 50$ (dotted line).

Midpoint Method

The idea in generalizing a Runge-Kutta method to a system of two equations is the same as for Euler's method; that is, we update each unknown function u and v, using the basic Runge-Kutta formulas and the appropriate right-hand-side function, f or g, from the differential equation for the unknown.

$$u' = f(x, u, v), \quad v' = g(x, u, v)$$

We rewrite the formulas for the Runge-Kutta midpoint method,

$$k_1 = hf(x_i, y_i), \quad k_2 = hf(x_i + 0.5h, y_i + 0.5k_1), \quad y_{i+1} = y_i + k_2$$

using k_1 and k_2 to represent the update quantities for the unknown function u and calling m_1 and m_2 the corresponding quantities for the function v. We update function f by the appropriate multiple of k_1 or k_2 and function g by the corresponding amount of m_1 or m_2. This means that k_1 and m_1 must be computed before k_2 and m_2 can be found. Thus,

$$\begin{aligned} k_1 &= hf(x_i, u_i, v_i) \\ m_1 &= hg(x_i, u_i, v_i) \\ k_2 &= hf(x_i + 0.5h, u_i + 0.5k_1, v_i + 0.5m_1) \\ m_2 &= hg(x_i + 0.5h, u_i + 0.5k_1, v_i + 0.5m_1) \\ u_{i+1} &= u_i + k_2 \\ v_{i+1} &= v_i + m_2 \end{aligned}$$

The following MATLAB function uses two inline functions, **f** and **g**, to define the ODE. The use of this function is illustrated in the next example.

Midpoint Method for Two ODE

```
function [ x, u, v ] = RK2_sys2( f, g, a, b, u0, v0, n )
h = (b-a)/n;
x = a : h : b;
u(1) = u0;
v(1) = v0;
for i = 1 : n
    k1 = h*f( x(i), u(i), v(i) );
    m1 = h*g( x(i), u(i), v(i) );
    k2 = h*f(x(i) + 0.5*h, u(i) +0.5*k1, v(i) +0.5*m1);
    m2 = h*g(x(i) + 0.5*h, u(i) +0.5*k1, v(i) +0.5*m1);
    u(i+1) = u(i) + k2;
    v(i+1) = v(i) + m2;
end
```

EXAMPLE 13.4 Motion of a Pendulum Using the Midpoint Method

Consider again the nonlinear system of ODE obtained from the second-order ODE for the motion of the pendulum described in Application 13-A and Examples 13.1 and 13.2, i.e.,

$$u' = f(x, u, v) = v$$
$$v' = g(x, u, v) = -0.3v - \sin(u)$$

with initial conditions

$$u_0 = \pi/2, \quad v_0 = 0$$

The update quantities for this problem are

$$k_1 = hv_i$$
$$m_1 = h(-0.3v_i - \sin(u_i))$$
$$k_2 = h(v_i + 0.5m_1)$$
$$m_2 = h(-0.3(v_i + 0.5m_1) - \sin(u_i + 0.5k_1))$$

The following MATLAB commands can be used to solve this problem.

```
f = inline('v', 'x','u', 'v');
g = inline('-0.3*v - sin(u)', 'x','u', 'v');
[ x, u, v ] = RK2_sys2( f, g, 0, 15, pi/2, 0, 50 );
plot(x,u,x,v,'--')
grid on
```

The motion for the first 15 seconds is shown in Figure 13.4.

Comparing the graphs of the solutions with 50, 100, and 200 subintervals shows that there is virtually no change in the solutions obtained by using more subintervals. This is in marked contrast to the results from Euler's method in Example 13.2.

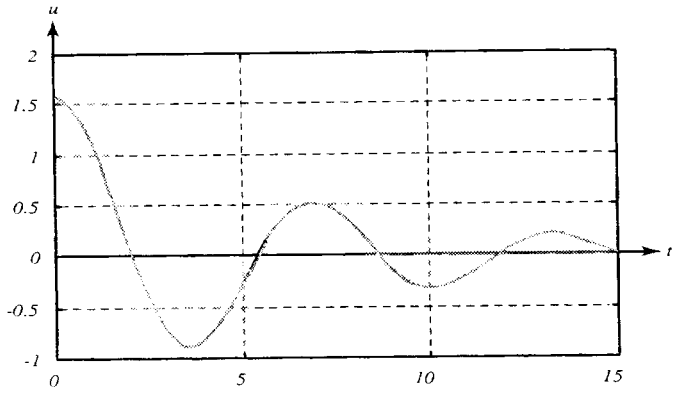

FIGURE 13.4: Oscillations of a pendulum (n = 50, 100, and 200).

EXAMPLE 13.5 Series Dilution Problem

To illustrate the use of Euler's method and the midpoint method for a system of two ODE, consider the concentration of a dye in a two-compartment dilution process. A pure substance flows into the first tank at the same rate that a mixture leaves the first tank and flows into the second tank; the dye leaves the first tank at a rate that is proportional to the concentration. The loss from the first tank becomes the influx to the second tank, which in turn loses fluid at the same rate; thus, the volume of fluid in each tank is constant. The differential equations describing the concentration of dye in the two tanks are

$$\frac{dC_1}{dt} = -\frac{L}{V_1}C_1, \quad \frac{dC_2}{dt} = -\frac{L}{V_2}[C_2 - C_1]$$

Taking $C_1(0) = 0.3$, $C_2(0) = 0$, $L = 2$, $V_1 = 10$, and $V_2 = 5$, we find the concentration in the two tanks for the first 10 minutes of the process.

For comparison, we note that the exact solutions are

$$C_1(t) = C_1(0)\exp(-\frac{L}{V_1}t) = 0.3\exp(-0.2t)$$

$$C_2(t) = \frac{V_1 C_1(0)}{V_1 - V_2}[\exp(-\frac{L}{V_1}t) - \exp(-\frac{L}{V_2}t)] = 0.6[\exp(-0.2t) - \exp(-0.4t)]$$

The values of C_1 and C_2 computed using Euler's method are plotted in Figure 13.5.

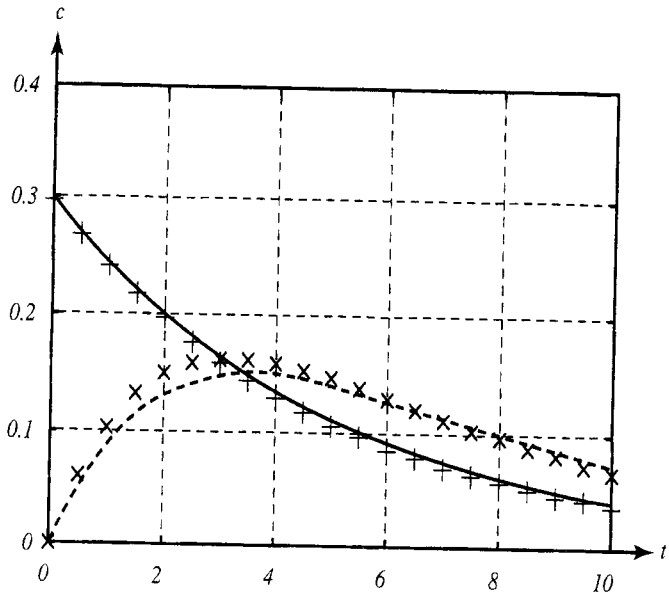

FIGURE 13.5: Concentration of dye in tank 1(+) and tank 2(×), Euler's method.

The computed results using the midpoint method are shown in Figure 13.6.

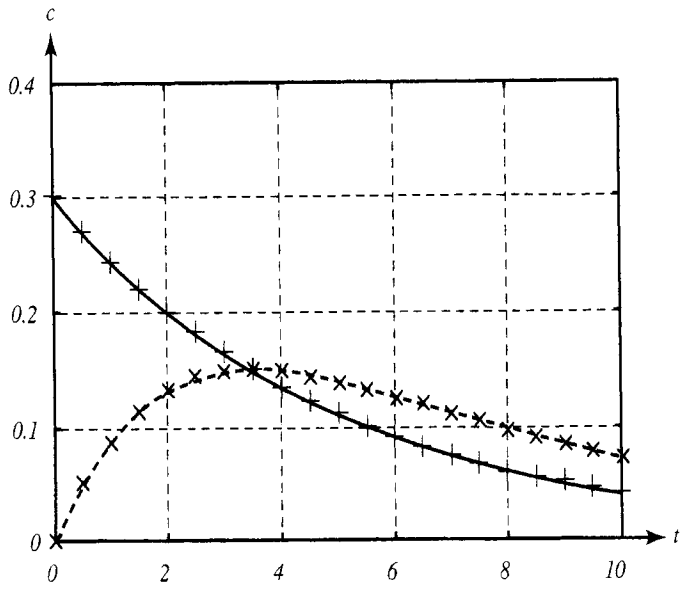

FIGURE 13.6: Concentration of dye in tank 1(+) and tank 2(×), midpoint method.

	Euler		Midpoint		Exact	
x	C_1	C_2	C_1	C_2	C_1	C_2
0.00	0.3000	0.0000	0.3000	0.0000	0.3000	0.0000
0.50	0.2700	0.0600	0.2715	0.0510	0.2715	0.0517
1.00	0.2430	0.1020	0.2457	0.0880	0.2456	0.0890
1.50	0.2187	0.1302	0.2224	0.1139	0.2222	0.1152
2.00	0.1968	0.1479	0.2012	0.1312	0.2011	0.1326
2.50	0.1771	0.1577	0.1821	0.1418	0.1820	0.1432
3.00	0.1594	0.1616	0.1648	0.1472	0.1646	0.1486
3.50	0.1435	0.1611	0.1492	0.1488	0.1490	0.1500
4.00	0.1291	0.1576	0.1350	0.1473	0.1348	0.1485
4.50	0.1162	0.1519	0.1222	0.1438	0.1220	0.1448
5.00	0.1046	0.1448	0.1106	0.1387	0.1104	0.1395
5.50	0.0941	0.1367	0.1001	0.1325	0.0999	0.1332
6.00	0.0847	0.1282	0.0906	0.1257	0.0904	0.1263
6.50	0.0763	0.1195	0.0820	0.1184	0.0818	0.1190
7.00	0.0686	0.1109	0.0742	0.1110	0.0740	0.1115
7.50	0.0618	0.1024	0.0671	0.1037	0.0669	0.1040
8.00	0.0556	0.0943	0.0607	0.0964	0.0606	0.0967
8.50	0.0500	0.0866	0.0550	0.0894	0.0548	0.0896
9.00	0.0450	0.0792	0.0498	0.0826	0.0496	0.0828
9.50	0.0405	0.0724	0.0450	0.0762	0.0449	0.0763
10.00	0.0365	0.0660	0.0407	0.0702	0.0406	0.0702

13.1.2 Euler's Method for Systems

To apply the basic Euler method, $y_{i+1} = y_i + hf(x_i, y_i)$, to the system of ODE

$$\begin{aligned} u_1' &= f_1(x, u_1, u_2, u_3) \\ u_2' &= f_2(x, u_1, u_2, u_3) \\ u_3' &= f_3(x, u_1, u_2, u_3) \end{aligned}$$

we update the function u_1 using f_1, u_2 using f_2, and u_3 using f_3. The same step size h is used for each function. We have

$$\begin{aligned} u_1(i+1) &= u_1(i) + hf_1(x(i), u_1(i), u_2(i), u_3(i)) \\ u_2(i+1) &= u_2(i) + hf_2(x(i), u_1(i), u_2(i), u_3(i)) \\ u_3(i+1) &= u_3(i) + hf_3(x(i), u_1(i), u_2(i), u_3(i)) \end{aligned}$$

EXAMPLE 13.6 Solving a Higher-Order System Using Euler's Method

We apply Euler's method with $n = 2$ to the system of ODE

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= u_3 \\ u_3' &= x + 2u_1 - 3u_2 + 4u_3 \end{aligned}$$

with initial conditions $u_1(0) = 4, u_2(0) = 3$, and $u_3(0) = 2$ on $[0, 1]$. The solution at $i = 1$ corresponds to $x = 0.5$:

$$\begin{aligned} u_1(1) &= u_1(0) + 0.5u_2(0) = 4 + 0.5(3) && = 5.5 \\ u_2(1) &= u_2(0) + 0.5u_3(0) = 3 + 0.5(2) && = 4.0 \\ u_3(1) &= u_3(0) + 0.5(x(0) + 2u_1(0) - 3u_2(0) + 4u_3(0)) \\ &= 2 + 0.5(0 + 2(4) - 3(3) + 4(2)) && = 5.5 \end{aligned}$$

The solution at $i = 2$ corresponds to $x = 1.0$:

$$\begin{aligned} u_1(2) &= u_1(1) + 0.5u_2(1) = 5.5 + 0.5(4) && = 7.50 \\ u_2(2) &= u_2(1) + 0.5u_3(1) = 4 + 0.5(5.5) && = 6.75 \\ u_3(2) &= u_3(1) + 0.5(x(1) + 2u_1(1) - 3u_2(1) + 4u_3(1)) \\ &= 5.5 + 0.5(0.5 + 2(5.5) - 3(4) + 4(5.5)) && = 16.25 \end{aligned}$$

A system of ODE can be expressed compactly in vector notation as

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u})$$

Since the components of the vectors, $\mathbf{u}$ and $\mathbf{f}$, are denoted by subscripts, we indicate the approximate solutions at the i^{th} grid point as $u_1(i)$.

The following MATLAB function for Euler's method is almost identical to the function in Chapter 12, only now $\mathbf{u}$ and $\mathbf{f}$ are vectors.

