

ORDINARY DIFFERENTIAL EQUATIONS

Differential equations are mathematical descriptions of how the variables and their derivatives (rates of change) with respect to one or more independent variable affect each other in a dynamical way. Their solutions show us how the dependent variable(s) will change with the independent variable(s). Many problems in natural sciences and engineering fields are formulated into a scalar differential equation or a vector differential equation—that is, a system of differential equations.

In this chapter, we look into several methods of obtaining the numerical solutions to ordinary differential equations (ODEs) in which all dependent variables (x) depend on a single independent variable (t). First, the initial value problems (IVPs) will be handled with several methods including Runge–Kutta method and predictor–corrector methods in Sections 6.1 to 6.5. The final section (Section 6.6) will introduce the shooting method and the finite difference method for solving the two-point boundary value problem (BVP). ODEs are called an IVP if the values $x(t_0)$ of dependent variables are given at the initial point t_0 of the independent variable, while they are called a BVP if the values $x(t_0)/x(t_f)$ are given at the initial/final points t_0 and t_f .

6.1 EULER'S METHOD

When talking about the numerical solutions to ODEs, everyone starts with the Euler's method, since it is easy to understand and simple to program. Even though its low accuracy keeps it from being widely used for solving ODEs, it gives us a

clue to the basic concept of numerical solution for a differential equation simply and clearly. Let's consider a first-order differential equation:

$$y'(t) + a y(t) = r \quad \text{with } y(0) = y_0 \tag{6.1.1}$$

It has the following form of analytical solution:

$$y(t) = \left(y_0 - \frac{r}{a}\right) e^{-at} + \frac{r}{a} \tag{6.1.2}$$

which can be obtained by using a conventional method or the Laplace transform technique [K-1, Chapter 5]. However, such a nice analytical solution does not exist for every differential equation; even if it exists, it is not easy to find even by using a computer equipped with the capability of symbolic computation. That is why we should study the numerical solutions to differential equations.

Then, how do we translate the differential equation into a form that can easily be handled by computer? First of all, we have to replace the derivative $y'(t) = dy/dt$ in the differential equation by a numerical derivative (introduced in Chapter 5), where the step-size h is determined based on the accuracy requirements and the computation time constraints. Euler's method approximates the derivative in Eq. (6.1.1) with Eq. (5.1.2) as

$$\begin{aligned} \frac{y(t+h) - y(t)}{h} + a y(t) &= r \\ y(t+h) &= (1 - ah)y(t) + hr \quad \text{with } y(0) = y_0 \end{aligned} \tag{6.1.3}$$

and solves this difference equation step-by-step with increasing t by h each time from $t = 0$.

$$\begin{aligned} y(h) &= (1 - ah)y(0) + hr = (1 - ah)y_0 + hr \\ y(2h) &= (1 - ah)y(h) + hr = (1 - ah)^2 y_0 + (1 - ah)hr + hr \\ y(3h) &= (1 - ah)y(2h) + hr = (1 - ah)^3 y_0 + \sum_{m=0}^2 (1 - ah)^m hr \\ &\dots \end{aligned} \tag{6.1.4}$$

This is a numeric sequence $\{y(kh)\}$, which we call a numerical solution of Eq. (6.1.1).

To be specific, let the parameters and the initial value of Eq. (6.1.1) be $a = 1$, $r = 1$, and $y_0 = 0$. Then, the analytical solution (6.1.2) becomes

$$y(t) = 1 - e^{-at} \tag{6.1.5}$$

```

%nm610: Euler method to solve a 1st-order differential equation
clear, clf
a = 1; r = 1; y0 = 0; tf = 2;
t = [0:0.01:tf]; yt = 1 - exp(-a*t); %Eq.(6.1.5): true analytical solution
plot(t,yt,'k'), hold on
klasts = [8 4 2]; hs = tf./klasts;
y(1) = y0;
for itr = 1:3 %with various step size h = 1/8,1/4,1/2
    klast = klasts(itr); h = hs(itr); y(1)=y0;
    for k = 1:klast
        y(k + 1) = (1 - a*h)*y(k) +h*r; %Eq.(6.1.3):
        plot([k - 1 k]*h,[y(k) y(k+1)],'b', k*h,y(k+1),'ro')
        if k < 4, pause; end
    end
end
end
    
```

and the numerical solution (6.1.4) with the step-size $h = 0.5$ and $h = 0.25$ are as listed in Table 6.1 and depicted in Fig. 6.1. We make a MATLAB program “nm610.m”, which uses Euler’s method for the differential equation (6.1.1), actually solving the difference equation (6.1.3) and plots the graphs of the numerical solutions in Fig. 6.1. The graphs seem to tell us that a small step-size helps reduce the error so as to make the numerical solution closer to the (true) analytical solution. But, as will be investigated thoroughly in Section 6.2, it is only partially true. In fact, a too small step-size not only makes the computation time longer (proportional as $1/h$), but also results in rather larger errors due to the accumulated round-off effect. This is why we should look for other methods to decrease the errors rather than simply reduce the step-size.

Euler’s method can also be applied for solving a first-order vector differential equation

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}) \quad \text{with } \mathbf{y}(t_0) = \mathbf{y}_0 \tag{6.1.6}$$

which is equivalent to a high-order scalar differential equation. The algorithm can be described by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k) \quad \text{with } \mathbf{y}(t_0) = \mathbf{y}_0 \tag{6.1.7}$$

Table 6.1 A Numerical Solution of the Differential Equation (6.1.1) Obtained by the Euler’s Method

t	$h = 0.5$	$h = 0.25$
0.25		$y(0.25) = (1 - ah)y_0 + hr = 1/4 = 0.25$
0.50	$y(0.50) = (1 - ah)y_0 + hr = 1/2 = 0.5$	$y(0.50) = (3/4)y(0.25) + 1/4 = 0.4375$
0.75		$y(0.75) = (3/4)y(0.50) + 1/4 = 0.5781$
1.00	$y(1.00) = (1/2)y(0.5) + 1/2 = 3/4 = 0.75$	$y(1.00) = (3/4)y(0.75) + 1/4 = 0.6836$
1.25		$y(1.25) = (3/4)y(1.00) + 1/4 = 0.7627$
1.50	$y(1.50) = (1/2)y(1.0) + 1/2 = 7/8 = 0.875$	$y(1.50) = (3/4)y(1.25) + 1/4 = 0.8220$
...

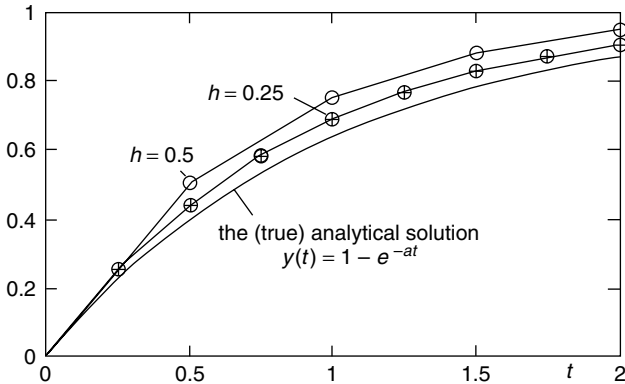


Figure 6.1 Examples of numerical solution obtained by using the Euler’s method.

and is cast into the MATLAB routine “ode_Euler()”.

```
function [t,y] = ode_Euler(f,tspan,y0,N)
%Euler's method to solve vector differential equation y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin<4 | N <= 0, N = 100; end
if nargin<3, y0 = 0; end
h = (tspan(2) - tspan(1))/N; %stepsize
t = tspan(1)+[0:N]'*h; %time vector
y(1,:) = y0(:)'; %always make the initial value a row vector
for k = 1:N
    y(k + 1,:) = y(k,:) + h*feval(f,t(k),y(k,:)); %Eq.(6.1.7)
end
```

6.2 HEUN’S METHOD: TRAPEZOIDAL METHOD

Another method of solving a first-order vector differential equation like Eq. (6.1.6) comes from integrating both sides of the equation.

$$\begin{aligned}
 \mathbf{y}'(t) &= \mathbf{f}(t, \mathbf{y}), & \mathbf{y}(t)|_{t_k}^{t_{k+1}} &= \mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y}) dt \\
 \mathbf{y}(t_{k+1}) &= \mathbf{y}(t_k) + \int_{t_k}^{t_{k+1}} \mathbf{f}(t, \mathbf{y}) dt & \text{with } \mathbf{y}(t_0) &= \mathbf{y}_0
 \end{aligned}
 \tag{6.2.1}$$

If we assume that the value of the (derivative) function $\mathbf{f}(t,\mathbf{y})$ is constant as $\mathbf{f}(t_k,\mathbf{y}(t_k))$ within one time step $[t_k,t_{k+1})$, this becomes Eq. (6.1.7) (with $h = t_{k+1} - t_k$), amounting to Euler’s method. If we use the trapezoidal rule (5.5.3), it becomes

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{2} \{ \mathbf{f}(t_k, \mathbf{y}_k) + \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \}
 \tag{6.2.2}$$

```
function [t,y] = ode_Heun(f,tspan,y0,N)
%Heun method to solve vector differential equation y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin<4 | N <= 0, N = 100; end
if nargin<3, y0 = 0; end
h = (tspan(2) - tspan(1))/N; %stepsize
t = tspan(1)+[0:N]'*h; %time vector
y(1,:) = y0(:)'; %always make the initial value a row vector
for k = 1:N
    fk = feval(f,t(k),y(k,:)); y(k+1,:) = y(k,:)+h*fk; %Eq.(6.2.3)
    y(k+1,:) = y(k,:) +h/2*(fk +feval(f,t(k+1),y(k+1,:))); %Eq.(6.2.4)
end
```

But, the right-hand side (RHS) of this equation has \mathbf{y}_{k+1} , which is unknown at t_k . To resolve this problem, we replace the \mathbf{y}_{k+1} on the RHS by the following approximation:

$$\mathbf{y}_{k+1} \cong \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k) \tag{6.2.3}$$

so that it becomes

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{2}\{\mathbf{f}(t_k, \mathbf{y}_k) + \mathbf{f}(t_{k+1}, \mathbf{y}_k + h\mathbf{f}(t_k, \mathbf{y}_k))\} \tag{6.2.4}$$

This is Heun’s method, which is implemented in the MATLAB routine “ode_Heun()”. It is a kind of predictor-and-corrector method in that it predicts the value of \mathbf{y}_{k+1} by Eq. (6.2.3) at t_k and then corrects the predicted value by Eq. (6.2.4) at t_{k+1} . The truncation error of Heun’s method is $O(h^2)$ (proportional to h^2) as shown in Eq. (5.6.1), while the error of Euler’s method is $O(h)$.

6.3 RUNGE-KUTTA METHOD

Although Heun’s method is a little better than the Euler’s method, it is still not accurate enough for most real-world problems. The fourth-order Runge-Kutta (RK4) method having a truncation error of $O(h^4)$ is one of the most widely used methods for solving differential equations. Its algorithm is described below.

$$\mathbf{y}_{k+1} = \mathbf{y}_k + \frac{h}{6}(\mathbf{f}_{k1} + 2\mathbf{f}_{k2} + 2\mathbf{f}_{k3} + \mathbf{f}_{k4}) \tag{6.3.1}$$

where

$$\mathbf{f}_{k1} = \mathbf{f}(t_k, \mathbf{y}_k) \tag{6.3.2a}$$

$$\mathbf{f}_{k2} = \mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{f}_{k1}h/2) \tag{6.3.2b}$$

$$\mathbf{f}_{k3} = \mathbf{f}(t_k + h/2, \mathbf{y}_k + \mathbf{f}_{k2}h/2) \tag{6.3.2c}$$

$$\mathbf{f}_{k4} = \mathbf{f}(t_k + h, \mathbf{y}_k + \mathbf{f}_{k3}h) \tag{6.3.2d}$$

```

function [t,y] = ode_RK4(f,tspan,y0,N,varargin)
%Runge-Kutta method to solve vector differential eqn y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
if nargin < 4 | N <= 0, N = 100; end
if nargin < 3, y0 = 0; end
y(1,:) = y0(:)'; %make it a row vector
h = (tspan(2) - tspan(1))/N; t = tspan(1)+[0:N]*h;
for k = 1:N
    f1 = h*feval(f,t(k),y(k,:),varargin{:}); f1 = f1(:)'; %(6.3.2a)
    f2 = h*feval(f,t(k) + h/2,y(k,:) + f1/2,varargin{:}); f2 = f2(:)';%(6.3.2b)
    f3 = h*feval(f,t(k) + h/2,y(k,:) + f2/2,varargin{:}); f3 = f3(:)';%(6.3.2c)
    f4 = h*feval(f,t(k) + h,y(k,:) + f3,varargin{:}); f4 = f4(:)'; %(6.3.2d)
    y(k + 1,:) = y(k,:) + (f1 + 2*(f2 + f3) + f4)/6; %Eq. (6.3.1)
end

%nm630: Heun/Euer/RK4 method to solve a differential equation (d.e.)
clear, clf
tspan = [0 2];
t = tspan(1)+[0:100]*(tspan(2) - tspan(1))/100;
a = 1; yt = 1 - exp(-a*t); %Eq.(6.1.5): true analytical solution
plot(t,yt,'k'), hold on
df61 = inline('-y + 1','t','y'); %Eq.(6.1.1): d.e. to be solved
y0 = 0; N = 4;
[t1,ye] = oed_Euler(df61,tspan,y0,N);
[t1,yh] = ode_Heun(df61,tspan,y0,N);
[t1,yr] = ode_RK4(df61,tspan,y0,N);
plot(t,yt,'k', t1,ye,'b:', t1,yh,'b:', t1,yr,'r:')
plot(t1,ye,'bo', t1,yh,'b+', t1,yr,'r*')
N = 1e3; %to estimate the time for N iterations
tic, [t1,ye] = ode_Euler(df61,tspan,y0,N); time_Euler = toc
tic, [t1,yh] = ode_Heun(df61,tspan,y0,N); time_Heun = toc
tic, [t1,yr] = ode_RK4(df61,tspan,y0,N); time_RK4 = toc

```

Equation (6.3.1) is the core of RK4 method, which may be obtained by substituting Simpson's rule (5.5.4)

$$\int_{t_k}^{t_{k+1}} f(x) dx \cong \frac{h'}{3}(f_k + 4f_{k+1/2} + f_{k+1}) \quad \text{with } h' = \frac{x_{k+1} - x_k}{2} = \frac{h}{2} \quad (6.3.3)$$

into the integral form (6.2.1) of differential equation and replacing $f_{k+1/2}$ with the average of the successive function values $(f_{k2} + f_{k3})/2$. Accordingly, the RK4 method has a truncation error of $O(h^4)$ as Eq. (5.6.2) and thus is expected to work better than the previous two methods.

The fourth-order Runge–Kutta (RK4) method is cast into the MATLAB routine “ode_RK4()”. The program “nm630.m” uses this routine to solve Eq. (6.1.1) with the step size $h = (t_f - t_0)/N = 2/4 = 0.5$ and plots the numerical result together with the (true) analytical solution. Comparison of this result with those of Euler's method (“ode_Euler()”) and Heun's method (“ode_Heun()”) is given in Fig. 6.2, which shows that the RK4 method is better than Heun's method, while Euler's method is the worst in terms of accuracy with the same step-size. But,

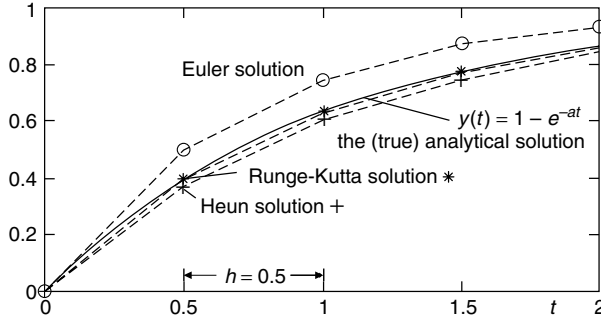


Figure 6.2 Numerical solutions for a first-order differential equation.

in terms of computational load, the order is reversed, because Euler’s method, Heun’s method, and the RK4 method need 1, 2, and 4 function evaluations (calls) per iteration, respectively.

(cf) Note that a function call takes much more time than a multiplication and thus the number of function calls should be a criterion in estimating and comparing computational time.

The MATLAB built-in routines “ode23()” and “ode45()” implement the Runge–Kutta method with an adaptive step-size adjustment, which uses a large/small step-size depending on whether $f(t)$ is smooth or rough. In Section 6.4.3, we will try applying these routines together with our routines to solve a differential equation for practice rather than for comparison.

6.4 PREDICTOR-CORRECTOR METHOD

6.4.1 Adams–Bashforth–Moulton Method

The Adams–Bashforth–Moulton (ABM) method consists of two steps. The first step is to approximate $\mathbf{f}(t, \mathbf{y})$ by the (Lagrange) polynomial of degree 4 matching the four points

$$\{(t_{k-3}, \mathbf{f}_{k-3}), (t_{k-2}, \mathbf{f}_{k-2}), (t_{k-1}, \mathbf{f}_{k-1}), (t_k, \mathbf{f}_k)\}$$

and substitute the polynomial into the integral form (6.2.1) of differential equation to get a predicted estimate of \mathbf{y}_{k+1} .

$$\mathbf{p}_{k+1} = \mathbf{y}_k + \int_0^h l_3(t) dt = \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) \tag{6.4.1a}$$

The second step is to repeat the same work with the updated four points

$$\{(t_{k-2}, \mathbf{f}_{k-2}), (t_{k-1}, \mathbf{f}_{k-1}), (t_k, \mathbf{f}_k), (t_{k+1}, \mathbf{f}_{k+1})\} \quad (\mathbf{f}_{k+1} = \mathbf{f}(t_{k+1}, \mathbf{p}_{k+1}))$$

to get a corrected estimate of \mathbf{y}_{k+1} .

$$\mathbf{c}_{k+1} = \mathbf{y}_k + \int_0^h l'_3(t) dt = \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}_{k+1}) \quad (6.4.1b)$$

The coefficients of Eqs. (6.4.1a) and (6.4.1b) can be obtained by using the MATLAB routines “lagranp()” and “polyint()”, each of which generates Lagrange (coefficient) polynomials and integrates a polynomial, respectively. Let’s try running the program “ABMc.m”.

```
>>abmc
cAP = -3/8    37/24   -59/24    55/24
cAC =  1/24    -5/24    19/24     3/8
```

```
%ABMc.m
% Predictor/Corrector coefficients in Adams–Bashforth–Moulton method
clear
format rat
[1,L] = lagranp([-3 -2 -1 0],[0 0 0 0]); %only coefficient polynomial L
for m = 1:4
    iL = polyint(L(m,:)); %indefinite integral of polynomial
    cAP(m) = polyval(iL,1)-polyval(iL,0); %definite integral over [0,1]
end
cAP %Predictor coefficients
[1,L] = lagranp([-2 -1 0 1],[0 0 0 0]); %only coefficient polynomial L
for m = 1:4
    iL = polyint(L(m,:)); %indefinite integral of polynomial
    cAC(m) = polyval(iL,1) - polyval(iL,0); %definite integral over [0,1]
end
cAC %Corrector coefficients
format short
```