Euler's Method for a System of ODE

```
function [x , u]  = Euler_sys( f, tspan, u0, n )
% f is an inline function that returns a column vector
a = tspan(1);  b = tspan(2);   h = (b-a)/n;
x = (a:h:b);   u(1, :) = u0;
for i = 1 : n
    u(i+1, :) = u(i, :) + h* f(x(i), u(i, :))';
end
```

EXAMPLE 13.7 Using Euler's Method

The system of ODE given in the previous example can be written as

$$\mathbf{u}' = \mathbf{f}(x, \mathbf{u})$$

where the function $\mathbf{f}(x, \mathbf{u})$ and the initial conditions $\mathbf{u}(0)$ are

$$\mathbf{f} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = \begin{bmatrix} u_2 \\ u_3 \\ x + 2u_1 - 3u_2 + 4u_3 \end{bmatrix}, \quad \mathbf{u}(0) = \begin{bmatrix} u_1(0) \\ u_2(0) \\ u_3(0) \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 2 \end{bmatrix}$$

and `tspan` $= [0, 1]$. The following commands illustrate the use of `Euler_sys`.

```
f = inline('[u(2); u(3); x+2*u(1)-3*u(2)+4*u(3)]', 'x', 'u');
[ x , u ]  = Euler_sys( f, [0, 1], [4, 3, 2], 2);
```

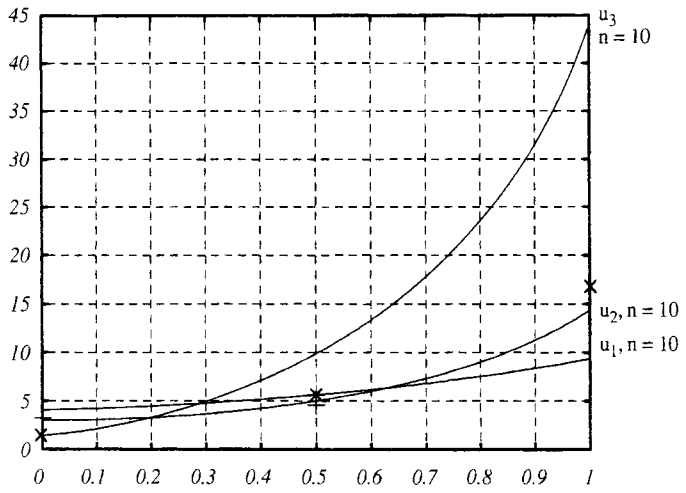

FIGURE 13.7: Solving a system of three ODE using Euler's method.

13.1.3 Runge-Kutta Methods for Systems

The idea in generalizing Runge-Kutta methods for use on systems of equations is the same as for Euler's method; that is, we update each unknown function $u_1, u_2, \ldots$, using the basic Runge-Kutta formulas and the appropriate right-hand-side function $f_1, f_2, \ldots$, from the differential equation for the unknown. We now consider the midpoint method and the classic fourth-order Runge-Kutta method for systems.

Midpoint Method

For the midpoint method, if we denote the two update parameters as k and m, then the basic second-order Runge-Kutta formulas (for the midpoint method) are

$$\begin{aligned} k &= hf(x_i, y_i) \\ m &= hf(x_i + 0.5h, y_i + 0.5k) \\ y_{i+1} &= y_i + m \end{aligned}$$

To apply these formulas to a system, we must compute k and m for each unknown function (i.e., for each component of the unknown vector u). In fact, k must be computed for each unknown before m can be found. We illustrate the process for a system of three ODE:

$$\begin{aligned} u_1' &= f_1(x, u_1, u_2, u_3) \\ u_2' &= f_2(x, u_1, u_2, u_3) \\ u_3' &= f_3(x, u_1, u_2, u_3) \end{aligned}$$

The values of the parameter k for the unknown functions u_1, u_2, and u_3 are

$$\begin{aligned} k_1 &= hf_1(x(i), u_1(i), u_2(i), u_3(i)) \\ k_2 &= hf_2(x(i), u_1(i), u_2(i), u_3(i)) \\ k_3 &= hf_3(x(i), u_1(i), u_2(i), u_3(i)) \end{aligned}$$

Similarly, the values of m are m_1, m_2, and m_3. Of course, to find the value of m for the first ODE, we use f_1; however, we must evaluate f_1 at the appropriate values of x, u_1, u_2, and u_3. Remembering that we are approximating the value of the unknown function employed in evaluating f makes it clear that we approximate each u using its value of k:

$$\begin{aligned} m_1 &= hf_1(x(i) + 0.5h, u_1(i) + 0.5k_1, u_2(i) + 0.5k_2, u_3(i) + 0.5k_3) \\ m_2 &= hf_2(x(i) + 0.5h, u_1(i) + 0.5k_1, u_2(i) + 0.5k_2, u_3(i) + 0.5k_3) \\ m_3 &= hf_3(x(i) + 0.5h, u_1(i) + 0.5k_1, u_2(i) + 0.5k_2, u_3(i) + 0.5k_3) \end{aligned}$$

Finally, the values of the unknown functions at the next grid point are found:

$$\begin{aligned} u_1(i+1) &= u_1(i) + m_1 \\ u_2(i+1) &= u_2(i) + m_2 \\ u_3(i+1) &= u_3(i) + m_3 \end{aligned}$$

EXAMPLE 13.8 Solving a Higher-Order System Using a Runge-Kutta Method

The system

$$u_1' = u_2, \quad u_2' = -\frac{2}{x}u_2, \quad u_3' = u_4, \quad u_4' = -\frac{2}{x}u_4$$

with initial conditions

$$u_1(1) = 10, \quad u_2(1) = 0, \quad u_3(1) = 0, \quad u_4(1) = 1$$

on the interval [1, 2], arises in the solution of the differential equation describing the electrostatic potential between two concentric spheres, one of radius 1 and the other of radius 2.

Using $n = 2$ (and $h = 0.5$), we calculate the values of each component of the solution as a function of x (not the mesh index). First, we find k for each component:

$$k_1 = 0.5(u_2(1)) = 0, \qquad k_2 = 0.5(-\frac{2}{x}u_2(1)) = 0$$
$$k_3 = 0.5(u_4(1)) = 0.5, \quad k_4 = 0.5(-\frac{2}{x}u_4(1)) = -1$$

Next, we find m for each component:

$$m_1 = 0.5(u_2(1) + 0.5k_2) = 0, \qquad m_2 = 0.5(-\frac{2}{1.25})(u_2(1) + 0.5k_2) = 0$$
$$m_3 = 0.5(u_4(1) + 0.5k_4) = 0.25, \quad m_4 = 0.5(-\frac{2}{1.25})(u_4(1) + 0.5k_4) = -0.4$$

The approximate solution at $x = 1.5$ is

$$u_1(1.5) = 10 + 0 \quad = 10, \qquad u_2(1.5) = 0 + 0 \quad = 0$$
$$u_3(1.5) = 0 + 0.25 = 0.25, \quad u_4(1.5) = 1 - 0.4 = 0.6$$

Now we again find k for each component:

$$k_1 = 0.5(u_2(1.5)) = 0, \qquad k_2 = 0.5(-\frac{2}{1.5}u_2(1.5)) = 0$$
$$k_3 = 0.5(u_4(1.5)) = 0.3, \quad k_4 = 0.5(-\frac{2}{1.5}u_4(1.5)) = -0.4$$

Next, we again find m for each component:

$$m_1 = 0.5(u_2(1.5) + 0.5k_2) = 0, \qquad m_2 = 0.5(-\frac{2}{1.75})(u_2(1.5) + 0.5k_2) = 0$$
$$m_3 = 0.5(u_4(1.5) + 0.5k_4) = 0.2, \quad m_4 = 0.5(-\frac{2}{1.75})(u_4(1.5) + 0.5k_4) = -0.2286$$

The approximate solution at $x = 2.0$ is

$$u_1(2.0) = 10 + 0 \quad = 10, \qquad u_2(2.0) = 0 + 0 \quad = 0$$
$$u_3(2.0) = 0.25 + 0.2 = 0.45, \quad u_4(2.0) = 0.6 - 0.2286 = 0.3714$$

The following MATLAB function for the Runge-Kutta midpoint method for systems relies on MATLAB's characteristic of treating all variables as vectors. Thus, the only changes from the Runge-Kutta program (midpoint method) in Chapter 12 that are required are the indexing of the matrix for the solution $\mathbf{u}$ at each step. The function $\mathbf{f}(x, \mathbf{u})$ returns a column vector of values.

Midpoint Method for Systems

```
function [ x , u ]  = RK2_sys( f, tspan, u0, n)
% f is inline function that returns a column vector
% u0 is row vector of initial values
a = tspan(1);      b = tspan(2);
h = (b-a)/n;       x = a : h : b;
u(1,:) = u0;
for i = 1 : n
    k1 = h*f(x(i), u(i, : ) )';
    k2 = h*f(x(i) + h/2, u(i, : )+ 0.5*k1)';
    u(i+1, : ) = u(i, : ) + k2;
end
```

EXAMPLE 13.9 Using the Midpoint Method for a System

We apply the midpoint method to find an approximate solution on the interval $[0, 1]$ of the system of ODE

$$\begin{aligned} u_1' &= u_2 \\ u_2' &= u_3 \\ u_3' &= x + 2u_1 - 3u_2 + 4u_3 \end{aligned}$$

with initial conditions $u_1(0) = 4, \quad u_2(0) = 3, \quad u_3(0) = 2.$

```
%example13_9
% f is a column vector;     u0 is a row vector
f = inline('[u(2); u(3); x+2*u(1)-3*u(2)+4*u(3)]', 'x', 'u');
[ x , u ]  = RK2_sys( f, [0, 1], [4, 3, 2], 2);
disp('      x       u(1)      u(2)     u(3)')
disp([x' u])
```

With $n = 2$, we find the following values for x, u_1, u_2, and u_3:

x	u_1	u_2	u_3
0.0	4.0000	3.0000	2.0000
0.5	5.7500	4.8750	9.1250
1.0	9.3281	13.6719	40.9219

Classic Runge-Kutta Method

We now consider the classic fourth-order Runge-Kutta method for systems of ODE. We denote the four update parameter vectors as $\mathbf{k1}, \mathbf{k2}, \mathbf{k3}$, and $\mathbf{k4}$.

The following MATLAB function assumes that the function $\mathbf{f}(x, \mathbf{u})$ accepts as input the row vector $\mathbf{u}$ and returns a column vector of values for $\mathbf{du}$.

Classic Runge-Kutta Method for Systems

```
function [ x , u ]  = RK4_sys( f, tspan, u0, n)
a = tspan(1);  b = tspan(2);  h = (b-a)/n;  x = (a:h:b)';  u(1,:)=u0
for i = 1 : n
    k1 = h*feval( f, x(i),      u(i,:)       )';
    k2 = h*feval( f, x(i)+h/2, u(i,:)+k1/2 )';
    k3 = h*feval( f, x(i)+h/2, u(i,:)+k2/2 )';
    k4 = h*feval( f, x(i)+h,    u(i,:)+k3    )';
    u(i+1, : ) = u(i, : ) + k1/6 + k2/3 + k3/3 + k4/6;
end
```

EXAMPLE 13.10 Using Runge-Kutta for a Chemical Reaction Problem

The circular reaction described in Application 13-B is described by the ODE:

$$\frac{dA}{dt} = r_3 C - r_1 A, \quad \frac{dB}{dt} = r_1 A - r_2 B, \quad \frac{dC}{dt} = r_2 B - r_3 C$$

The initial values of A, B, and C are

$$A(0) = \frac{1}{1.15} = 0.8696, \quad B(0) = \frac{0.05}{1.15} = 0.0435, \quad C(0) = \frac{0.1}{1.15} = 0.0870$$

Let $r_2 = 2$, $r_3 = 1$, and allow r_1 to change slowly from an initial value of 0.1 according to the linear equation $r_1 = 0.1(t + 1)$.

```
f=inline('[(u(3)-0.1*(t+1)*u(1));(0.1*(t+1)*u(1)-2*u(2));(2*u(2)-u(3))]','t','u');
[ t , u ]  = RK4_sys( f, [0, 40], [0.8696, 0.0435, 0.0870], 40);
```

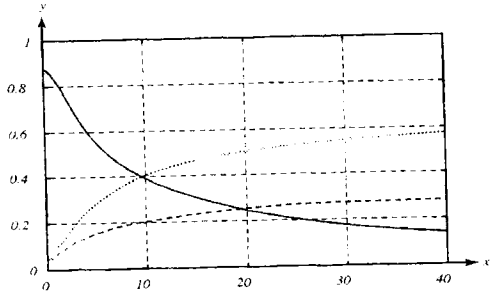

FIGURE 13.8: Concentrations of three reactants in circular chemical reaction. A = solid, B = dashed, C = dotted.

13.1.4 Multistep Methods for Systems

The basic two-step Adams-Bashforth method, in which y_0 is given by the initial condition for the differential equation, y_1 is found from a one-step method, such as a Runge-Kutta technique, and for $i = 1, \ldots, n-1$, and $h = (b-a)/n$,

$$y_{i+1} = y_i + \frac{h}{2}[3f(x_i, y_i) - f(x_{i-1}, y_{i-1})]$$

can be extended for use with a system of three ODE

$$\begin{aligned} u_1' &= f_1(x, u_1, u_2, u_3) \\ u_2' &= f_2(x, u_1, u_2, u_3) \\ u_3' &= f_3(x, u_1, u_2, u_3) \end{aligned}$$

in a similarly straightforward manner. That is, for $i = 1, ..., n-1$:

$$\begin{aligned} u_1(i+1) &= u_1(i) \\ &\quad + \frac{h}{2}[3f_1(x(i), u_1(i), u_2(i), u_3(i)) - f_1(x(i-1), u_1(i-1), u_2(i-1), u_3(i-1))] \\ u_2(i+1) &= u_2(i) \\ &\quad + \frac{h}{2}[3f_2(x(i), u_1(i), u_2(i), u_3(i)) - f_2(x(i-1), u_1(i-1), u_2(i-1), u_3(i-1))] \\ u_3(i+1) &= u_3(i) \\ &\quad + \frac{h}{2}[3f_3(x(i), u_1(i), u_2(i), u_3(i)) - f_3(x(i-1), u_1(i-1), u_2(i-1), u_3(i-1))] \end{aligned}$$

In vector form the differential equation is $\mathbf{u}' = \mathbf{f}(t, \mathbf{u})$, $\mathbf{u}(0) = \mathbf{u}_0$, and the updates (after the first step) are

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}[3\mathbf{f}(t_i, \mathbf{u}_i) - \mathbf{f}(t_{i-1}, \mathbf{u}_{i-1})]$$

EXAMPLE 13.11 Using the Two-Step Adams-Bashforth Method

We again consider the system of ODE

$$u_1' = u_2, \qquad u_2' = u_3, \qquad u_3' = x + 2u_1 - 3u_2 + 4u_3$$

with initial conditions $u_1(0) = 4, \quad u_2(0) = 3, \quad u_3(0) = 2$.
Using the solution at $x = 0.5$ found by the RK midpoint method in Example 13.9, $u(0.5) = [\ 5.7500,\ 4.8750,\ 9.1250]$, we compute $u(1.0)$:

$$\begin{aligned} u_1(1) &= 5.75 + 0.25[3f_1(0.5, 5.75, 4.875, 9.125) - f_1(0.0, 4, 3, 2)] \\ &= 5.75 + 0.25[3(4.875) - 3] = 8.6563 \\ u_2(1) &= 4.875 + 0.25[3f_2(0.5, 5.75, 4.875, 9.125) - f_2(0.0, 4, 3, 2)] \\ &= 4.875 + 0.25[3(9.125) - 2] = 11.2188 \\ u_3(1) &= 9.125 + 0.25[3f_3(0.5, 5.75, 4.875, 9.125) - f_3(0.0, 4, 3, 2)] \\ &= 9.125 + 0.25[3(0.5 + 2(5.75) - 3(4.875) + 4(9.125)) - (0.0 + 2(4) - 3(3) + 4(2))] \\ &= 32.7813 \end{aligned}$$

We now consider the third-order Adams-Bashforth method. The following function allows f to be either an inline function or an m-file function.

Third-Order Adams-Bashforth Method for Systems

```
function [x, u]  = AB3_sys(f, tspan, u0, n)
% initializations
a = tspan(1);          b = tspan(2);          h = (b-a)/n;
k = h/12;              x = (a : h : b)'       u(1, : ) = u0
%  midpoint method to start
for i = 1:2
    z(i, : ) = feval(f, x(i), u(i, : ))'
    k1 = h*z(i, : )
    k2 = h*feval(f, x(i)+0.5*h, u(i, : )+0.5*k1)'
    u(i+1, : ) = u(i, : ) + k2
end
%  third-order A-B method to continue
for i = 3 : n
    z(i, : ) = feval(f, x(i), u(i, : ))';
    u(i+1,:) = u(i,:) +k*(23*z(i,:) -16*z(i-1,:) +5*z(i-2,:));
end
```

EXAMPLE 13.12 Motion of a Pendulum Using Adams-Bashforth

```
% motion of a pendulum using third-order Adams-Bashforth method
% f is a column vector;     u0 is a row vector
f = inline('[u(2); -0.3*u(2)-sin(u(1))]', 'x', 'u');
[t, y] = AB3_sys( f, [0, 15], [pi/2, 0], 50); [t,y]
plot(t, y(:, 1), t, y(:, 2)), grid on
```

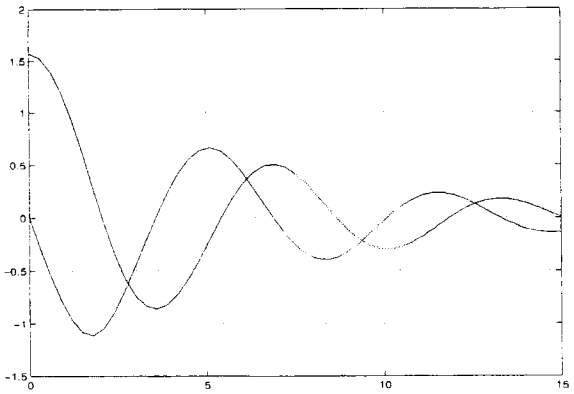

FIGURE 13.9: Position and velocity of a pendulum.

The Adams-Bashforth-Moulton predictor-corrector methods are extended for use with systems of ODE in a similar manner. The third-order method is implemented in the MATLAB function that follows. The dimensions of the vectors u and **u0** depend on the ODE system being solved.

Third-Order Adams-Bashforth-Moulton Method for Systems

```
function [x, u]  = ABM3_sys(f, tspan, u0, n)
a = tspan(1); b = tspan(2);   h = (b-a)/n;  hh = h/12;
x = a : h : b';     u(1, : ) = u0;
for i = 1:2                          %  Use midpoint method to start
    k1 = h*feval(f,a,u(i, : ))';
    k2 = h*feval(f,a+0.5*h,u(i, : )+0.5*k1)';
    u(i+1, : ) = u(1, : ) + k2;
end
for i = 3 : n                        %  Use third-order A-B-M method
    z(i, : ) = feval(f, x(i), u(i, : ))';
    uu(i+1,:) = u(i,:) + hh*(23*z(i,:) - 16*z(i-1,:) + 5*z(i-2,:));
    zz = feval(f, x(i+1), uu(i+1, : ))';
    u(i+1,:) = u(i,:) + hh*(5*zz + 8*z(i,:) - z(i-1,:));
end
```

The following script illustrates the use of the previous function to solve the problem described in the next example.

```
%  S_mass_spring
a = 0;     b = 5;  tspan = [ a  b ];
n = 100;  y0 = [ 0.5  0  0.25  0 ];
% r1 and r2 to include both deflection and length
global  s1   m1   s2   m2
s1 = 100;   m1 = 10;   s2 = 120;   m2 = 2;   r1 = 10;   r2 = 15;
[t, y] = ABM3_sys( 'f_mass_spring', tspan, y0, n);
[ nn, mm] = size(y);   out = [ t  y ];
% down is positive direction, so plot -y
plot(t(1:nn), -(r1+y(1:nn, 1)), '-'  );  hold on
plot(t(1:nn), -(r2+y(1:nn, 3)), '-.' );  grid on
plot([a  b], [0  0]);  hold off
```

The ODE is defined by the following function.

```
function du = f_mass_spring(t, u)
global s1   m1   s2   m2
du = [  u(2);
        -s1*u(1) + s2*(u(3) - u(1))/m1;
        u(4);
        -s2*(u(3) - u(1))/m2  ];
```

EXAMPLE 13.13 Mass-and-Spring System

The vertical displacements of two masses m_1 and m_2 suspended in series by springs with spring constants s_1 and s_2 are given by Hooke's law as a system of two second-order ODE. The displacements are x_1 and x_2. (See Figure 13.10; the displacement of each mass is measured from its equilibrium position, with positive direction downward.) The ODE are

$$m_1 x_1'' = -s_1 x_1 + s_2(x_2 - x_1)$$
$$m_2 x_2'' = -s_2(x_2 - x_1)$$

Converting to a system of first-order ODE gives the differential equations

$$u_1' = u_2$$
$$u_2' = -\frac{s_1}{m_1} u_1 + \frac{s_2}{m_1}(u_3 - u_1)$$
$$u_3' = u_4$$
$$u_4' = -\frac{s_2}{m_2}(u_3 - u_1)$$

with initial conditions

$$u_1(0) = \alpha_1, \quad u_2(0) = \alpha_2, \quad u_3(0) = \alpha_3, \quad u_4(0) = \alpha_4$$

The displacement profiles of a 10-kg mass and a 2-kg mass are illustrated in Figure 13.10. The spring constants are 100 and 120, respectively.

A script for this example and the function defining the ODE are given on the previous page.

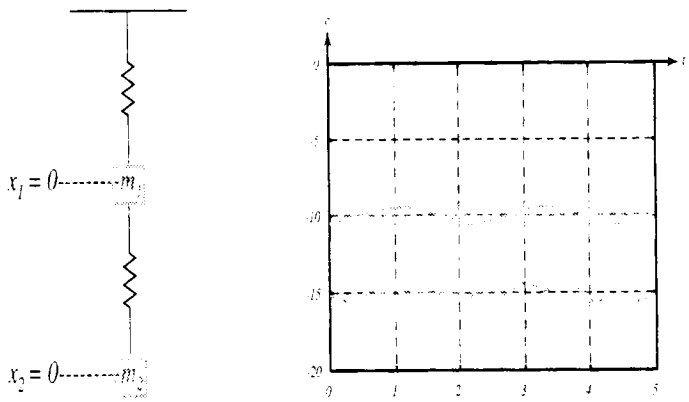

FIGURE 13.10: Two-mass system and displacements.

EXAMPLE 13.14 Motion of a Baseball

Air resistance is one of the factors influencing how far a fly ball travels. In this example, we illustrate the effect of changing assumptions about the form of the air resistance. If a ball is hit with an initial velocity of [100, 45] and is subject to air resistance proportional to its velocity (acting on the horizontal component only), the motion can be found by the MATLAB script that follows.

```
%  S_13_13
a = 0; b = 3; tspan = [a  b]; n = 100; y0 = [ 0  100  3  45 ];
[t, y] = ABM3_sys( 'f_bb', tspan, y0, n);
[ nn, mm] = size(y);  out = [ t  y ];
plot(y(1:nn,1),y(1:nn,3), '-' )
```

Function `f_bb_1` assumes air resistance proportional to velocity in the x-direction only. Function `f_bb_2` assumes air resistance proportional to velocity. The results are shown in Figure 13.11.

```
function dz = f_bb_1(x, z)
dz = [z(2);   -0.1*z(2);    z(4);   -32];

function dz = f_bb_2(x, z)
dz = [z(2);        -0.1*sqrt(z(2)^2+z(4)^2);
      z(4);        -32-0.1*sqrt(z(2)^2+z(4)^2)*sign(z(4))];
```

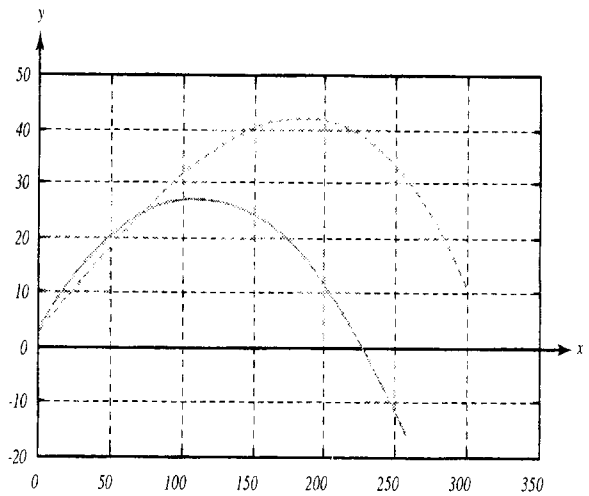

FIGURE 13.11: Flight of a baseball; results for air resistance proportional to velocity in x-direction (dashed) and air resistance proportional to velocity (solid).

To take air resistance proportional to the velocity squared, we take a larger initial velocity so that the ball goes a comparable distance. We use

$$y_0 = [0 \quad 150 \quad 3 \quad 50]$$

Function `f_bb_21` models air resistance proportional to velocity squared in the x-direction only.

```
function dz = f_bb_21(x, z)
dz = [          z(2)
       -0.0025*z(2)^2
                z(4)
                -32 ];
```

Function `f_bb_22` models air resistance proportional to velocity squared.

```
function dz = f_bb_22(x, z)
dz = [                  z(2)
          -0.0025*(z(2)^2 + z(4)^2)
                        z(4)
          -32-0.0025*(z(2)^2 + z(4)^2)*sign(z(4)) ];
```

The results are shown in Figure 13.12.

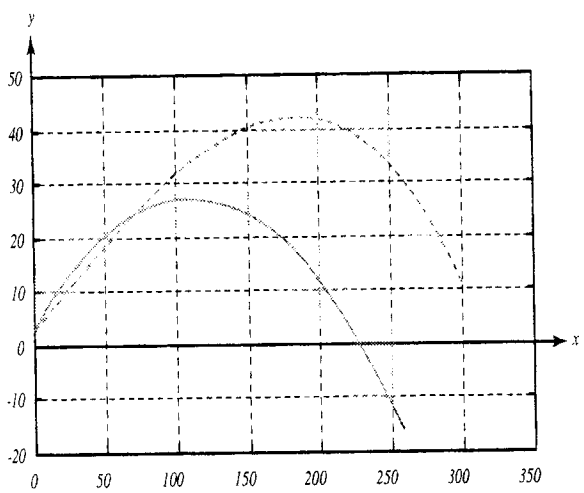

FIGURE 13.12: Flight of a baseball; results for air resistance proportional to velocity squared in x-direction (dashed) and air resistance proportional to velocity squared (solid).

13.1.5 Second-Order ODE

Second-order ODE occur frequently enough in applications to make special methods for solving them of some interest. In this section, we present several such methods.

Predictor-Corrector Methods

We begin by presenting a predictor-corrector method for a second-order ODE of the form $y'' = yg(x)$. We then consider several methods for second-order ODE of the form $y'' = f(x, y)$.

The ODE $y'' = yg(x)$

For the special case when $y'' = f(x, y) = y\,g(x)$, we have

$$y_{i+1} = \frac{b_i y_i - a_{i-1} y_{i-1}}{a_{i+1}}$$

where

$$a_i = 1 - \frac{h^2}{12} g(x_i)$$

and

$$b_i = 2 + \frac{5h^2}{6} g(x_i)$$

(For further details see Froberg, p. 340.)

The ODE $y'' = f(x, y)$

If y' does not appear in the function that defines the ODE, a predictor-corrector method can be constructed for solving the ODE without converting it to a system of first-order ODE. Hamming (p. 214) presents the predictor

$$y^*_{i+1} = 2y_{i-1} - y_{i-3} + \frac{4h^2}{3}(f_i + f_{i-1} + f_{i-2})$$

with local truncation error $16h^6 y^{(6)}/240$ and the corrector

$$y_{i+1} = 2y_i - y_{i-1} + \frac{h^2}{12}(f^*_{i+1} + 10f_i + f_{i-1})$$

with local truncation error $-h^6 y^{(6)}/240$. The error in using the corrector ($Y - y$) is approximately $(1/17)(y^* - y)$.

Several other methods are also considered in Hamming, including some for the case of $y'' = f(x, y, y')$.

Jain attributes the corrector formula above to Numerov; Hamming gives a variation on this method, for the specific case of a linear second-order ODE, as Numerov's method.

The following tables summarize several explicit and several implicit methods for ODE of the form $y'' = f(x, y)$. Note that the order of a stable linear k-step method for $y'' = f(x, y)$ cannot exceed $k + 2$; if k is odd, the order cannot exceed $k + 1$. See Jain, pp. 149–150, for consistency analysis and further discussion of these results.