Alternatively, we write the Taylor series expansion of \mathbf{y}_{k+1} about t_k and that of \mathbf{y}_k about t_{k+1} as

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_k + \frac{h^2}{2}\mathbf{f}'_k + \frac{h^3}{3!}\mathbf{f}^{(2)}_k + \frac{h^4}{4!}\mathbf{f}^{(3)}_k + \frac{h^5}{5!}\mathbf{f}^{(4)}_k + \dots \quad (6.4.2a)$$

$$\mathbf{y}_k = \mathbf{y}_{k+1} - h\mathbf{f}_{k+1} + \frac{h^2}{2}\mathbf{f}'_{k+1} - \frac{h^3}{3!}\mathbf{f}^{(2)}_{k+1} + \frac{h^4}{4!}\mathbf{f}^{(3)}_{k+1} - \frac{h^5}{5!}\mathbf{f}^{(4)}_{k+1} + \dots$$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}_{k+1} - \frac{h^2}{2}\mathbf{f}'_{k+1} + \frac{h^3}{3!}\mathbf{f}^{(2)}_{k+1} - \frac{h^4}{4!}\mathbf{f}^{(3)}_{k+1} + \frac{h^5}{5!}\mathbf{f}^{(4)}_{k+1} - \dots \quad (6.4.2b)$$

and replace the first, second, and third derivatives by their difference approximations.

$$\begin{aligned}
 \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{f}_k + \frac{h^2}{2} \left(\frac{-\frac{1}{3}\mathbf{f}_{k-3} + \frac{3}{2}\mathbf{f}_{k-2} - 3\mathbf{f}_{k-1} + \frac{11}{6}\mathbf{f}_k}{h} + \frac{1}{4}h^3\mathbf{f}_k^{(4)} + \dots \right) \\
 &\quad + \frac{h^3}{3!} \left(\frac{-\mathbf{f}_{k-3} + 4\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 2\mathbf{f}_k}{h^2} + \frac{11}{12}h^2\mathbf{f}_k^{(4)} + \dots \right) \\
 &\quad + \frac{h^4}{4!} \left(\frac{-\mathbf{f}_{k-3} + 3\mathbf{f}_{k-2} - 3\mathbf{f}_{k-1} + \mathbf{f}_k}{h^3} + \frac{3}{2}h\mathbf{f}_k^{(4)} + \dots \right) + \frac{h^5}{120}\mathbf{f}_k^{(4)} + \dots \\
 &= \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) + \frac{251}{720}h^5\mathbf{f}_k^{(4)} + \dots \\
 &\stackrel{(6.4.1a)}{\approx} \mathbf{p}_{k+1} + \frac{251}{720}h^5\mathbf{f}_k^{(4)} \tag{6.4.3a}
 \end{aligned}$$

$$\begin{aligned}
 \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{f}_{k+1} - \frac{h^2}{2} \left(\frac{-\frac{1}{3}\mathbf{f}_{k-2} + \frac{3}{2}\mathbf{f}_{k-1} - 3\mathbf{f}_k + \frac{11}{6}\mathbf{f}_{k+1}}{h} + \frac{1}{4}h^3\mathbf{f}_{k+1}^{(4)} + \dots \right) \\
 &\quad + \frac{h^3}{3!} \left(\frac{-\mathbf{f}_{k-2} + 4\mathbf{f}_{k-1} - 5\mathbf{f}_k + 2\mathbf{f}_{k+1}}{h^2} + \frac{11}{12}h^2\mathbf{f}_{k+1}^{(4)} + \dots \right) \\
 &\quad - \frac{h^4}{4!} \left(\frac{-\mathbf{f}_{k-2} + 3\mathbf{f}_{k-1} - 3\mathbf{f}_k + \mathbf{f}_{k+1}}{h^3} + \frac{3}{2}h\mathbf{f}_{k+1}^{(4)} + \dots \right) + \frac{h^5}{120}\mathbf{f}_{k+1}^{(4)} + \dots \\
 &= \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}_{k+1}) - \frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} + \dots \\
 &\stackrel{(6.4.1b)}{\approx} \mathbf{c}_{k+1} - \frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} \tag{6.4.3b}
 \end{aligned}$$

These derivations are supported by running the MATLAB program ‘‘ABMc1.m’’.

```

%ABMc1.m
%another way to get the ABM coefficients together with the error term
clear, format rat
for i = 1:3, [ci,erri] = difapx(i,[-3 0]); c(i,:) = ci; err(i) = erri;
end
cAP = [0 0 0 1]+[1/2 1/6 1/24]*c, errp = -[1/2 1/6 1/24]*err' + 1/120
CAC = [0 0 0 1]+[-1/2 1/6 -1/24]*c, errc = -[-1/2 1/6 -1/24]*err' + 1/120
format short
    
```

From these equations and under the assumption that $\mathbf{f}_{k+1}^{(4)} \cong \mathbf{f}_k^{(4)} \cong K$, we can write the predictor/corrector errors as

$$E_{P,k+1} = \mathbf{y}_{k+1} - \mathbf{p}_{k+1} \approx \frac{251}{720}h^5\mathbf{f}_k^{(4)} \cong \frac{251}{720}Kh^5 \tag{6.4.4a}$$

$$E_{C,k+1} = \mathbf{y}_{k+1} - \mathbf{c}_{k+1} \approx -\frac{19}{720}h^5\mathbf{f}_{k+1}^{(4)} \cong -\frac{19}{720}Kh^5 \tag{6.4.4b}$$

We still cannot use these formulas to estimate the predictor/corrector errors, since K is unknown. But, from the difference between these two formulas

$$E_{P,k+1} - E_{C,k+1} = \mathbf{c}_{k+1} - \mathbf{p}_{k+1} \cong \frac{270}{720}Kh^5 \equiv \frac{270}{251}E_{P,k+1} \equiv -\frac{270}{19}E_{C,k+1} \quad (6.4.5)$$

we can get the practical formulas for estimating the errors as

$$E_{P,k+1} = \mathbf{y}_{k+1} - \mathbf{p}_{k+1} \cong \frac{251}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \quad (6.4.6a)$$

$$E_{C,k+1} = \mathbf{y}_{k+1} - \mathbf{c}_{k+1} \cong -\frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \quad (6.4.6b)$$

These formulas give us rough estimates of how close the predicted/corrected values are to the true value and so can be used to improve them as well as to adjust the step-size.

$$\mathbf{p}_{k+1} \rightarrow \mathbf{p}_{k+1} + \frac{251}{270}(\mathbf{c}_k - \mathbf{p}_k) \Rightarrow \mathbf{m}_{k+1} \quad (6.4.7a)$$

$$\mathbf{c}_{k+1} \rightarrow \mathbf{c}_{k+1} - \frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \Rightarrow \mathbf{y}_{k+1} \quad (6.4.7b)$$

These modification formulas are expected to reward our efforts that we have made to derive them.

The Adams–Bashforth–Moulton (ABM) method with the modification formulas can be described by Eqs. (6.4.1a), (6.4.1b), and (6.4.7a), (6.4.7b) summarized below and is cast into the MATLAB routine “ode_ABM()”. This scheme needs only two function evaluations (calls) per iteration, while having a truncation error of $O(h^5)$ and thus is expected to work better than the methods discussed so far. It is implemented by the MATLAB built-in routine “ode113()” with many additional sophisticated techniques.

(Adams–Bashforth–Moulton method with modification formulas)

$$\text{Predictor: } \mathbf{p}_{k+1} = \mathbf{y}_k + \frac{h}{24}(-9\mathbf{f}_{k-3} + 37\mathbf{f}_{k-2} - 59\mathbf{f}_{k-1} + 55\mathbf{f}_k) \quad (6.4.8a)$$

$$\text{Modifier: } \mathbf{m}_{k+1} = \mathbf{p}_{k+1} + \frac{251}{270}(\mathbf{c}_k - \mathbf{p}_k) \quad (6.4.8b)$$

$$\text{Corrector: } \mathbf{c}_{k+1} = \mathbf{y}_k + \frac{h}{24}(\mathbf{f}_{k-2} - 5\mathbf{f}_{k-1} + 19\mathbf{f}_k + 9\mathbf{f}(t_{k+1}, \mathbf{m}_{k+1})) \quad (6.4.8c)$$

$$\mathbf{y}_{k+1} = \mathbf{c}_{k+1} - \frac{19}{270}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \quad (6.4.8d)$$

```

function [t,y] = ode_ABM(f,tspan,y0,N,KC,varargin)
%Adams-Bashforth-Moulton method to solve vector d.e. y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
% using the modifier based on the error estimate depending on KC = 1/0
if nargin < 5, KC = 1; end %with modifier by default
if nargin < 4 | N <= 0, N = 100; end %default maximum number of iterations
y0 = y0(:)'; %make it a row vector
h = (tspan(2) - tspan(1))/N; %step size
tspan0 = tspan(1)+[0 3]*h;
[t,y] = rk4(f,tspan0,y0,3,varargin{:}); %initialize by Runge-Kutta
t = [t(1:3)' t(4):h:tspan(2)]';
for k = 1:4, F(k,:) = feval(f,t(k),y(k,:),varargin{:}); end
p = y(4,:); c = y(4,:); KC22 = KC*251/270; KC12 = KC*19/270;
h24 = h/24; h241 = h24*[1 -5 19 9]; h249 = h24*[-9 37 -59 55];
for k = 4:N
    p1 = y(k,:) + h249*F; %Eq.(6.4.8a)
    m1 = pk1 + KC22*(c-p); %Eq.(6.4.8b)
    c1 = y(k,)+ ...
        h241*[F(2:4,:); feval(f,t(k + 1),m1,varargin{:})]; %Eq.(6.4.8c)
    y(k + 1,:) = c1 - KC12*(c1 - p1); %Eq.(6.4.8d)
    p = p1; c = c1; %update the predicted/corrected values
    F = [F(2:4,:); feval(f,t(k + 1),y(k + 1,:),varargin{:})];
End
    
```