Explicit k-Step Method for $y'' = f(x, y)$

$$y_{i+1} = 2y_i - y_{i-1} + h^2 \sum_{j=1}^{k} b_j f_{i-j+1}$$

k	b_1	b_2	b_3	b_4	b_5
2	1				
3	$\frac{13}{12}$	$\frac{-2}{12}$	$\frac{1}{12}$		
4	$\frac{14}{12}$	$\frac{-5}{12}$	$\frac{4}{12}$	$\frac{-1}{12}$	
5	$\frac{15}{12}$	$\frac{-9}{12}$	$\frac{10}{12}$	$\frac{-5}{12}$	$\frac{-1}{12}$

Implicit k-Step Method for $y'' = f(x, y)$

$$y_{i+1} = 2y_i - y_{i-1} + h^2 \sum_{j=0}^{k} b_j f_{i-j+1}$$

k	b_0	b_1	b_2	b_3	b_4	b_5
0	1					
2	$\frac{1}{12}$	$\frac{10}{12}$	$\frac{1}{12}$			
4	$\frac{19}{240}$	$\frac{204}{240}$	$\frac{14}{240}$	$\frac{4}{240}$	$\frac{-1}{240}$	
5	$\frac{18}{240}$	$\frac{209}{240}$	$\frac{4}{240}$	$\frac{14}{240}$	$\frac{-6}{240}$	$\frac{1}{240}$

Runge-Kutta Methods

We begin with an example of a Runge-Kutta method for an ODE of the form $y'' = f(x, y)$. We then consider a Runge-Kutta method for the general second-order ODE.

The ODE $y'' = f(x, y)$

For the second-order ODE

$$y'' = f(x, y)$$

with initial conditions $y(a) = y_0$, $y'(a) = z_0$, a Runge-Kutta method with local truncation error $O(h^4)$ in both y and y' can be obtained of the form

$$\begin{aligned} k_1 &= \frac{h^2}{2} f(x_i, y_i, y_i') \\ k_2 &= \frac{h^2}{2} f(x_i + a, y_i + a h y_i' + b k_1) \\ y_{i+1} &= y_i + h y_i' + v_1 k_1 + v_2 k_2 \\ y_{i+1}' &= y_i' + \frac{1}{h} y_i' + w_1 k_1 + w_2 k_2 \end{aligned}$$

if the coefficients satisfy

$$\begin{aligned} v_1 + v_2 &= 1 \\ w_1 + w_2 &= 2 \\ a\, v_2 &= \frac{1}{3} \\ a\, w_2 &= 1 \\ a^2 w_2 &= \frac{2}{3} \\ b\, w_2 &= \frac{2}{3} \end{aligned}$$

A simple solution of these equations is given by the coefficients

$$\begin{aligned} v_1 = v_2 &= \frac{1}{2} \\ a &= \frac{2}{3} \\ b &= \frac{4}{9} \\ w_1 &= \frac{1}{2} \\ w_2 &= \frac{3}{2} \end{aligned}$$

For a discussion of the ODE $y'' = f(y)$, which has applications in celestial mechanics, see Gear, p. 49. Gear refers to this as "Nystrom's formula for special second order equations." The global truncation error is $O(h^3)$ in each component.

The ODE $y'' = f(x, y, y')$

We consider the general second-order ODE

$$y'' = f(x, y, y')$$

with initial conditions $y(a) = y_0,\ y'(a) = z_0$.

A Runge-Kutta method for this problem has the form

$$\begin{aligned} k_1 &= \frac{h^2}{2} f(x_i, y_i, y_i') \\ k_2 &= \frac{h^2}{2} f(x_i + a, y_i + a h y_i' + b k_1, y_i' + \frac{c}{h} k_1) \\ y_{i+1} &= y_i + h y_i' + v_1 k_1 + v_2 k_2 \\ y_{i+1}' &= y_i' + \frac{1}{h} y_i' + w_1 k_1 + w_2 k_2 \end{aligned}$$

where the parameters a, b, c and weights v_1, v_2, w_1, w_2 are determined so that the appropriate Taylor expansions agree (see Jain, p. 81, for details). The local truncation error that can be achieved is $O(h^4)$ in y and $O(h^3)$ in y'.

The equations that must be satisfied by the coefficients are

$$\begin{aligned} v_1 + v_2 &= 1 \\ w_1 + w_2 &= 2 \\ a\, v_2 &= \frac{1}{3} \\ a\, w_2 &= 1 \\ c &= 2a \end{aligned}$$

A simple solution of these equations is given by the coefficients

$$\begin{aligned} v_1 = v_2 &= \frac{1}{2} \\ a = b &= \frac{2}{3} \\ c &= \frac{4}{3} \\ w_1 &= \frac{1}{2} \\ w_2 &= \frac{3}{2} \end{aligned}$$

Derivations are given in Gear, p. 49, and Jain, p. 82.

13.2 STIFF ODE

An ODE in which there is a rapidly decaying transient solution also causes difficulties for numerical solution, requiring an extremely small step size in order to obtain an accurate solution. One source of such equations is in the description of a spring–mass system with large spring constants, hence these problems are known as stiff ODE. Stiff ODE are very common in chemical kinetic studies and occur also in many network analysis and simulation problems.

As an illustration, consider the system

$$u' = 98u + 198v$$
$$v' = -99u - 199v$$

with initial conditions $u(0) = 1$, $v(0) = 0$. The exact solution is

$$u(t) = 2e^{-t} - e^{-100t}$$
$$v(t) = -e^{-t} + e^{-100t}$$

In general, a system of ODE of the form $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ is stiff if some of the eigenvalues of the Jacobian matrix $[\partial \mathbf{f}/\partial \mathbf{y}]$ are large negative. For this example, the eigenvalues are -1 and -100.

A second-order ODE of the form $y'' = f(x)y$ is stiff if $f(x) \gg 0$; this ODE is equivalent to a system of two first-order ODE.

It is also possible for a single first-order ODE to be stiff, as the following problem shows. Consider the ODE $y' = \lambda(y - g(t)) + g'(t)$ with $\lambda \ll 0$ and $g(t)$ a smooth, slowly varying function. The solution is

$$y = (y_0 - g(0))e^{\lambda t} + g(t)$$

The first term in the solution will soon be insignificant compared with $g(t)$, but stability will continue to be governed by $h\lambda$, necessitating a very small step size.

For a system of equations

$$\mathbf{y}' = \mathbf{A}(\mathbf{y} - \mathbf{g}(t)) + \mathbf{g}'(t)$$

the eigenvalues of $\mathbf{A}$ correspond to λ; if all of the eigenvalues have negative real parts, the solution $\mathbf{y}(t) \to \mathbf{g}(t)$ as $t \to \infty$.

To motivate the basic numerical methods for stiff ODE, consider the simple ODE $y' = -cy$, with positive constant c. Euler's method gives

$$y_{k+1} = y_k + hy'_k = y_k - hcy_k = (1 - ch)y_k$$

The method is unstable if $(1 - ch) > 1$ (or equivalently, $h > 2/c$) because in that case, $y \to \infty$ as the iterations progress (whereas the true solution is a decaying exponential function).

On the other hand, if the right-hand side is evaluated at y_{k+1}, we get the implicit method known as the *backward Euler method.*

$$y_{k+1} = y_k + hy'_{k+1} = y_k - hcy_{k+1}$$

Solving for y_{k+1} gives

$$y_{k+1} = \frac{y_k}{1 + ch}$$

which is stable for all h (although a larger h gives less accurate results).

13.2.1 BDF Methods

BDF methods are obtained by replacing y' by a backward differentiation formula. Implicit methods have the form,

$$y'_{i+1} = f(x_{i+1}, y_{i+1})$$

with y'_{i+1} replaced by a backward difference formula.

Backward Euler Method

The simplest method for stiff problems is the backward Euler method; consider $y'_{i+1} = f(x_{i+1}, y_{i+1})$ and replace y'_{i+1} by $(y_{i+1} - y_i)/h$ (the backward difference formula) to get

$$y_{i+1} = y_i + hf_{i+1}$$

This is a single-step method that is first-order accurate.

Two-Step BDF

From the backward difference formula $y'_{i+1} = \dfrac{3y_{i+1} - 4y_i + y_{i-1}}{2h}$, we obtain the two-step BDF

$$y_{i+1} = \frac{4}{3}y_i - \frac{1}{3}y_{i-1} + \frac{2}{3}h\, f_{i+1}$$

BDF k-Step Methods

$$y_{i+1} = \sum_{j=1}^{k} a_j y_{i-j} + b_0\, h\, f_{i+1}$$

k	b_0	a_1	a_2	a_3	a_4	a_5	a_6
1	1	1					
2	$\frac{2}{3}$	$\frac{4}{3}$	$\frac{-1}{3}$				
3	$\frac{6}{11}$	$\frac{18}{11}$	$\frac{-9}{11}$	$\frac{2}{11}$			
4	$\frac{12}{25}$	$\frac{48}{25}$	$\frac{-36}{25}$	$\frac{16}{25}$	$\frac{-3}{25}$		
5	$\frac{60}{137}$	$\frac{300}{137}$	$\frac{-300}{137}$	$\frac{200}{137}$	$\frac{-75}{137}$	$\frac{12}{137}$	
6	$\frac{60}{147}$	$\frac{360}{147}$	$\frac{-450}{147}$	$\frac{400}{147}$	$\frac{-225}{147}$	$\frac{72}{147}$	$\frac{-10}{147}$

The leading error term is $1/(k+1)$ (Celia and Gray, p. 406). The regions of absolute stability decrease for higher k; for $k \geq 7$ the stability properties make the methods unsuitable for stiff problems.

Explicit Gear Methods

Explicit methods have the form:

$$y'_{i+1} = f(x_i, y_i)$$

with y'_{i+1} replaced by a backward difference formula. This gives

$$y_{i+1} = \sum_{j=1}^{k} a_j y_{i-j} + b_1 h f_i$$

k	b_1	a_1	a_2	a_3	a_4	a_5	a_6
1	1	1					
2	2	0	1				
3	3	$\frac{-3}{2}$	3	$\frac{-1}{2}$			
4	4	$\frac{-10}{3}$	$\frac{18}{3}$	$\frac{-6}{3}$	$\frac{1}{3}$		
5	5	$\frac{-65}{12}$	$\frac{120}{12}$	$\frac{-60}{12}$	$\frac{20}{12}$	$\frac{-3}{12}$	
6	6	$\frac{-77}{10}$	$\frac{150}{10}$	$\frac{-100}{10}$	$\frac{50}{10}$	$\frac{-15}{10}$	$\frac{2}{10}$

The consistency requirements for general multistep methods, and conditions for the highest possible order, are as given in Sec. 12.4, but also note that the highest-order methods may have very limited regions of stability, or be unconditionally unstable. See Celia and Gray, p. 405, for further discussion.

An Implicit-Explicit Method

The simplest implicit BDF method has the form $y_{k+1} = y_k + hf(x_{k+1}, y_{k+1})$; the simplest explicit Gear method has the form $y_{k+1} = y_k + hf(x_k, y_k)$. Each of these is first-order accurate. Averaging the two methods gives the second-order method

$$y_{k+1} = y_k + 0.5h(f(x_{k+1}, y_{k+1}) + f(x_k, y_k))$$

If $f(x, y)$ does not depend on y this reduces to the trapezoid rule for integration.

Applying this method to the simple problem $y = -cy$ and solving for y_{k+1} we find

$$y_{k+1} = \frac{2 - ch}{2 + ch} y_k$$

so the method is stable. See Press et al., 1992, for further discussion of these ideas.

13.2.2 Implicit Runge-Kutta Methods

The general form (for an r-stage method) is

$$k_j = hf(x_i + b_j h, y_i + \sum_{s=1}^{r} c_{js} k_s)$$

$$y_{i+1} = y_i + \sum_{s=1}^{r} w_i k_j$$

Note that if this is written in a table, the tableau is no longer lower triangular. Iteration is needed for computation of each k. The extra effort may be justified for stiff ODE, for which explicit methods generally do not give satisfactory results. As with the predictor-corrector methods, the implicit RK (IRK) methods have higher order than the corresponding explicit method.

Derivation (at least for low-order methods) is based on matching Taylor series expansions, as for the explicit formulas.

A one-stage IRK (second-order) has the form

$$\begin{aligned} k_1 &= hf(x_i + b_1 h, y_i + c_{11} k_1) \\ y_{i+1} &= y_i + w_1 k_1 \end{aligned}$$

The coefficients must satisfy

$$\begin{aligned} w_1 &= 1 \\ w_1 b_1 &= \frac{1}{2} \\ w_1 c_{11} &= \frac{1}{2} \end{aligned}$$

So, the second-order implicit Runge-Kutta method is

$$\begin{aligned} k_1 &= hf(x_i + 0.5h, y_i + 0.5k_1) \\ y_{i+1} &= y_i + k_1 \end{aligned}$$

For comparison, recall that an explicit second-order Runge-Kutta method has the general form:

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_i + c_2 h, y_i + a_{21} k_1) \\ y_{i+1} &= y_i + w_1 k_1 + w_2 k_2 \end{aligned}$$

One example, the midpoint method, is given by:

$$\begin{aligned} k_1 &= hf(x_i, y_i) \\ k_2 &= hf(x_i + 0.5h, y_i + 0.5k_1) \\ y_{i+1} &= y_i + k_2 \end{aligned}$$

13.3 STABILITY

In Sec. 12.4 we introduced two key concepts for numerical methods for ordinary differential equations. The concept of consistency deals with the question of whether the numerical method (the difference equation) gives the same solution as the differential equation, for suitably simple problems. The concept of convergence deals with whether the solutions to the difference equation become closer and closer to the solution of the differential equation for smaller and smaller step sizes.

We now consider the third main concept regarding numerical methods for ODE, namely the stability of the method.

The term "stability" is used in a variety of ways in the description of differential equations, and in particular, numerical methods for solving differential equations.

The basic idea is that a differential equation, or a numerical method, is "stable" if a small change in the problem causes only a small change in the solution. For a differential equation, this is can be described in terms of the behavior of the solutions as $t \to \infty$. For a stable ODE the solutions come together for large t; for an unstable ODE the solutions separate as t increases. Thus, for ODE the small change in the problem would be a change in the initial condition.