6.4.2 Hamming Method

```

function [t,y] = ode_Ham(f,tspan,y0,N,KC,varargin)
% Hamming method to solve vector d.e. y'(t) = f(t,y(t))
% for tspan = [t0,tf] and with the initial value y0 and N time steps
% using the modifier based on the error estimate depending on KC = 1/0
if nargin < 5, KC = 1; end %with modifier by default
if nargin < 4 | N <= 0, N = 100; end %default maximum number of iterations
if nargin < 3, y0 = 0; end %default initial value
y0 = y0(:)'; end %make it a row vector
h = (tspan(2)-tspan(1))/N; %step size
tspan0 = tspan(1)+[0 3]*h;
[t,y] = ode_RK4(f,tspan0,y0,3,varargin{:}); %Initialize by Runge-Kutta
t = [t(1:3)' t(4):h:tspan(2)]';
for k = 2:4, F(k - 1,:) = feval(f,t(k),y(k,:),varargin{:}); end
p = y(4,:); c = y(4,:); h34 = h/3*4; KC11 = KC*112/121; KC91 = KC*9/121;
h312 = 3*h*[-1 2 1];
for k = 4:N
    p1 = y(k - 3,:) + h34*(2*(F(1,:) + F(3,:)) - F(2,:)); %Eq.(6.4.9a)
    m1 = p1 + KC11*(c - p); %Eq.(6.4.9b)
    c1 = (-y(k - 2,:) + 9*y(k,)) + ...
        h312*[F(2:3,:); feval(f,t(k + 1),m1,varargin{:})]/8; %Eq.(6.4.9c)
    y(k+1,:) = c1 - KC91*(c1 - p1); %Eq.(6.4.9d)
    p = p1; c = c1; %update the predicted/corrected values
    F = [F(2:3,:); feval(f,t(k + 1),y(k + 1,:),varargin{:})];
end
    
```

(Hamming method with modification formulas)

$$\text{Predictor: } \mathbf{p}_{k+1} = \mathbf{y}_{k-3} + \frac{4h}{3}(2\mathbf{f}_{k-2} - \mathbf{f}_{k-1} + 2\mathbf{f}_k) \quad (6.4.9a)$$

$$\text{Modifier: } \mathbf{m}_{k+1} = \mathbf{p}_{k+1} + \frac{112}{121}(\mathbf{c}_k - \mathbf{p}_k) \quad (6.4.9b)$$

$$\text{Corrector: } \mathbf{c}_{k+1} = \frac{1}{8}\{9\mathbf{y}_k - \mathbf{y}_{k-2} + 3h(-\mathbf{f}_{k-1} + 2\mathbf{f}_k + \mathbf{f}(t_{k+1}, \mathbf{m}_{k+1}))\} \quad (6.4.9c)$$

$$\mathbf{y}_{k+1} = \mathbf{c}_{k+1} - \frac{9}{121}(\mathbf{c}_{k+1} - \mathbf{p}_{k+1}) \quad (6.4.9d)$$

In this section, we introduce just the algorithm of the Hamming method [H-1] summarized in the box above and the corresponding routine “ode_Ham()”, which is another multistep predictor–corrector method like the Adams–Bashforth–Moulton (ABM) method.

This scheme also needs only two function evaluations (calls) per iteration, while having the error of $O(h^5)$ and so is comparable with the ABM method discussed in the previous section.

6.4.3 Comparison of Methods

The major factors to be considered in evaluating/comparing different numerical methods are the accuracy of the numerical solution and its computation time. In this section, we will compare the routines “ode_RK4()”, “ode_AB()”, “ode_Ham()”, “ode23()”, “ode45()”, and “ode113()” by trying them out on the same differential equations, hopefully to make some conjectures about their performances. It is important to note that the evaluation/comparison of numerical methods is not so simple because their performances may depend on the characteristic of the problem at hand. It should also be noted that there are other factors to be considered, such as stability, versatility, proof against run-time error, and so on. These points are being considered in most of the MATLAB built-in routines.

The first thing we are going to do is to validate the effectiveness of the modifiers (Eqs. (6.4.8b,d) and (6.4.9b,d)) in the ABM (Adams–Bashforth–Moulton) method and the Hamming method. For this job, we write and run the program “nm643_1.m” to get the results depicted in Fig. 6.3 for the differential equation

$$y'(t) = -y(t) + 1 \quad \text{with } y(0) = 0 \quad (6.4.10)$$

which was given at the beginning of this chapter. Fig. 6.3 shows us an interesting fact that, although the ABM method and the Hamming method, even without modifiers, are theoretically expected to have better accuracy than the RK4 (fourth-order Runge–Kutta) method, they turn out to work better than RK4 only with modifiers. Of course, it is not always the case, as illustrated in Fig. 6.4, which

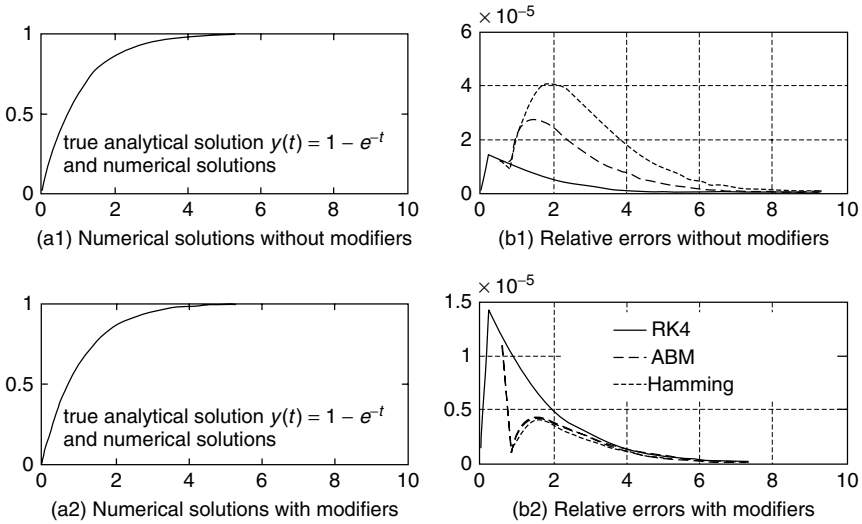


Figure 6.3 Numerical solutions and their errors for the differential equation $y'(t) = -y(t) + 1$.

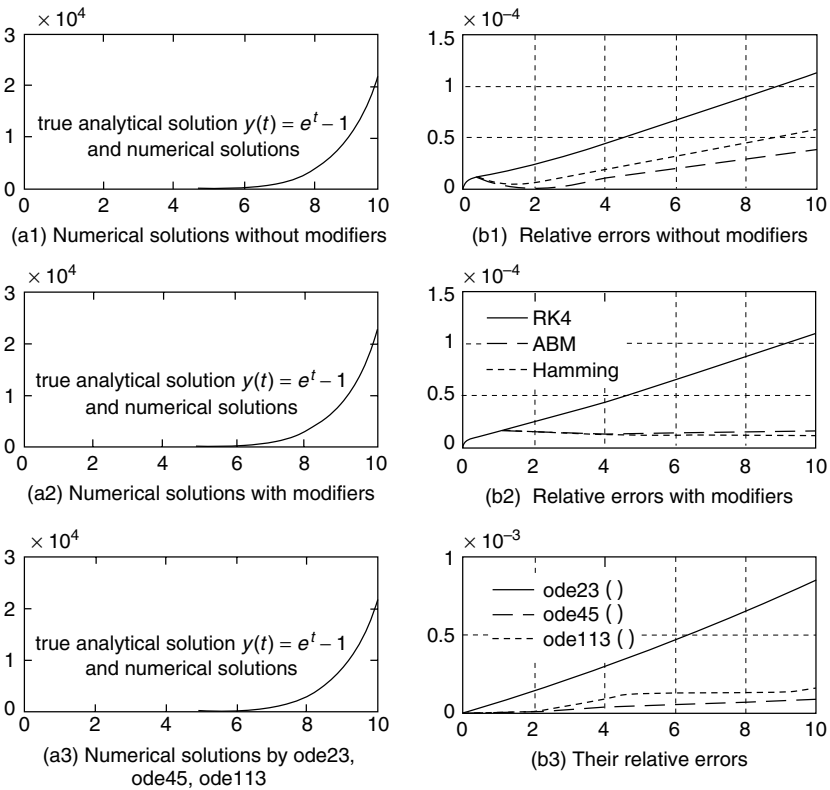


Figure 6.4 Numerical solutions and their errors for the differential equation $y'(t) = y(t) + 1$.

we obtained by applying the same routines to solve another differential equation

$$y'(t) = y(t) + 1 \quad \text{with } y(0) = 0 \quad (6.4.11)$$

where the true analytical solution is

$$y(t) = e^t - 1 \quad (6.4.12)$$

```
%nm643_1: RK4/Adams/Hamming method to solve a differential eq
clear, clf
t0 = 0; tf = 10; y0 = 0; %starting/final time, initial value
N = 50; %number of segments
df643 = inline('-y+1','t','y'); %differential equation to solve
f643 = inline('1-exp(-t)','t'); %true analytical solution
for KC = 0:1
    tic, [t1,yR] = ode_RK4(df643,[t0 tf],y0,N); tR = toc
    tic, [t1,yA] = ode_ABM(df643,[t0 tf],y0,N,KC); tA = toc
    tic, [t1,yH] = ode_Ham(df643,[t0 tf],y0,N,KC); tH = toc
    yt1 = f643(t1); %true analytical solution to plot
    subplot(221 + KC*2) %plot analytical/numerical solutions
    plot(t1,yt1,'k', t1,yR,'k', t1,yA,'k--', t1,yH,'k:')
    tmp = abs(yt1)+eps; l_t1 = length(t1);
    eR = abs(yR - yt1)./tmp; e_R=norm(eR)/l_t1
    eA = abs(yA - yt1)./tmp; e_A=norm(eA)/l_t1
    eH = abs(yH - yt1)./tmp; e_H=norm(eH)/l_t1
    subplot(222 + KC*2) %plot relative errors
    plot(t1,eR,'k', t1,eA,'k--', t1, eH,'k:')
end
```

```
%nm643_2: ode23()/ode45()/ode113() to solve a differential eq
clear, clf
t0 = 0; tf = 10; y0 = 0; N = 50; %starting/final time, initial value
df643 = inline('y + 1','t','y'); %differential equation to solve
f643 = inline('exp(t) - 1','t'); %true analytical solution
tic, [t1,yR] = ode_RK4(df643,[t0 tf],y0,N); time(1) = toc;
tic, [t1,yA] = ode_ABM(df643,[t0 tf],y0,N); time(2) = toc;
yt1 = f643(t1);
tmp = abs(yt1)+ eps; l_t1 = length(t1);
eR = abs(yR-yt1)./tmp; err(1) = norm(eR)/l_t1;
eA = abs(yA-yt1)./tmp; err(2) = norm(eA)/l_t1;
options = odeset('RelTol',1e-4); %set the tolerance of relative error
tic, [t23,yode23] = ode23(df643,[t0 tf],y0,options); time(3) = toc;
tic, [t45,yode45] = ode45(df643,[t0 tf],y0,options); time(4) = toc;
tic, [t113,yode113] = ode113(df643,[t0 tf],y0,options); time(5) = toc;
yt23 = f643(t23); tmp = abs(yt23) + eps;
eode23 = abs(yode23-yt23)./tmp; err(3) = norm(eode23)/length(t23);
yt45 = f643(t45); tmp = abs(yt45) + eps;
eode45 = abs(yode45 - yt45)./tmp; err(4) = norm(eode45)/length(t45);
yt113 = f643(t113); tmp = abs(yt113) + eps;
eode113 = abs(yode113 - yt113)./tmp; err(5) = norm(eode113)/length(t113);
subplot(221), plot(t23,yode23,'k', t45,yode45,'b', t113,yode113,'r')
subplot(222), plot(t23,eode23,'k', t45,eode45,'b--', t113,eode113,'r:')
err, time
```

Table 6.2 Results of Applying Several Routines to solve a Simple Differential Equation

	ode_RK4()	ode_ABM()	ode_Ham()	ode23()	ode45()	ode113()
Relative error	0.0925×10^{-4}	0.0203×10^{-4}	0.0179×10^{-4}	0.4770×10^{-4}	0.0422×10^{-4}	0.1249×10^{-4}
Computing time	0.05 sec	0.03 sec	0.03 sec	0.07 sec	0.05 sec	0.05 sec

Readers are invited to supplement the program “nm643_2.m” in such a way that “ode_Ham()” is also used to solve Eq. (6.4.11). Running the program yields the results depicted in Fig. 6.4 and listed in Table 6.2. From Fig. 6.4, it is noteworthy that, without the modifiers, the ABM method seems to be better than the Hamming method; however, with the modifiers, it is the other way around or at least they run a neck-and-neck race. Anyone will see that the predictor–corrector methods such as the ABM method (ode_ABM()) and the Hamming method (ode_Ham()) give us a better numerical solution with less error and shorter computation time than the MATLAB built-in routines “ode23()”, “ode45()”, and “ode113()” as well as the RK4 method (ode_RK4()), as listed in Table 6.2. But, a general conclusion should not be deduced just from one example.

6.5 VECTOR DIFFERENTIAL EQUATIONS

6.5.1 State Equation

Although we have tried using the MATLAB routines only for scalar differential equations, all the routines made by us or built inside MATLAB are ready to entertain first-order vector differential equations, called state equations, as below.

$$\begin{aligned}
 x_1'(t) &= f_1(t, x_1(t), x_2(t), \dots) && \text{with } x_1(t_0) = x_{10} \\
 x_2'(t) &= f_2(t, x_1(t), x_2(t), \dots) && \text{with } x_2(t_0) = x_{20} \\
 &\dots\dots\dots \\
 \mathbf{x}'(t) &= \mathbf{f}(t, \mathbf{x}(t)) && \text{with } \mathbf{x}(t_0) = \mathbf{x}_0
 \end{aligned}
 \tag{6.5.1}$$

For example, we can define the system of first-order differential equations

$$\begin{aligned}
 x_1'(t) &= x_2(t) && \text{with } x_1(0) = 1 \\
 x_2'(t) &= -x_2(t) + 1 && \text{with } x_2(0) = -1
 \end{aligned}
 \tag{6.5.2}$$

in a file named “df651.m” and solve it by running the MATLAB program “nm651_1.m”, which uses the routines “ode_Ham()”/“ode45()” to get the numerical solutions and plots the results as depicted in Fig. 6.5. Note that the function given as the first input argument of “ode45()” must be fabricated to generate its value in a column vector or at least, in the same form of vector as the input argument ‘x’ so long as it is a vector-valued function.

```
%nm651_1 to solve a system of differential eqs., i.e., state equation
df = 'df651';
t0 = 0; tf = 2; x0 = [1 -1]; %start/final time and initial value
N = 45; [tH,xH] = ode_Ham(df,[t0 tf],x0,N); %with N = number of segments
[t45,x45] = ode45(df,[t0 tf],x0);
plot(tH,xH), hold on, pause, plot(t45,x45)
```

```
function dx = df651(t,x)
dx = zeros(size(x)); %row/column vector depending on the shape of x
dx(1) = x(2); dx(2) = -x(2) + 1;
```

Especially for the state equations having only constant coefficients like Eq. (6.5.2), we can change it into a matrix–vector form as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \quad (6.5.3)$$

$$\text{with } \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ and } u_s(t) = 1 \quad \forall t \geq 0$$

$$\mathbf{x}'(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t) \text{ with the initial state } \mathbf{x}(0) \text{ and the input } u(t) \quad (6.5.4)$$

which is called a linear time-invariant (LTI) state equation, and then try to find the analytical solution. For this purpose, we take the Laplace transform of both sides to write

$$sX(s) - \mathbf{x}(0) = \mathbf{A}X(s) + \mathbf{B}U(s) \quad \text{with } X(s) = L\{x(t)\}, U(s) = L\{u(t)\}$$

$$[sI - \mathbf{A}]X(s) = \mathbf{x}(0) + \mathbf{B}U(s), \quad X(s) = [sI - \mathbf{A}]^{-1}\mathbf{x}(0) + [sI - \mathbf{A}]^{-1}\mathbf{B}U(s) \quad (6.5.5)$$

where $L\{x(t)\}$ and $L^{-1}\{X(s)\}$ denote the Laplace transform of $x(t)$ and the inverse Laplace transform of $X(s)$, respectively. Note that

$$[sI - \mathbf{A}]^{-1} = s^{-1}[I - \mathbf{A}s^{-1}]^{-1} = s^{-1} [I + \mathbf{A}s^{-1} + \mathbf{A}^2s^{-2} + \dots]$$

$$\phi(t) = L^{-1}\{[sI - \mathbf{A}]^{-1}\} \quad (6.5.6)$$

$$= I + \mathbf{A}t + \frac{\mathbf{A}^2}{2}t^2 + \frac{\mathbf{A}^3}{3!}t^3 + \dots = e^{\mathbf{A}t} \quad \text{with } \phi(0) = I$$

By applying the convolution property of Laplace transform (Table D.2(4) in Appendix D)

$$L^{-1}\{[sI - \mathbf{A}]^{-1}\mathbf{B}U(s)\} = L^{-1}\{[sI - \mathbf{A}]^{-1}\} * L^{-1}\{\mathbf{B}U(s)\} = \phi(t) * \mathbf{B}u(t)$$

$$= \int_{-\infty}^{\infty} \phi(t - \tau)\mathbf{B}u(\tau) d\tau \stackrel{u(\tau)=0 \text{ for } \tau < 0 \text{ or } \tau > t}{=} \int_0^t \phi(t - \tau)\mathbf{B}u(\tau) d\tau \quad (6.5.7)$$

we can take the inverse Laplace transform of Eq. (6.5.5) to write

$$\mathbf{x}(t) = \phi(t)\mathbf{x}(0) + \phi(t) * Bu(t) = \phi(t)\mathbf{x}(0) + \int_0^t \phi(t - \tau)Bu(\tau) d\tau \quad (6.5.8)$$