For a numerical method, the "small change in the problem" would be the inevitable errors that occur because the numerical solution does not solve the differential equation exactly (truncation error) as well as the fact that computations are carried out only to some finite precision (round-off error). Of course these sources of error are present in numerical methods in general, so the question of stability of a numerical method is not restricted to our current consideration of numerical methods for ODE.

In illustrating these ideas it is helpful to consider the very simple ODE-IVP $y' = Ky$, $y(0) = y_0$, for which the solution is $y = y_0 e^{Kt}$. This ODE is stable for $K < 0$, since all solutions approach 0 as $t \to \infty$. The ODE is unstable for $K > 0$, since the solutions go to either ∞ or $-\infty$, depending on the initial condition. For the general first-order ODE $y' = f(t, y)$, the stability of the ODE is controlled by f_y, which is called the Jacobian (denoted J). The ODE is stable if J is negative. For a system of first-order ODE, the stability of the system depends on the eigenvalues of the Jacobian matrix, where $J_{ij} = \partial f_i / \partial y_j$. As we will see, the Jacobian is also important in the analysis of the stability of a numerical method.

We begin our consideration of stability for numerical methods with Euler's method for a single ODE. We denote the true solution of the ODE $y' = f(t, y)$ at $t = t_k$ as $y(t_k)$ and we indicate the computed, approximate solution as y_k. We assume that the ODE is stable ($J < 0$). Now, consider the errors that occur in solving this ODE numerically using Euler's method, $y_{k+1} = y_k + h\, f(t_k, y_k)$. Except at $t = t_0$, we will generally have $y_k \neq y(t_k)$. The *global error* at t_{k+1} (GE_{k+1}) comes from two sources. The first is the *truncation error* (or local error) that results from the truncation in using only the first term of the Taylor series in formulating Euler's method; this is the error that would occur in going from t_k to t_{k+1} even if we had the exact value of y_k. It is just the Taylor series remainder $y''(\eta)h^2/2$, so by reducing the step size h we can reduce the local error. The second part of the error comes from the effect that the error in the solution at t_k has on the solution at t_{k+1}.

namely $h\,f(t_k, y(t_k)) - h\,f(t_k, y_k)$. Using the mean value theorem we find that this is $h\,f(t_k, y(t_k)) - h\,f(t_k, y_k) = h\,f_y(\eta)(y(t_k) - y_k)) = J(\eta)(y(t_k) - y_k))$, where the Jacobian is evaluated at a fixed (but unknown) point. This second part of the error is called the *propagation error*.

The global error at t_{k+1}, $GE_{k+1} = (1+h\,J)GE_k$. The factor $(1+h\,J)$ is called the amplification factor. As long as $|1+h\,J| < 1$ the errors in Euler's method do not grow. This is equivalent to $-2 < h\,J < 0$, which cannot be satisfied if the ODE is unstable, but can be satisfied for appropriate h if the ODE is stable ($J < 0$). Thus, Euler's method is stable for $h < -2/J$, and the stability interval is $-2 < h\,J < 0$.

For a system of first-order ODE, the corresponding requirement for stability is that $||\mathbf{I} + h\mathbf{J}|| < 1$, which will be satisfied if $|1 + h\lambda| < 1$, where λ is the largest eigenvalue of the Jacobian matrix $\mathbf{J}$. Since eigenvalues may be complex, this describes a stability region in the complex plane rather than an interval on the real axis.

The restriction on the step size for Euler's method that comes from the stability interval $(-2, 0)$ is one of the difficulties with using this method, especially for problems in which the Jacobian is large negative (stiff problems).

For the second-order Runge-Kutta methods (with two function evaluations at each step) applied to the stable ODE $y' = Ky$, we define $H = -hK$; it can be shown that the amplification factor is $1 - H + 0.5H^2$, which is less than 1 for $0 < H < 2$. For the general ODE, $y' = f(t, y) \approx f_y(\eta)y$.

One way of looking at Euler's method is that it is the result of numerically integrating $y' = f(t, y)$ from t_k to t_{k+1} using the value of $f(t, y)$ at the left end of the interval. Using this idea, we can derive another simple method for ODE by using $f(t, y)$ at t_{k+1}; this is known as the backward Euler method. A third method comes from using the average of $f(t_k, y_k)$ and $f(t_{k+1}, y_{k+1})$; this gives the trapezoid method. The backward Euler and trapezoid methods are implicit, in that they require information about y_{k+1} in computing y_{k+1}.

Backward Euler: $y_{i+1} = y_i + h f_{i+1}$.
The amplification factor for backward Euler is $(1 - h\,J)^{-1}$.

Trapezoid Method: $y_{i+1} = y_i + (h/2)(f_{i+1} + f_i)$.
The amplification factor for the trapezoid method is $(1 - 0.5h\,J)^{-1}(1 + 0.5h\,J)$.

Implicit methods often have much larger regions of stability than explicit methods. Both the backward Euler and trapezoid methods are absolutely stable (stable for all h).

For an absolutely stable method, the step size must be chosen small enough to ensure sufficient accuracy; the restriction is not based on stability.

In general, a method with a larger region of absolute stability will impose less restriction on the step size h. The region of absolute stability for the Adams-Bashforth second-order method is $-1 < hK < 0$. Note that although this is a smaller region than for Euler's method, the fact that the Adams-Bashforth method is higher order than Euler's method gives it some advantage. The second-order Adams-Moulton method, an implicit method, is absolutely stable for $-\infty < hK < 0$. (See Atkinson, 1989, for further details.)

13.3.1 A-Stable and Stiffly Stable Methods

A numerical method for solving ODE is called *A-stable* if the region of absolute stability includes all of the left half-plane. The backward Euler method, the trapezoid method, and the second-order Adams-Moulton method are examples of A-stable methods. The trapezoidal method is the second-order multistep method with the smallest error constant. Dahlquist (1963) showed that a multistep method that is A-stable cannot have order greater than two.

An alternate definition of A-stability is that a method is A-stable if any solution produced when the method is applied (with fixed step size $h > 0$) to the problem $y' = Ky$ (with $K = \alpha + \beta i$ and $\alpha < 0$) tends to zero as $n \to \infty$.

The q-stage implicit Runge-Kutta method of order $2q$ is A-stable. The lowest-order method of this type is the implicit midpoint rule. Higher-order methods are not widely used, due to the difficulty of solving the implicit equation.

$$k_1 = hf(y_i + 0.5k_1)$$
$$y_{i+1} = y_i + k_1$$

Stiffly Stable Methods

Since A-stability is difficult to achieve, a somewhat less restrictive stability condition, known as stiff stability, is often sufficient. Methods for stiff ODE are implicit and often require iterative techniques for their solution. Newton's method may be used, with the required Jacobian either supplied by the user or generated numerically.

A multistep method

$$y_{i+1} = a_1 y_i + a_2 y_{i-1} + \cdots + a_k y_{i-k+1} + h(b_0 f_{i+1} + b_1 f_i + \cdots + f_{i-k+1})$$

is stiffly stable if, when applied to the problem $y' = \lambda y,\ y(x_0) = y_0$, it is absolutely stable in R_1 and accurate and stable in R_2 (λ is a complex constant with $Re\lambda < 0$). The regions are defined as

$$R_1 = (\lambda h | \mathrm{Re}(\lambda h) \leq D)$$
$$R_2 = (\lambda h | D < \mathrm{Re}(\lambda h) \leq \alpha, |\mathrm{Im}(\lambda h)| < \theta)$$

The BDF k-step methods discussed in the previous section (implicit Gear methods) are stiffly stable.

A(α)-Stability

Another relaxation of the requirements for A-stability is based on the observation that for a multistep method applied to the linear problem $\mathbf{y}' = \mathbf{A}\mathbf{y}$, the stability is determined by the eigenvalues of $\mathbf{A}$, which are fixed. Therefore, it is not necessary to have stability in the entire left half-plane, but rather only in the wedges determined by the eigenvalues, within the region $\pi \pm \alpha$. For $\alpha = \pi/2$ A(α)-stability becomes A-stability.

See Gear, 1971, for further details on A-stability, $A(\alpha)$-stability, and stiffly stable methods. See Butcher, 2003, for a full discussion of implicit Runge-Kutta methods.

13.3.2 Stability in the Limit

Our next consideration is the stability of the difference equation that defines the numerical method. In the notation of multistep methods (Sec. 12.3), a one-step method (such as Euler, or backward Euler, or the Runge-Kutta methods) has the general form

$$y_{i+1} = y_i + h(b_0 f_{i+1} + b_1 f_i)$$

The basic idea for DE is similar to that for ODE — namely, we consider the homogeneous DE ($y_{i+1} = y_i$ for the one-step case) and seek a solution of the form $y_k = r^k$. This gives the characteristic equation, $r^{k+1} = r^k$, which simplifies to $r = 1$.

We now consider the stability of the difference equation that defines a multistep method. We investigate the form of the characteristic equation, and the information that the roots of the characteristic equation provide. Since a convergent numerical method is one in which the solution of the difference equation (DE) and the differential equation (ODE) converge as the step size goes to zero, we are concerned with the behavior of the DE and the ODE for small values of h.

We generalize the notation introduced in Section 12.3 for a two-step method to represent an m-step method as

$$y_{i+1} = a_1 y_i + a_2 y_{i-1} + \cdots + a_m y_{i+1-m} + h(b_0 f_{i+1} + b_1 f_i + \cdots + b_m f_{i+1-m})$$

Difference equations have similar theory to that for differential equations; we consider solutions of the homogeneous DE

$$y_{i+1} = a_1 y_i + a_2 y_{i-1} \ldots + a_m y_{i+1-m}$$

This is an m^{th}-order DE, even if only one of the a_i coefficients is nonzero, because the nonhomogeneous term includes information from m previous steps.

We rewrite this as

$$y_m - a_1 y_{m-1} - a_2 y_{m-2} - \cdots - a_m y_0 = 0$$

We look for solutions of the form $y_k = r^k$ and seek to determine the suitable values of r. Substituting in the DE, we have

$$r^m - a_1 r^{m-1} - a_2 r^{m-2} - \cdots - a_m r^0 = 0$$

The characteristic polynomial is

$$p(r) = r^m - a_1 r^{m-1} - a_2 r^{m-2} - \cdots - a_m$$

If the roots of the characteristic polynomial, $r_1, r_2, \ldots, r_m$, are distinct, then every solution can be expressed as a linear combination

$$y_k = \sum_{i=1}^{m} c_i r_i^k$$

Terminology

The method is called *stable* if all roots of the characteristic polynomial satisfy $|r_i| \leq 1$, and any root with $|r_i| = 1$ is simple. The requirement that $|r_i| \leq 1$ prevents any of the solution components from growing as k increases.

We have seen that any method that is at least first-order accurate must have $a_1 + a_2 + \cdots + a_m = 1$ (consistency requirement), so $r = 1$ is a root. If the other $m - 1$ roots satisfy $|r_i| < 1$, the method is called *strongly stable*.

As we noted at the end of the previous section, the characteristic equation for a one-step method is $r - 1 = 0$, so the only root is $r = 1$ and a one-step method is strongly stable. In particular, since the Runge-Kutta methods do not use information about the computed solution at any previous points (even though RK methods use a more complicated nonhomogeneous term than that discussed for the standard multistep methods), RK methods are one-step methods in the setting of the stability conditions presented above.

If a method is stable, but not strongly stable, it is called *weakly stable*.

A strongly stable method is stable for $y' = \lambda y$ regardless of the sign of λ; a method that is only weakly stable can yield unstable numerical solutions when $\lambda < 0$, as the following example illustrates. The stability analysis based on the roots of the characteristic polynomial reveals the behavior of the method in the limit as the step size becomes arbitrarily small. Even a stable method can exhibit unstable behavior if the step size is too large.

EXAMPLE 13.15 A Weakly Stable Method

Consider the simple two-step method

$$y_{i+1} = y_{i-1} + 2hf(x_i, y_i)$$

and the differential equation

$$y' = -4y, \quad y(0) = 1$$

for which the exact solution is $y = e^{-4x}$. If we perturb the initial condition slightly to $y(0) = 1 + \epsilon$, the solution becomes $y = (1 + \epsilon)e^{-4x}$; the solution is stable since the change in the solution is only $y = \epsilon e^{-4x}$.

However, the numerical method is only weakly stable; the characteristic polynomial is $\lambda^2 - 1 = 0$, which has roots $\lambda = 1$ and $\lambda = -1$. The numerical results for $h = 0.1$ are illustrated in Figure 13.13. Figure 13.14 illustrates the fact that a smaller step size ($h = 0.02$) delays the onset of the instability, but does not prevent it from occurring.

On the other hand, for the differential equation

$$y' = 4y, \quad y(0) = 1$$

the weakly stable method yields acceptable results, even for $h = 0.1$, as illustrated in Figure 13.15.

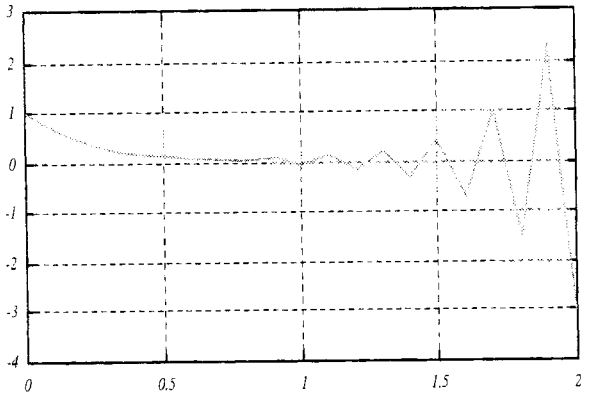

FIGURE 13.13: The solution of $y' = -4y$ with a weakly stable method, $h = 0.1$.