For Eq. (6.5.3), we use Eq. (6.5.6) to find

$$\begin{aligned} \phi(t) &= L^{-1}\{[sI - A]^{-1}\} \\ &= L^{-1}\left\{\left[\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}\right]^{-1}\right\} = L^{-1}\left\{\begin{bmatrix} s & -1 \\ 0 & s+1 \end{bmatrix}^{-1}\right\} \\ &= L^{-1}\left\{\frac{1}{s(s+1)}\begin{bmatrix} s+1 & 1 \\ 0 & s \end{bmatrix}\right\} \\ &= L^{-1}\left\{\begin{bmatrix} 1/s & 1/s - 1/(s+1) \\ 0 & 1/(s+1) \end{bmatrix}\right\} = \begin{bmatrix} 1 & 1 - e^{-t} \\ 0 & e^{-t} \end{bmatrix} \end{aligned} \quad (6.5.9)$$

and use Eqs. (6.5.8), (6.5.9), and $u(t) = u_s(t) = 1 \forall t \geq 0$ to obtain

$$\begin{aligned} \mathbf{x}(t) &= \begin{bmatrix} 1 & 1 - e^{-t} \\ 0 & e^{-t} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \int_0^t \begin{bmatrix} 1 & 1 - e^{-(t-\tau)} \\ 0 & e^{-(t-\tau)} \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} 1 d\tau \\ &= \begin{bmatrix} e^{-t} \\ -e^{-t} \end{bmatrix} + \left. \begin{bmatrix} \tau - e^{-(t-\tau)} \\ e^{-(t-\tau)} \end{bmatrix} \right|_0^t = \begin{bmatrix} t - 1 + 2e^{-t} \\ 1 - 2e^{-t} \end{bmatrix} \end{aligned} \quad (6.5.10)$$

Alternatively, we can directly take the inverse transform of Eq. (6.5.5) to get

$$\begin{aligned} X(s) &= [sI - A]^{-1}\{\mathbf{x}(0) + [sI - A]^{-1}BU(s)\} \\ &= \frac{1}{s(s+1)} \begin{bmatrix} s+1 & 1 \\ 0 & s \end{bmatrix} \left\{ \begin{bmatrix} 1 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \frac{1}{s} \right\} \\ &= \frac{1}{s^2(s+1)} \begin{bmatrix} s+1 & 1 \\ 0 & s \end{bmatrix} \begin{bmatrix} s \\ -s+1 \end{bmatrix} = \frac{1}{s^2(s+1)} \begin{bmatrix} s^2+1 \\ s(1-s) \end{bmatrix} \end{aligned} \quad (6.5.11)$$

$$X_1(s) = \frac{s^2+1}{s^2(s+1)} = \frac{1}{s^2} - \frac{1}{s} + \frac{2}{s+1}, \quad x_1(t) = t - 1 + 2e^{-t} \quad (6.5.12a)$$

$$X_2(s) = \frac{1-s}{s(s+1)} = \frac{1}{s} - \frac{2}{s+1}, \quad x_2(t) = 1 - 2e^{-t} \quad (6.5.12b)$$

which conforms with Eq. (6.5.10).

The MATLAB program “nm651_2.m” uses a symbolic computation routine “ilaplace()” to get the inverse Laplace transform, uses “eval()” to evaluate

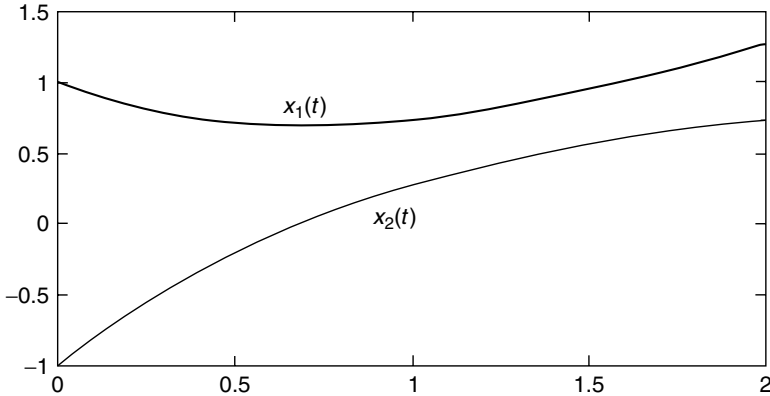


Figure 6.5 Numerical/analytical solutions of the continuous-time state equation (6.5.2)/(6.5.3).

it, and plots the result as depicted in Fig. 6.5, which supports this derivation procedure. Additionally, it uses another symbolic computation routine “`dsolve()`” to get the analytical solution directly.

```
>>nm651_2
Solution of Differential Equation based on Laplace transform
Xs = [ 1/s + 1/s/(s + 1)*(-1 + 1/s) ]
      [          1/(s + 1)*(-1 + 1/s) ]
xt = [ -1 + t + 2*exp(-t) ]
      [ -2*exp(-t) + 1 ]

Analytical solution
xt1 = -1 + t + 2*exp(-t)
xt2 = -2*exp(-t) + 1
```

```
%nm651_2: Analytical solution for state eq.  $x'(t) = Ax(t) + Bu(t)$ (6.5.3)
clear
syms s t %declare s,t as symbolic variables
A = [0 1;0 -1]; B = [0 1]'; %Eq.(6.5.3)
x0 = [1 -1]'; %initial value
disp('Solution of Differential Eq based on Laplace transform')
disp('Laplace transformed solution X(s)')
Xs = (s*eye(size(A)) - A)^-1*(x0 + B/s) %Eq.(6.5.5)
disp('Inverse Laplace transformed solution x(t)')
xt = ilaplace(Xs) %inverse Laplace transform %Eq.(6.5.12)
t0 = 0; tf = 2; N = 45; %initial/final time
t = t0 + [0:N]*(tf - t0)/N; %time vector
xtt = eval(xt:); %evaluate the inverse Laplace transform
plot(t,xtt)
disp('Analytical solution')
xt = dsolve('Dx1 = x2, Dx2 = -x2 + 1', 'x1(0) = 1, x2(0) = -1');
xt1 = xt.x1, xt2 = xt.x2 %Eq.(6.5.10)
```

6.5.2 Discretization of LTI State Equation

In this section, we consider a discretization method of converting a continuous-time LTI (linear time-invariant) state equation

$$\mathbf{x}'(t) = A\mathbf{x}(t) + Bu(t) \quad \text{with the initial state } \mathbf{x}(0) \text{ and the input } u(t) \quad (6.5.13)$$

into an equivalent discrete-time LTI state equation with the sampling period T

$$\mathbf{x}[n + 1] = A_d\mathbf{x}[n] + B_d u[n] \quad (6.5.14)$$

with the initial state $\mathbf{x}[0]$ and the input $u[n] = u(nT)$ for $nT \leq t < (n + 1)T$

which can be solved easily by an iterative scheme mobilizing just simple multiplications and additions.

For this purpose, we rewrite the solution (6.5.8) of the continuous-time LTI state equation with the initial time t_0 as

$$\mathbf{x}(t) = \phi(t - t_0)\mathbf{x}(t_0) + \int_{t_0}^t \phi(t - \tau)Bu(\tau) d\tau \quad (6.5.15)$$

Under the assumption that the input is constant as the initial value within each sampling interval—that is, $u[n] = u(nT)$ for $nT \leq t < (n + 1)T$ —we substitute $t_0 = nT$ and $t = (n + 1)T$ into this equation to write the discrete-time LTI state equation as

$$\begin{aligned} \mathbf{x}((n + 1)T) &= \phi(T)\mathbf{x}(nT) + \int_{nT}^{(n+1)T} \phi((n + 1)T - \tau)Bu(nT) d\tau \\ \mathbf{x}[n + 1] &= \phi(T)\mathbf{x}[n] + \int_{nT}^{(n+1)T} \phi(nT + T - \tau) d\tau Bu[n] \\ \mathbf{x}[n + 1] &= A_d\mathbf{x}[n] + B_d u[n] \end{aligned} \quad (6.5.16)$$

where the discretized system matrices are

$$A_d = \phi(T) = e^{AT} \quad (6.5.17a)$$

$$B_d = \int_{nT}^{(n+1)T} \phi(nT + T - \tau) d\tau B^{\sigma=nT+T-\tau} - \int_T^0 \phi(\sigma) d\sigma B = \int_0^T \phi(\tau) d\tau B \quad (6.5.17b)$$

Here, let us consider another way of computing these system matrices, which is to the taste of digital computers. It comes from making use of the definition of a matrix exponential function in Eq. (6.5.6) to rewrite Eq. (6.5.17) as

$$A_d = e^{AT} = \sum_{m=0}^{\infty} \frac{A^m T^m}{m!} = I + AT \sum_{m=0}^{\infty} \frac{A^m T^m}{(m + 1)!} = I + AT\Psi \quad (6.5.18a)$$

$$B_d = \int_0^T \phi(\tau) d\tau B = \int_0^T \sum_{m=0}^{\infty} \frac{A^m \tau^m}{m!} d\tau B = \sum_{m=0}^{\infty} \frac{A^m T^{m+1}}{(m + 1)!} B = \Psi TB \quad (6.5.18b)$$

where

$$\Psi = \sum_{m=0}^{\infty} \frac{A^m T^m}{(m+1)!}$$

$$\cong I + \frac{AT}{2} \left\{ I + \frac{AT}{3} \left\{ I + \dots + \frac{AT}{N-1} \left(I + \frac{AT}{N} \right) \right\} \dots \right\} \quad \text{for } N > 1 \tag{6.5.19}$$