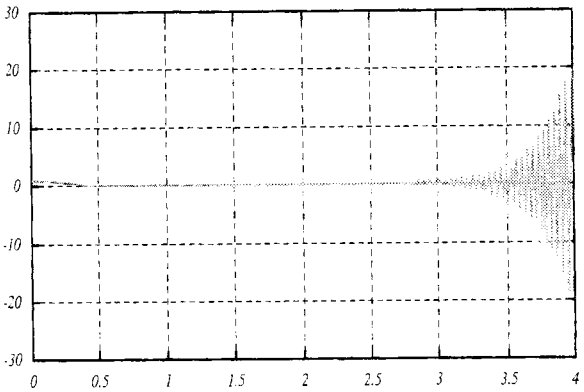

FIGURE 13.14: Instability occurs even with a much smaller step size, $h = 0.02$.

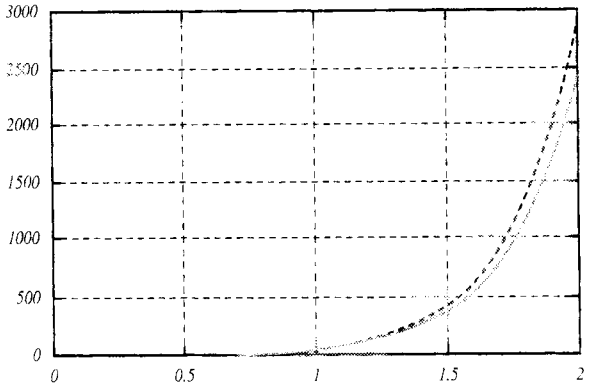

FIGURE 13.15: The solution of $y' = 4y$ with a weakly stable method, $h = 0.1$.

13.4 FURTHER TOPICS

We begin by introducing a few methods that are more advanced than those presented in previous sections. We then summarize MATLAB's built-in functions for stiff ODE. Finally, we illustrate the use of built-in functions for systems of ODE for solving a problem of robot motion.

13.4.1 MATLAB's Methods for Stiff ODE

MATLAB has several built-in functions for solving stiff, or moderately stiff, ODE of the form $y' = f(t, y)$. The basic syntax is the same for all of the ODE solvers. The indications as to the appropriate use of each function are based on the description in the MATLAB help file. The basic call (where `solver` is any of the functions described below) is

```
[ t, Y] = solver(odefun, tspan, y0)
```

where `odefun(t, Y)` is a function (m-file or inline) with input scalar t and column vector `Y` that returns a column vector `f` of function values. To obtain solutions at specified times, define them as `tspan = [t0, t1, ..., tf]`. The stiff solvers use the Jacobian matrix. The default is that the Jacobian is approximated by finite differences. The function `odeset` can be used to specify a function that returns the value of the Jacobian, or the matrix that gives the Jacobian if it is constant.

ode15s

This function gives low to medium accuracy and is recommended if `ode45` is slow because the problem is stiff. It is a variable order solver based on numerical differentiation formulas, or optionally backward differentiation formulas (which are usually less efficient).

ode23s

This function gives a solution with low accuracy; it is appropriate for solving a stiff system of the form `M y' = f(t,y)` with a constant mass matrix `M` and relatively crude error tolerance. `ode23s` is a one-step solver that may be more efficient than `ode15s` at crude tolerances; in addition it can solve some kinds of stiff problems for which `ode15s` is not effective. `ode23s` is based on a modified second-order Rosenbrock formula.

ode23t

This function gives a low-accuracy solution for moderately stiff problems; it gives a solution without numerical damping. It is an implementation of the trapezoidal method.

ode23tb

This function gives a low-accuracy solution for solving stiff systems with crude error tolerances. It is an implementation of an implicit Runge-Kutta formula (TR-BDF2) with the first stage a trapezoid rule and the second stage a second-order backward differentiation formula. The same iteration matrix is used for both stages. This function may be more efficient than `ode15s` at crude tolerance.

Extrapolation Methods

The usefulness of Euler's method is primarily a result of its simplicity, which makes it convenient for hand calculations. It may be used to provide a very few starting values for a multistep method (Sec. 12.3), although the Runge-Kutta methods presented in Sec. 12.2 give more accurate results for only slightly more computational effort.

Extrapolation of Euler's method

Since the total truncation error can be expressed as a series (in powers of h) with coefficients that depend on x and $f(x, y)$, the basic Euler's method can be improved by using it with Richardson extrapolation. We sketch the process here, following the discussion in Jain (1979, pp. 57–58). For a given value of h, the value of $y(x)$ at x_{i+1} is

$$y_{i+1} = y_i + \sum_{j=1}^{\infty} c_i h^i$$

If we denote the computed value as $Y(h)$, we can also find y_{i+1} by taking two steps with $h/2$ or by taking four steps with $h/4$, and so on. Because the error expansion includes all powers of h, rather than only even powers, as we saw before, the extrapolation formula is

$$Y_m(k) = \frac{2^m Y_{m-1}(k+1) - Y_{m-1}(k)}{2^m - 1}$$

where $k = 0, 1, \ldots$ correspond to step sizes of $h/2^0, h/2^1, h/2^2, \ldots$. The parameter m gives the extrapolation level, with Y_0 ($m = 1$) being the values computed directly from Euler's method.

Modified Midpoint Method

In using acceleration, it is highly advantageous to use a method for which the error is strictly an even function of h — i.e., the error expressed in powers of the step size h contains only even powers. In this case, the order of the accelerated method increases by 2 at each stage. Gragg has shown that the error expansion for the modified midpoint method has this form.

```
%  to solve
%  y'  = f(x,y) ,  y(a) = ya,
%  on [a, b], using n-1 intermediate points.
x0 = a
for k = 1 : n
    x(k) = a + k h
end
y0   = ya
y(1) = y0 + h f(x0, y0)
for k = 1 : n-1
    y(k+1) = y(k-1) + 2 h f(x(k), y(k))
end
y(n) = 0.5(y(n) + y(n-1) + h f(x(n), y(n))
```

Bulirsch-Stoer-Deuflhard Method

The Bulirsch-Stoer method uses the idea of acceleration to combine the results of many applications of the midpoint method (with modified first and last steps) to obtain a highly accurate solution with minimal computational effort. Bulirsch-Stoer-Deuflhard extrapolation (BSD method) is based on applying the modified midpoint method using a sequence of n steps (for $n = 2, 4, 6, 8, \ldots$). Each application of the acceleration gives both an improved estimate of the solution (the extrapolated value) and an error estimate. We begin by computing the solution at x_1, starting with the modified midpoint method. Using $n = 2$, we find the solution we denote as z_2; using $n = 4$, we find z_4, and the extrapolated value, $(4z_4 - z_2)/3$. If the error estimate indicates that the extrapolated value is not a sufficiently accurate approximation to the solution at x_1, we take $n = 6$ and extrapolate again, $(4z_6 - z_4)/3$.

The extrapolation continues until the error estimate indicates that the solution is satisfactory (or we reach an upper bound on the number of extrapolation steps allowed). When we have a satisfactory value for y_1, we go to the next subinterval (beginning the solution process with $n = 2$ and $n = 4$).

(For a more detailed discussion, including the use of the error estimate to adjust step size, see Press et al., 1992, pp. 724–732. See also Acton, p. 136.)

Semi-Implicit Extrapolation Methods

The Bulirsch-Stoer method with adaptive step size is a semi-implicit extrapolation method, due to Bader and Deuflhard; it uses a semi-implicit midpoint rule for the extrapolation (Press et al., 1992, pp. 742–747). The method is based on the implicit midpoint rule

$$y_{i+1} - y_{i-1} = 2hf\left(\frac{y_{i+1} + y_{i-1}}{2}\right)$$

The semi-implicit form comes from linearizing the right-hand side. This, together with a special first step (the semi-implicit Euler step), and a smoothing last step, forms the basic algorithm. For computational efficiency, the equations are written in terms of the change in the solution, $D_k = y_{k+1} - y_k$, as follows:

$$D_0 = [I - hJ]^{-1} hf(y_0)$$
$$y_1 = y_0 + D_0$$

For $k = 1, \ldots, m-1$,

$$D_k = D_{k-1} + 2[I - hJ]^{-1}[hf(y_k) - D_{k-1}]$$
$$y_{k+1} = y_k + D_k$$

Finally, to smooth the last step, compute

$$D_m = [I - hJ]^{-1}[hf(y_m) - D_{m-1}]$$
$$y_m = y_m + D_m$$

The matrix $\mathbf{J}$ is the Jacobian matrix, $[\partial f/\partial y]$; m is the number of steps to be used in going across the interval from x_0 to x_1, and $h = (x_1 - x_0)/m$ is the step size.

(See Press et al., 1992, pp. 742–747 for further details. See also Stoer and Bulirsch, p. 491.)

13.4.3 Rosenbrock Methods

The Rosenbrock method is a generalization of the Runge-Kutta methods, for stiff equations. The first of the practical implementations of these methods is due to Kaps and Rentrop (1979). The Rosenbrock methods are competitive with more complicated algorithms for moderate-sized systems (on the order of ten or fewer equations) with moderate accuracy criterion (relative error of 10^{-4} or 10^{-5}).

The general idea is to find a solution of the form

$$y(x+h) = y(x) + \text{corrections}$$

where the corrections are solutions of a set of linear equations involving Runge-Kutta terms and terms depending on the Jacobian of the system. The effectiveness of the method relies on an effective embedded Runge-Kutta method for step-size adjustment.

Several different forms of the update equations have been proposed; the form favored by Press et al. was originally developed by Shampine (1982).

The coefficients from Kaps and Rentrop are given in Stoer and Bulirsch. The methods are intended for stiff autonomous systems of the form $y' = f(y)$.

13.4.4 Multivalue Methods

In solving an ODE, the Adams-type multistep methods use the solution at the current step, y_k, together with information about $y' = f(x, y)$ at one or more previous points. The previous information can also be expressed in terms of various combinations of information about y and its derivatives at one or more previous points. This corresponds to a variety of representations of the underlying interpolating polynomial used for the integration of the right-hand side of the ODE. Representation in terms of scaled higher derivatives of y, all at the previous point, allows for easier step-size adjustment.

We begin with some illustrations of how some well-known multistep methods can be expressed in vector-matrix form, and then show how these methods can be expressed in terms of y and its higher derivatives at one previous point, rather than y and its first derivative at several previous points.

The general approach is to take a multistep method, such as the general Adams-type two-step method

$$y_{i+1} = y_i + h[b_0 f_{i+1} + b_1 f_i + 5 f_{i-1}]$$

and express it in terms of the vector

$$\mathbf{w}_k = [y_k, hy'_k, hy'_{k-1}]$$

and write the method as

$$\begin{aligned} \mathbf{z} &= \mathbf{B}\mathbf{w}_k \\ d &= f(x_{k+1}, \mathbf{z}(1)) - \mathbf{z}(2) \\ \mathbf{w}_{k+1} &= \mathbf{z} + \mathbf{c}d \end{aligned}$$

The Adams-Bashforth Three-Step Method

The Adams-Bashforth method given by the update equation

$$y_{i+1} = y_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}]$$

can be expressed in matrix-vector form as

$$\mathbf{z} = \begin{bmatrix} 1 & 23/12 & -16/12 & 5/12 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} y_k \\ hy'_k \\ hy'_{k-1} \\ hy'_{k-2} \end{bmatrix}$$

$$d = f(x_{k+1}, \mathbf{z}(1)) - \mathbf{z}(2)$$

$$\mathbf{w}_{k+1} = \mathbf{z} + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} d$$

The quantity d represents the amount by which the solution computed as z fails to satisfy the ODE. Adjusting the second component of the solution vector w by that amount assures that the ODE is satisfied (to third order). The following simple MATLAB function carries out these computations. If only the vector y is desired as output, the intermediate computations do not need to be retained.

Adams-Bashforth Third-Order Method, Multivalue Form

```
function [x, y]  = AB3MV(f, tspan, y0, n)
% Adams-Bashforth third-order method, multivalue form
a = tspan(1);  b = tspan(2);  h = (b-a)/n;   x = a:h:b;  y(1) = y0;
for i = 1:2    % Use midpoint method to start
    k1(i) = h*f(x(i), y(i));
    k2(i) = h*f(x(i) + 0.5*h, y(i) + 0.5*k1(i) );
    y(i+1) = y(i) + k2(i);
end
w = [ y(3),  h*f(x(3), y(3)), k1(2),  k1(1)  ]';
% Use 3rd order AB method to continue
B = [ 1  23/12  -16/12  5/12
      0    0       0     0
      0    1       0     0
      0    0       1     0  ];
c = [ 0  1  0  0  ]';
for i = 3 : n
    z = B*w;
    yy = h*f(x(i+1), z(1)) - z(2);
    w = z + c(:)*yy;
    y(i+1) = w(1)
end
```

To retain all components of the solution, modify the code by saving the starting values of **z** (instead of forming the vector **w**), and replace the main loop to use **z**, as follows:

```
.....
%  Use midpoint method to start, then
z(:, 1) = [ y(1),  k1(1),           0,      0      ]';
z(:, 2) = [ y(2),  k1(2),           k1(1),  0      ]';
z(:, 3) = [ y(3),  h*f(x(3), y(3)), k1(2),  k1(1)  ]';
......
for i = 3 : n
    z(:, i+1) = B*z(:, i);
    yy = h*f(x(i+1), z(1,i+1)) - z(2, i+1);
    z(:, i+1) = z(:, i+1) + c(:)*yy;
end
....
```

Adams-Bashforth-Moulton Two-Step Method

We now consider the third-order Adams-Bashforth-Moulton method, in the matrix-vector form. The ABM23 method uses the second-order Adams-Bashforth method as a predictor with the third-order Adams-Moulton method as a corrector:

$$y^*_{i+1} = y_i + \frac{h}{2}[3f_i - 1f_{i-1}]$$
$$y_{i+1} = y_i + \frac{h}{12}[5f^*_{i+1} + 8f_i - f_{i-1}]$$

These can be combined and simplified to give

$$y^*_{i+1} = y_i + \frac{h}{2}[3f_i - 1f_{i-1}]$$
$$y_{i+1} = y^*_{i+1} + \frac{5h}{12}f^*_{i+1} + \frac{5h}{12}[-2f_i + f_{i-1}]$$

In matrix form, this can be written as

$$\mathbf{z} = \mathbf{B}\mathbf{w}_k$$
$$d = f(x_{k+1}, \mathbf{z}(1)) - \mathbf{z}(2)$$
$$\mathbf{w}_{k+1} = \mathbf{z} + \mathbf{c}d$$

where

$$\mathbf{B} = \begin{bmatrix} 1 & 3/2 & -1/2 \\ 0 & 2 & -1 \\ 0 & 1 & 0 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 5/12 \\ 1 \\ 0 \end{bmatrix}$$

Thus we see that the computation of **z** corresponds to the predictor, y^*, the quantities hf_i are the hy_i terms, and the adjustment d incorporates the evaluation of the right-hand side of the ODE at the predicted value.