Now, we apply these discretization formulas for the continuous-time state equation (6.5.3)

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t)$$

with $\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $u_s(t) = 1 \forall t \geq 0$

to get the discretized system matrices and the discretized state equation as

$$\phi(t) = L^{-1}\{[sI - A]^{-1}\} = L^{-1} \left\{ \begin{bmatrix} s & -1 \\ 0 & s+1 \end{bmatrix}^{-1} \right\} \stackrel{(6.5.9)}{=} \begin{bmatrix} 1 & 1 - e^{-t} \\ 0 & e^{-t} \end{bmatrix} \tag{6.5.20a}$$

$$A_d \stackrel{(6.5.17a)}{=} \phi(T) \stackrel{(6.5.20a)}{=} \begin{bmatrix} 1 & 1 - e^{-T} \\ 0 & e^{-T} \end{bmatrix} \tag{6.5.20b}$$

$$B_d \stackrel{(6.5.17b)}{=} \int_0^T \phi(\tau) d\tau B$$

$$\stackrel{(6.5.20a)}{=} \int_0^T \begin{bmatrix} 1 & 1 - e^{-\tau} \\ 0 & e^{-\tau} \end{bmatrix} d\tau \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} T - 1 + e^{-T} \\ 1 - e^{-T} \end{bmatrix} \tag{6.5.20c}$$

$$\mathbf{x}[n+1] \stackrel{(6.5.16)}{=} A_d \mathbf{x}[n] + B_d u[n]$$

$$\begin{bmatrix} x_1[n+1] \\ x_2[n+1] \end{bmatrix} = \begin{bmatrix} 1 & 1 - e^{-T} \\ 0 & e^{-T} \end{bmatrix} \begin{bmatrix} x_1[n] \\ x_2[n] \end{bmatrix} + \begin{bmatrix} T - 1 + e^{-T} \\ 1 - e^{-T} \end{bmatrix} u[n] \tag{6.5.21}$$

We don't need any special algorithm other than an iterative scheme to solve this discrete-time state equation. The formulas (6.5.18a,b) for computing the discretized system matrices are cast into the routine "c2d_steq()". The program "nm652.m" discretizes the continuous-time state equation (6.5.3) by using the routine and alternatively, the MATLAB built-in routine "c2d()". It solves the discretized state equation and plots the results as in Fig. 6.6. As long as the assumption that $u[n] = u(nT)$ for $nT \leq t < (n+1)T$ is valid, the solution ($x[n]$) of the discretized state equation is expected to match that ($x(t)$) of

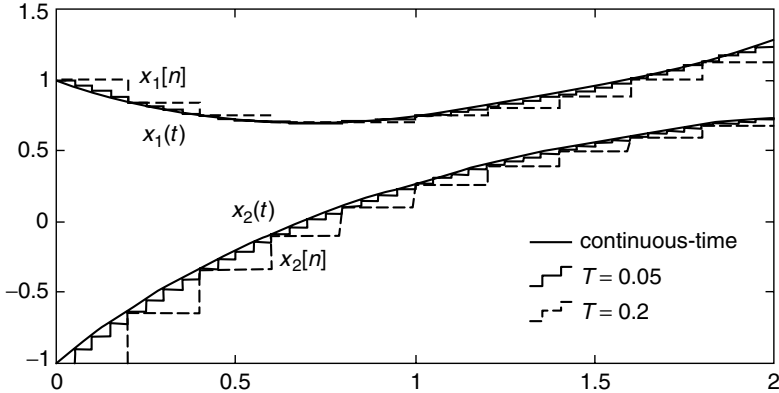


Figure 6.6 The solution of the discretized state equation (6.5.21).

the continuous-time state equation at every sampling instant $t = nT$ and also becomes closer to $x(t) \forall t$ as the sampling interval T gets shorter (see Fig. 6.6).

```
%nm652.m
% discretize a state eqn  $x'(t) = Ax(t) + Bu(t)$  to  $x[n+1] = Ad*x[n] + Bd*u[n]$ 
clear, clf
A = [0 1; 0 -1]; B = [0;1]; %Eq.(6.5.3)
x0 = [1 -1]; t0 = 0; tf = 2; %initial value and time span
T = 0.2; %sampling interval(period)
eT = exp(-T);
AD = [1 1 - eT; 0 eT]; %discretized system matrices obtained analytically
BD = [T + eT - 1; 1 - eT]; %Eq.(6.5.21)
[Ad,Bd] = c2d_steq(A,B,T,100) %continuous-to-discrete conversion
[Ad1,Bd1] = c2d(A,B,T) %by the built-in routine
t(1) = 0; xd(1,:) = x0; %initial time and initial value
for k = 1:(tf - t0)/T %solve the discretized state equation
    t(k+1) = k*T; xd(k+1,:) = xd(k,)*Ad' + Bd';
end
stairs([0; t'],[x0; xd]), hold on %stairstep graph
N = 100; t = t0 + [0:N]'*(tf - t0)/N; %time (column) vector
x(:,1) = t-1 + 2*exp(-t); %analytical solution
x(:,2) = 1-2*exp(-t); %Eq.(6.5.12)
plot(t,x)
```

```
function [Ad,Bd] = c2d_steq(A,B,T,N)
if nargin < 4, N = 100; end
I = eye(size(A,2)); PSI = I;
for m = N:-1:1, PSI = I + A*PSI*T/(m + 1); end %Eq.(6.5.19)
Ad = I + A*PSI*T; Bd = PSI*T*B; %Eq.(6.5.18)
```

6.5.3 High-Order Differential Equation to State Equation

Suppose we are given an N th-order scalar differential equation together with the initial values of the variable and its derivatives of up to order $N - 1$, which is

called an IVP (Initial Value Problem):

$$[\text{IVP}]_N : x^{(N)}(t) = f(t, x(t), x'(t), x^{(2)}(t), \dots, x^{(N-1)}(t)) \quad (6.5.22)$$

with the initial values $x(t_0) = x_{10}, x'(t_0) = x_{20}, \dots, x^{(N-1)}(t_0) = x_{N0}$

Defining the state vector and the initial state as

$$\mathbf{x}(t) = \begin{bmatrix} x_1 = x \\ x_2 = x' \\ x_3 = x^{(2)} \\ \vdots \\ x_N = x^{(N-1)} \end{bmatrix}, \quad \mathbf{x}(t_0) = \begin{bmatrix} x_{10} \\ x_{20} \\ x_{30} \\ \vdots \\ x_{N0} \end{bmatrix} \quad (6.5.23)$$

we can rewrite Eq. (6.5.22) in the form of a first-order vector differential equation—that is, a state equation—as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \\ x_3'(t) \\ \vdots \\ x_N'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ x_3(t) \\ x_4(t) \\ \vdots \\ f(t, x(t), x'(t), x^{(2)}(t), \dots, x^{(N-1)}(t)) \end{bmatrix}$$

$$\mathbf{x}'(t) = \mathbf{f}(t, \mathbf{x}(t)) \quad \text{with } \mathbf{x}(t_0) = \mathbf{x}_0 \quad (6.5.24)$$

For example, we can convert a third-order scalar differential equation

$$x^{(3)}(t) + a_2x^{(2)}(t) + a_1x'(t) + a_0x(t) = u(t)$$

into a state equation of the form

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \\ x_3'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u(t) \quad (6.5.25a)$$

$$\mathbf{x}(t) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{bmatrix} \quad (6.5.25b)$$

6.5.4 Stiff Equation

Suppose that we are given a vector differential equation involving more than one dependent variable with respect to the independent variable t . If the magnitudes of the derivatives of the dependent variables with respect to t (corresponding

to their changing rates) are significantly different, such a differential equation is said to be stiff because it is difficult to be solved numerically. For such a stiff differential equation, we should be very careful in choosing the step-size in order to avoid numerical instability problem and get a reasonably accurate solution within a reasonable computation time. Why? Because we should use a small step-size to grasp rapidly changing variables, and it requires a lot of computation to cover slowly changing variables for such a long time as it lasts.

Actually, there is no clear distinction between stiff and non-stiff differential equations, since stiffness of a differential equation is a matter of degree. Then, is there any way to estimate the degree of stiffness for a given differential equation? The answer is yes, if the differential equation can be arranged into an LTI state equation like Eq. (6.5.4), the solution of which consists of components having the time constants (modes) equal to the eigenvalues of the system matrix A . For example, the system matrix of Eq. (6.5.3) has the eigenvalues

$$|sI - A| = 0, \quad \det \left\{ \begin{bmatrix} s & -1 \\ 0 & s + 1 \end{bmatrix} \right\} = s(s + 1) = 0, \quad s = 0 \text{ and } s = -1$$

which can be observed as the time constants of two terms $1 = e^{0t}$ and e^{-t} in the solution (6.5.12). In this context, a measure of stiffness is the ratio of the maximum over the minimum among the absolute values of (negative) real parts of the eigenvalues of the system matrix A :

$$\eta(A) = \frac{\text{Max}\{|\text{Re}(\lambda_i)|\}}{\text{Min}\{|\text{Re}(\lambda_i)| \neq 0\}} \tag{6.5.26}$$

This can be thought of as the degree of unbalance between the fast mode and the slow mode.