Adams-Bashforth-Moulton Three-Step Method

We now consider the three-step Adams-Bashforth-Moulton method, in the matrix-vector form. The ABM34 method uses the third-order Adams-Bashforth method as a predictor with the fourth-order Adams-Moulton method as a corrector:

$$y^*_{i+1} = y_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}]$$
$$y_{i+1} = y_i + \frac{h}{24}[9f^*_{i+1} + 19f_i - 5f_{i-1} + 24f_{i-2}]$$

These can be combined and simplified to give

$$y^*_{i+1} = y_i + \frac{h}{12}[23f_i - 16f_{i-1} + 5f_{i-2}]$$
$$y_{i+1} = y^*_{i+1} + \frac{3h}{8}f^*_{i+1} + \frac{3h}{8}[-3f_i + 3f_{i-1} - f_{i-2}]$$

The following simple MATLAB function carries out these computations.

Adams-Bashforth-Moulton Three-Step Method, Multivalue Form

```
function [x, y]  = ABM34MV(f, tspan, y0, n)
% Adams-Bashforth-Moulton three-step method, multivalue form
% (third-order predict, fourth-order correct)
a = tspan(1);      b = tspan(2);         h = (b-a) / n;
x = a : h : b;     y(1) = y0;
%  Use midpoint method to start
for i = 1:2
    k1(i) = h*f(x(i), y(i));
    k2(i) = h*f(x(i) + 0.5*h, y(i) + 0.5*k1(i) );
    y(i+1) = y(i) + k2(i);
end
w = [ y(3),  h*f(x(3), y(3)), k1(2),  k1(1)  ]';
%  Use ABM34 method to continue
 B = [ 1  23/12  -16/12  5/12
       0    3       -3     1
       0    1        0     0
       0    0        1     0  ];
 c = [ 3/8  1  0  0  ]';
for i = 3 : n
    z = B*w;
    yy = h*f(x(i+1), z(1))- z(2);
    w = z + c(:)*yy;
    y(i+1) = w(1);
end
```

Adams-Bashforth-Moulton Four-Step Method

We now consider the four-step Adams-Bashforth-Moulton method, in the matrix-vector form. The ABM45 method uses the fourth-order Adams-Bashforth method as a predictor with the fifth-order Adams-Moulton method as a corrector:

$$y^*_{i+1} = y_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]$$
$$y_{i+1} = y_i + \frac{h}{720}[251f^*_{i+1} + 646f_i - 264f_{i-1} + 106f_{i-2} - 19f_{i-3}]$$

These can be combined and simplified to give

$$y^*_{i+1} = y_i + \frac{h}{24}[55f_i - 59f_{i-1} + 37f_{i-2} - 9f_{i-3}]$$
$$y_{i+1} = y^*_{i+1} + \frac{251h}{720}f^*_{i+1} + \frac{251h}{720}[-4f_i + 6f_{i-1} - 4f_{i-2} + f_{i-3}]$$

The coefficient matrix and correction vector are given in the following MATLAB function, which carries out the computations for this method.

Adams-Bashforth-Moulton Four-Step Method, Multivalue Form

```
function [xx, zz]  = ABM45MV(f, tspan, y0, n)
a = tspan(1);     b = tspan(2);     h = (b-a) / n;
x = a : h : b;    y(1) = y0;
%  Use midpoint method to start
for i = 1:3
    k1(i) = h*f(x(i), y(i));
    k2(i) = h*f(x(i) + 0.5*h, y(i) + 0.5*k1(i) );
    y(i+1) = y(i) + k2(i);
end
w(:, 1) = [ y(1),  k1(1),           0,      0,      0]';
w(:, 2) = [ y(2),  k1(2),           k1(1),  0,      0]';
w(:, 3) = [ y(3),  k1(3),           k1(2),  k1(1),  0 ]';
w(:, 4) = [ y(4),  h*f(x(4), y(4)), k1(3),  k1(2),  k1(1) ]';
A = [  1          55/24       -59/24        37/24        -3/8
       0           4            -6           4            -1
       0           1            0            0             0
       0           0            1            0             0
       0           0            0            1             0   ];
 c = [ 251/720    1    0    0     0 ]';
for i = 4 : n
    z(:, i+1) = A*w(:, i);
    yy = h*f(x(i+1), z(1, i+1)) - z(2, i+1);
    w(:, i+1) = z(:, i+1) + c(:)*yy;
end
xx = x';       zz = w';
```

Change of Basis

The multivalue formulation proposed by Nordsieck (1962) expresses the methods described above in terms of the vector

$$\mathbf{W}_k = [y_k, hy'_k, \frac{h^2}{2}y''_k, \frac{h^3}{6}y'''_k, \ldots]'$$

rather than the vector

$$\mathbf{w}_k = [y_k, hy'_k, hy'_{k-1}, hy'_{k-2}, \ldots]'$$

The transformation can be made using the backward-difference formulas to represent the higher derivatives (at the current point) in terms of the values of the first derivative at the previous points. The first two components of the solution vector are the same in both representations. If a method is given in vector-matrix form in the original basis ($\mathbf{w}_k$) as

$$\begin{aligned} \mathbf{z} &= \mathbf{B}\mathbf{w}_k \\ yy &= h\ f(x(i+1), \mathbf{z}(1)) - \mathbf{z}(2) \\ \mathbf{w}_{k+1} &= \mathbf{z} + \mathbf{c}(:)yy \end{aligned}$$

Then, in the new basis

$$\begin{aligned} \mathbf{Z} &= \mathbf{A}\mathbf{W}_k \\ yy &= h\ f(x(i+1), \mathbf{Z}(1)) - \mathbf{Z}(2) \\ \mathbf{W}_{k+1} &= \mathbf{Z} + \mathbf{C}(:)yy \end{aligned}$$

where

$$\mathbf{Z} = \mathbf{T}\mathbf{z}, \quad \mathbf{A} = \mathbf{T}\mathbf{B}\mathbf{T}^{-1}, \quad \mathbf{C} = \mathbf{T}\mathbf{c}, \quad \mathbf{W}_k = \mathbf{T}\mathbf{w}_k$$

Thus, to convert any of the previous ABM methods to the new basis, we find the transformation matrix (which uses the backward-difference formulas of the appropriate order depending on the order of the method we are converting).

The starting values may be converted in a similar manner, or they can be generated by using a sequence of lower-order methods. Computations using the new matrix and adjustment vector proceed in the same manner as for the original methods.

For example, for ABM23 in the original basis

$$\mathbf{B} = \begin{bmatrix} 1 & 1.5 & -0.5 \\ 0 & 2 & -1 \\ 0 & 1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{c} = \begin{bmatrix} 5/12 \\ 1 \\ 0 \end{bmatrix}$$

The transformation to the Nordsieck basis uses

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & -1/2 \end{bmatrix}$$

$\mathbf{T}$ is also used to convert the starting values to a new basis; $\mathbf{U} = \mathbf{T}^{-1}$.

Thus, if we enter the matrices **B**, **T**, and vector **c** for the method in the original basis, we can compute the matrix and vector for the method in the Nordsieck basis:

```
B = [ 1    1.5    -0.5
      0     2     -1
      0     1      0  ];
T = [ 1     0      0
      0     1      0
      0    1/2   -1/2 ];
U = inv(T) = [ 1     0     0
               0     1     0
               0     1    -2  ]
A = T*B*U = [  1     1     1
               0     1     2
               0     0     1  ];
c = [ 5/12   1   0 ]';
d = T*c  = [ 5/12  1  1/2  ]';
```

Converting a Three-Step Method

To convert a three-step method, the transformation is $\mathbf{W} = \mathbf{T}\mathbf{w}$, where

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 3/4 & -1 & 1/4 \\ 0 & 1/6 & -1/3 & 1/6 \end{bmatrix}$$

This reflects the use of the backward-difference approximations for y_k'' and y_k''' in terms of hy_{k-1}', hy_{k-2}'. The update equation

$$\mathbf{z} = \mathbf{B}\mathbf{w}$$

becomes

$$\mathbf{T}\mathbf{z} = \mathbf{T}\mathbf{B}\mathbf{T}^{-1}\mathbf{T}\mathbf{w}$$

or

$$\mathbf{Z} = \mathbf{A}\mathbf{w}$$

for $\mathbf{Z} = \mathbf{T}\mathbf{z}$ and $\mathbf{A} = \mathbf{T}\mathbf{B}\mathbf{T}^{-1}$. The adjustment vector becomes $\mathbf{C} = \mathbf{T}\mathbf{c}$.

For example, the ABM34 method uses the coefficient matrix

$$\mathbf{B} = \begin{bmatrix} 1 & 23/12 & -16/12 & 5/12 \\ 0 & 3 & -3 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In the new (Nordsieck) basis, the coefficient matrix, $\mathbf{A} = \mathbf{T}\mathbf{B}\mathbf{T}^{-1}$, is

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The MATLAB function for the Adams Bashforth Moulton three-step (third-order predict, fourth-order correct) method becomes:

Adams-Bashforth-Moulton Three-Step Method, Nordsieck Basis

```
function [xx, zz]  = ABM34MVnb(f, tspan, y0, n)
% Adams-Bashforth-Moulton three-step method, multivalue form
a = tspan(1);     b = tspan(2);     h = (b-a) / n;
x = a : h : b;    y(1) = y0;
%  Use midpoint method to start
for i = 1:2
    k1(i) = h*f(x(i), y(i));
    k2(i) = h*f(x(i) + 0.5*h, y(i) + 0.5*k1(i) );
    y(i+1) = y(i) + k2(i);
end
w(:, 1) = [ y(1),  k1(1),           0,      0      ]';
w(:, 2) = [ y(2),  k1(2),           k1(1),  0      ]';
w(:, 3) = [ y(3),  h*f(x(3), y(3)), k1(2),  k1(1)  ]';
% convert starting values to new basis
wn(:, 1) = w(:, 1);
T1 = [ 1    0     0     0
       0    1     0     0
       0   1/2   -1/2   0
       0    0     0     1 ];
wn(:, 2) = T1*w(:, 2);
T2 = [ 1    0     0     0
       0    1     0     0
       0   3/4   -1    1/4
       0   1/6  -1/3   1/6 ];
wn(:, 3) = T2*w(:, 3);
% Use ABM method to continue
A = [ 1    1       1     1
      0    1       2     3
      0    0       1     3
      0    0       0     1   ];
c = [ 3/8  1      3/4   1/6  ]';
for i = 3 : n
    zn(:, i+1) = A*wn(:, i);
    yy = h*f(x(i+1), zn(1, i+1)) - zn(2, i+1);
    wn(:, i+1) = zn(:, i+1) + c(:)*yy;
end
xx = x';      zz = wn';
```

The following MATLAB function converts the ABM four-step method.

Adams-Bashforth-Moulton Four-Step Method, Nordsieck Basis

```
function [xx, zz]  = ABM45MVnb(f, tspan, y0, n)
a = tspan(1);     b = tspan(2);     h = (b-a)/n;     x = a:h:b;     y(1)=y0;
for i = 1:3                         %  Use midpoint method to start
    k1(i) = h*f(x(i), y(i));        k2(i) = h*f(x(i)+0.5*h, y(i)+0.5*k1(i));
    y(i+1) = y(i) + k2(i);
end
w(:, 1) = [ y(1),  k1(1),           0,      0,      0      ]';
w(:, 2) = [ y(2),  k1(2),           k1(1),  0,      0      ]';
w(:, 3) = [ y(3),  k1(3),           k1(2),  k1(1),  0      ]';
w(:, 4) = [ y(4),  h*f(x(4), y(4)), k1(3),  k1(2),  k1(1)  ]';
wn(:, 1) = w(:, 1);                 % convert starting values to new basis
T = [   1   0     0     0  0
        0   1     0     0  0
        0  1/2   -1/2   0  0
        0   0     0     0  0
        0   0     0     0  0 ];              wn(:, 2) = T*w(:, 2);
T = [   1   0     0    0  0
        0   1     0    0  0
        0  3/4   -1   1/4  0
        0  1/6  -1/3  1/6  0
        0   0     0     0  0 ];              wn(:, 3) = T*w(:, 3);
T = [   1   0      0       0      0
        0   1      0       0      0
        0  11/12  -3/2    3/4   -1/6
        0   1/3   -5/6    2/3   -1/6
        0   1/24  -1/8    1/8   -1/24 ];      wn(:, 4) = T*w(:, 4);
% Use four-step, fifth-order ABM method to continue; A = T B inv(T)
 A = [ 1        1      1      1      1
       0        1      2      3      4
       0        0      1      3      6
       0        0      0      1      4
       0        0      0      0      1  ];
c = [ 251/720   1    11/12   1/3   1/24  ]';
for i = 4 : n
    zn(:, i+1) = A*wn(:, i);
    yy = h*f(x(i+1), zn(1, i+1)) - zn(2, i+1);
    wn(:, i+1) = zn(:, i+1) + c(:)*yy;
end
xx = x';        zz = wn';
```

EXAMPLE 13.16 Motion of a Two-Link Robot

We illustrate the use of the function `ode23` to solve the problem of simulating the motion of a two-link planar robot arm. This is an example of a forward dynamics problem — i.e., given the applied joint torques, we solve for the resulting motion of the system.

Attaining a solution involves integrating the equations of motion, which are two nonlinear coupled ODE.

In order to solve the problem, a user-supplied function (called `robot2` in this example) is required to calculate the accelerations of the two-link robot. These are the systems of differential equations to solved by `ode23`. A MATLAB script defines the various parameters needed, calls the function `ode23`, and plots the results. The MATLAB function and script are given following the graphs of the results.

The required parameters for this problem are

- length of each link of the robot arm
- mass of each link
- initial angular position of each link
- desired final angular position of each link
- gravitational constant
- matrices for P-D control with gravity compensation

The moment of inertia for each link is calculated from the mass and length. The unknowns for this system of ODE are

- `q1` position (in radians) of the first link
- `q2` position (in radians) of the second link
- `q1d` velocity of the first link
- `q2d` velocity of the second link

The differential equations make use of several quantities which depend on the masses and lengths of the links (constants) as well as the current positions of the links (found from solving the ODE). These quantities include the mass matrix, the Christoffel matrix, and the gravity vector. From these, the generalized accelerations are computed (which are the values returned by the function for the derivative of the third and fourth components of the vector of unknowns). The derivative of the first component is the third component; the derivative of the second component is the fourth component.

In addition to plotting the position and velocity of each line as a function of time, the torques are also computed and plotted. Figure 13.16 shows the motion of the arm.

(See Spong and Vidyasagar, 1989, for derivation of the equations and description of the coordinate-system conventions.)

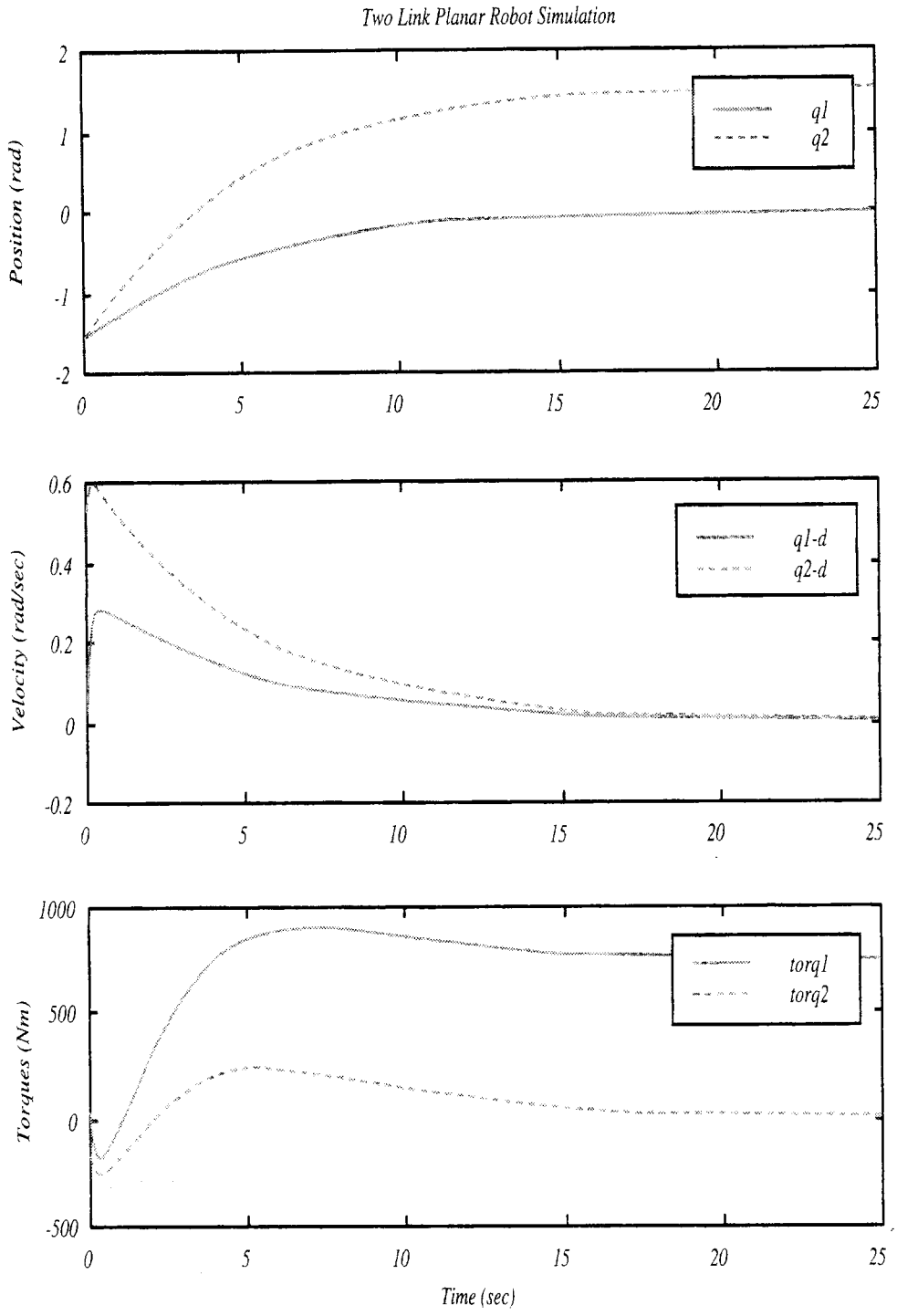

FIGURE 13.16: Motion of a two-link robot arm.

```
% S_robot_motion
% This is the Matlab script file that sets up and runs the planar
% robot simulation with PD control and gravity compensation.
% Thanks to Pierre Larochelle, Florida Institute of Technology, 5-12-97
% The equations of motion are from page 148 of Spong and Vid.
% Make sure the function robot2.m can see the preset values
global L1  Lc1  L2  Lc2  m1  m2  g
global I1  I2  q1_goal  q2_goal  K_p  K_v
% Set up parameters (in metric units)
L1  = 1.0;   Lc1 = 0.5;   L2  = 1.0;  Lc2 = 0.5;   % meters
m1  = 50.0;  m2  = 50.0;                            % kilograms
g   = 9.81;                                         % gravity
% Set up control parameters (in metric units)
q1_start = -pi/2;      q2_start = -pi/2;
q1_goal  = -0.0;       q2_goal  = pi/2;
K_p = 200.0*eye(2);    K_v = 1000.0*eye(2);
% Calculate moment of inertia for a long, slender rod
I1 = m1*L1^2 / 12;    I2 = m2*L2^2 / 12;
% Set up values for ode23 call
t0 = 0;    tf = 25.0;  tspan = [ t0  tf];
y0 = [q1_start   q2_start   0   0]';
[T, Y] = ode23('robot2',tspan, y0);
% make the output easy to look at
t = T;    q1  = Y(:,1);    q2  = Y(:,2);
          q1d = Y(:,3);    q2d = Y(:,4);
% Find the required torques for each instant of time(T)
imax = max(size(T));
for i = 1:1:imax,
    phi1 = (m1*Lc1 + m2*L1)*g*cos(q1(i)) + m2*Lc2*g*cos(q1(i) + q2(i)
    phi2 = m2*Lc2*g*cos(q1(i) + q2(i));
    phi = [phi1 phi2]';
    q_dot = [q1d(i)   q2d(i)]';
    e = [q1(i)-q1_goal   q2(i)-q2_goal]';        ed = q_dot;
    torq = -K_p*e - K_v*ed + phi;
    torq1(i) = torq(1);   torq2(i) = torq(2);
end
% plot position vs time for both link coordinates
subplot(3,1,1);
plot(t,q1,'-',t,q2,':');     legend('q1','q2');
ylabel('Position (rad)');    title('Two Link Planar Robot Simulation')
% plot velocity vs time for both link coordinates
subplot(3,1,2);              plot(t,q1d,'-',t,q2d,':');
legend('q1-d','q2-d');       ylabel('Velocity (rad/sec)')
% plot torques vs time for both link coordinates
subplot(3,1,3);
plot(t,torq1,'-',t,torq2,':');   legend('torq1','torq2');
ylabel('Torques (Nm)');          xlabel('Time (sec)')
```

```
function ydot = robot2(t,y);
% Calculate the accelerations for a two-link planar robot
% Equations of motion for a 2-r planar robot

% make sure this function has access to the global variables
global L1  Lc1  L2  Lc2  m1  m2  g
global I1  I2  q1_goal  q2_goal  K_p  K_v
% Use aliases for input states to match standard robotics notation
q1  = y(1);   q2  = y(2);
q1d = y(3);   q2d = y(4);

% Calculate the mass matrix [M]
d11 = m1*Lc1^2 + m2*(L1^2 + Lc2^2 + 2*L1*Lc2*cos(q2)) + I1 + I2;
d12 = m2*(Lc2^2 + L1*Lc2*cos(q2)) + I2;
d21 = d12;
d22 = m2*Lc2^2 + I2;
M = [ d11    d12
      d21    d22 ];

% Calculate the Christoffel matrix
h = -m2*L1*Lc2*sin(q2);
C = h * [q2d   q2d+q1d
        -q1d       0    ];

% Calculate the gravity vector
phi1 = (m1*Lc1 + m2*L1)*g*cos(q1) + m2*Lc2*g*cos(q1 + q2);
phi2 = m2*Lc2*g*cos(q1 + q2);
phi = [  phi1
         phi2  ];

% Calculate the error vectors
e = [ (q1 - q1_goal)
      (q2 - q2_goal) ];
ed = [  q1d
        q2d  ];
% Calculate control torque vector; P-D Control with gravity compensation
tau = -K_p*e - K_v*ed + phi;
% Calculate the generalized accelerations
qd = [ q1d
       q2d ];
qdd = inv(M)*(tau - C*qd - phi);

% Generate the change in the state vector
ydott(1) = q1d; ydott(2) = q2d; ydott(3) = qdd(1); ydott(4) = qdd(2);
ydot = ydott';
```

13.5 CHAPTER WRAP-UP

SUMMARY

Convert Higher-Order ODE to System of First-Order ODE

The n^{th}-order ODE

$$y^{(n)} = f(x, y, y', y', \ldots, y^{(n-1)})$$

$$y(0) = a_0, \quad y'(0) = a_1, \quad y''(0) = a_2, \quad \ldots, \quad y^{(n-1)}(0) = a_{n-1}$$

becomes a system of first-order ODE by the following change of variables:

$$u_1 = y, \quad u_2 = y', \quad u_3 = y'', \quad \ldots, \quad u_n = y^{(n-1)}$$

The differential equations relating these variables are

$$\begin{aligned} u_1' &= g_1(x, u_1, u_2, u_3, \ldots u_n) = u_2 \\ u_2' &= g_2(x, u_1, u_2, u_3, \ldots u_n) = u_3 \\ u_3' &= g_3(x, u_1, u_2, u_3, \ldots u_n) = u_4 \\ &\vdots \\ u_n' &= g_n(x, u_1, u_2, u_3, \ldots u_n) = f(x, u_1, u_2, u_3, \ldots, u_n) \end{aligned}$$

with the initial conditions

$$u_1(0) = a_0, \quad u_2(0) = a_1, \quad u_3(0) = a_2, \quad \ldots, \quad u_n(0) = a_{n-1}$$

Euler's Method for Two ODE

$$\begin{aligned} u' &= f(x, u, v) \\ v' &= g(x, u, v) \end{aligned}$$

Update u using $f(x, u, v)$ and update v using $g(x, u, v)$:

$$\begin{aligned} u(i+1) &= u(i) + hf(x(i), u(i), v(i)) \\ v(i+1) &= v(i) + hg(x(i), u(i), v(i)) \end{aligned}$$

Midpoint Method for Two ODE

$$u' = f(x, u, v), \quad v' = g(x, u, v)$$

Using k_1 and k_2 to represent the update quantities for u and calling m_1 and m_2 the corresponding quantities for the function v,

$$\begin{aligned} k_1 &= hf(x_i, u_i, v_i) \\ m_1 &= hg(x_i, u_i, v_i) \\ k_2 &= hf(x_i + 0.5h, u_i + 0.5k_1, v_i + 0.5m_1) \\ m_2 &= hg(x_i + 0.5h, u_i + 0.5k_1, v_i + 0.5m_1) \\ u_{i+1} &= u_i + k_2 \\ v_{i+1} &= v_i + m_2 \end{aligned}$$

SUGGESTIONS FOR FURTHER READING

The following texts include discussion of applications of ODE (see Bibliography for full details):

- Ayyub, B. M., and R. H. McCuen, *Numerical Methods for Engineers*, Prentice Hall, 1996.
- Greenberg, M. D., *Advanced Engineering Mathematics*, 2nd ed., Prentice Hall, 1998.
- Greenberg, M. D., *Foundations of Applied Mathematics*, Prentice-Hall, 1978.
- Grossman, S. I., and W. R. Derrick, *Advanced Engineering Mathematics*, Harper & Row, 1988.
- Hanna, O. T., and O. C. Sandall, *Computational Methods in Chemical Engineering*, Prentice Hall, 1995.
- Hildebrand, F. B., *Advanced Calculus for Applications*, 2nd ed., Prentice Hall, 1976.
- Inman, D. J., *Engineering Vibration*, Prentice Hall, 1996.
- Simon, W., *Mathematical Techniques for Biology and Medicine*, Dover, 1986.
- Spong and Vidyasagar, *Robot Dynamics and Control*, John Wiley & Sons, 1989. (See p. 145, exercise 6.4.2.)
- Thomson, W. T., *Theory of Vibrations with Applications*, Prentice Hall, 1993.

For further information on stiff ODE:

- Byrne, G. D., and A. C. Hindmarsh, "Stiff ODE Solvers: A Review of Current and Coming Attractions," *J. Comput. Phys.*, Vol. 70, pp. 1–62, 1987.
- Deuflhard, P., "Recent Progress in Extrapolation Methods for Ordinary Differential Equations," *SIAM Review*, Vol. 27, pp. 505–535, 1985.
- Enright, W. H., T. E. Hull, and B. Lindberg, "Comparing Numerical Methods for Stiff Systems of ODEs," *BIT*, Vol. 15, pp. 10–48, 1975.
- Enright, W. H., and J. D. Pryce, "Two FORTRAN Packages for Assessing Initial Value Methods," *ACM Transactions on Mathematical Software*, Vol. 13, pp. 1–27, 1978.
- Gear, C. W., *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, 1971.
- Shampine, L. F., "Ill-Conditioned Matrices and the Integration of Stiff ODEs," *J. Comput. Appl. Math*, Vol. 48, pp. 279–292, 1993.

For further information on stability see:

- Kahaner, D., C. Moler, and S. Nash, *Numerical Methods and Software*, Sec. 8.5., Prentice Hall, 1988.