Now, what we must know is how to handle stiff differential equations. Fortunately, MATLAB has several built-in routines like “ode15s()”, “ode23s()”, “ode23t()”, and “ode23tb()”, which are fabricated to deal with stiff differential equations efficiently. One may use the `help` command to see their detailed usages. Let’s apply them for a Van der Pol equation

$$\frac{d^2y(t)}{dt^2} - \mu(1 - y^2(t))\frac{dy(t)}{dt} + y(t) = 0 \quad \text{with} \quad y(0) = 2, \frac{dy(t)}{dt} = 0 \tag{6.5.27a}$$

which can be written in the form of a state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ \mu(1 - x_1^2(t))x_2(t) - x_1(t) \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \tag{6.5.27b}$$

For this job, we defined this equation in an M-file named “df_van.m” and made the MATLAB program “nm654.m”, where we declared the parameter μ (mu) as

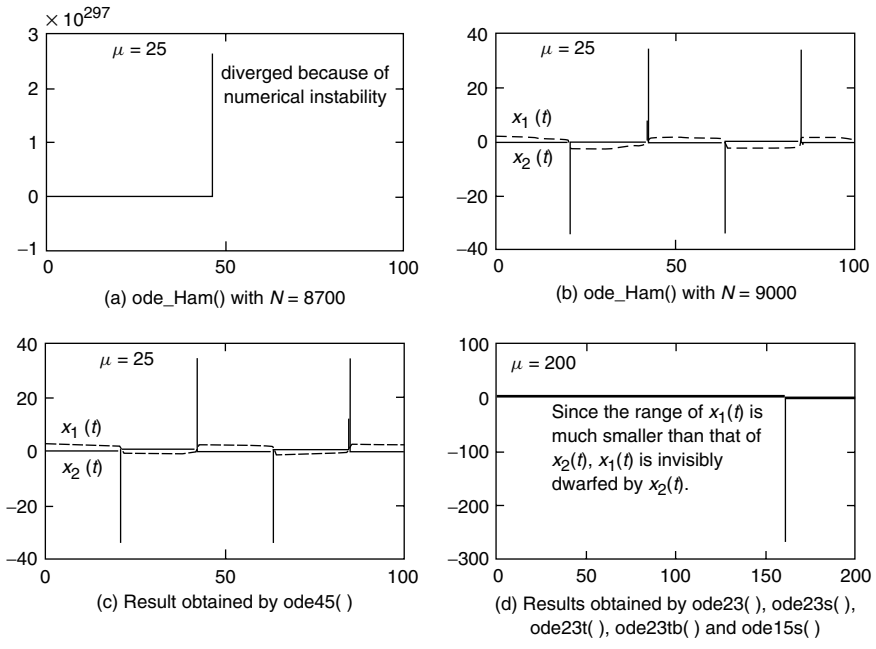


Figure 6.7 Numerical solutions of Van der Pol equation obtained by various routines.

a global variable so that it could be passed on to any related routines/functions as well as “df_van.m”. In the beginning of the program, we set the global parameter μ to 25 and applied “ode_Ham()” with the number of segments $N = 8700$ and 9000 . The results are depicted in Figs. 6.7a and 6.7b, which show how crucial the choice of step-size is for a stiff equation. Next, we applied “ode45()” to obtain the solution depicted in Fig. 6.7c, which is almost the same as Fig. 6.7b, but with the computation time less than one fourth of that taken by “ode_Ham()”. This reveals the merit of the MATLAB built-in routines that may save the computation time as well as spare our trouble to choose the step-size, because the step-size is adaptively determined inside the routines. Then, setting $\mu = 200$, we applied the MATLAB built-in routines “ode45()”/“ode23()”/“ode15s()”/“ode23s()”/“ode23t()”/“ode23tb()” to get the results that are little different as depicted in Fig. 6.7d, each taking the computation time as

time = 24.9530 14.9690 0.1880 0.2650 0.2500 0.2820

The computation time-efficiency of “ode15s()”/“ode23s()”/“ode23t()”/“ode23tb()” (designed deliberately for handling stiff differential equations) over “ode45()”/“ode23()” becomes prominent as the value of parameter μ (mu) gets large, reflecting high stiffness.

```

%nm654.m
% to solve a stiff differential eqn called Van der Pol equation
global mu
mu=25, t0=0; tf = 100; tspan = [t0 tf]; xo = [2 0];
[tH1,xH1] = ode_Ham('df_van',tspan,x0,8700);
subplot(221), plot(tH1,xH1)
tic,[tH2,xH2] = ode_Ham('df_van',tspan,x0,9000); time_Ham = toc
tic,[t45,x45] = ode45('df_van',tspan,x0); time_o45 = toc
subplot(222), plot(tH2,xH2), subplot(223), plot(t45,x45)
mu = 200; tf = 200; tspan = [t0 tf];
tic,[t45,x45] = ode45('df_van',tspan,x0); time(1) = toc;
tic,[t23,x23] = ode23('df_van',tspan,x0); time(2) = toc;
tic,[t15s,x15s] = ode15s('df_van',tspan,x0); time(3) = toc;
tic,[t23s,x23s] = ode23s('df_van',tspan,x0); time(4) = toc;
tic,[t23t,x23t] = ode23t('df_van',tspan,x0); time(5) = toc;
tic,[t23tb,x23tb] = ode23tb('df_van',tspan,x0); time(6) = toc;
plot(t45,x45, t23,x23, t15s,x15s, t23s,x23s, t23t,x23t, t23tb,x23tb)
disp(' ode23 ode15s ode23s ode23t ode23tb')
time

function dx = df_van(t,x)
%Van der Pol differential equation (6.5.27)
global mu
dx=zeros(size(x));
dx(1) = x(2); dx(2) = mu*(1-x(1).^2).*x(2) - x(1);

```

6.6 BOUNDARY VALUE PROBLEM (BVP)

A boundary value problem (BVP) is an N th-order differential equation with some of the values of dependent variable $x(t)$ and its derivative specified at the initial time t_0 and others specified at the final time t_f .

$$[\text{BVP}]_N : \quad x^{(N)}(t) = f(t, x(t), x'(t), x^{(2)}(t), \dots, x^{(N-1)}(t))$$

$$\text{with the boundary values } x(t_1) = x_{10}, x'(t_2) = x_{21}, \dots, x^{(N-1)}(t_N) = x_{N,N-1} \quad (6.6.1)$$

In some cases, some relations between the initial values and the final values may be given as a mixed-boundary condition instead of the initial/final values specified. This section covers the shooting method and the finite difference method that can be used to solve a second-order BVP as

$$[\text{BVP}]_2 : x''(t) = f(t, x(t), x'(t)) \quad \text{with } x(t_0) = x_0, x(t_f) = x_f \quad (6.6.2)$$

6.6.1 Shooting Method

The idea of this method is to assume the value of $x'(t_0)$, then solve the differential equation (IVP) with the initial condition $[x(t_0) \ x'(t_0)]$ and keep adjusting the value

of $x'(t_0)$ and solving the IVP repetitively until the final value $x(t_f)$ of the solution matches the given boundary value x_f with enough accuracy. It is similar to adjusting the angle of firing a cannon so that the shell will eventually hit the target and that's why this method is named the *shooting* method. This can be viewed as a nonlinear equation problem, if we regard $x'(t_0)$ as an independent variable and the difference between the resulting final value $x(t_f)$ and the desired one x_f as a (mismatching) function of $x'(t_0)$. So the solution scheme can be systemized by using the secant method (Section 4.5) and is cast into the MATLAB routine "bvp2_shoot()".

(cf) We might have to adjust the shooting position with the angle fixed, instead of adjusting the shooting angle with the position fixed or deal with the mixed-boundary conditions. See Problems 6.6, 6.7, and 6.8.

For example, let's consider a BVP consisting of the second-order differential equation

$$x''(t) = 2x^2(t) + 4t x(t)x'(t) \quad \text{with } x(0) = \frac{1}{4}, x(1) = \frac{1}{3} \quad (6.6.3)$$

```
function [t,x] = bvp2_shoot(f,t0,tf,x0,xf,N,tol,kmax)
%To solve BVP2: [x1,x2]' = f(t,x1,x2) with x1(t0) = x0, x1(tf) = xf
if nargin < 8, kmax = 10; end
if nargin < 7, tol = 1e-8; end
if nargin < 6, N = 100; end
dx0(1) = (xf - x0)/(tf-t0); % the initial guess of x'(t0)
[t,x] = ode_RK4(f,[t0 tf],[x0 dx0(1)],N); % start up with RK4
plot(t,x(:,1)), hold on
e(1) = x(end,1) - xf; % x(tf) - xf: the 1st mismatching (deviation)
dx0(2) = dx0(1) - 0.1*sign(e(1));
for k = 2: kmax-1
    [t,x] = ode_RK4(f,[t0 tf],[x0 dx0(k)],N);
    plot(t,x(:,1))
    %difference between the resulting final value and the target one
    e(k) = x(end,1) - xf; % x(tf) - xf
    ddx = dx0(k) - dx0(k - 1); % difference between successive derivatives
    if abs(e(k)) < tol | abs(ddx) < tol, break; end
    deddx = (e(k) - e(k - 1))/ddx; % the gradient of mismatching error
    dx0(k + 1) = dx0(k) - e(k)/deddx; %move by secant method
end

%do_shoot to solve BVP2 by the shooting method
t0 = 0; tf = 1; x0 = 1/4; xf = 1/3; %initial/final times and positions
N = 100; tol = 1e-8; kmax = 10;
[t,x] = bvp2_shoot('df661',t0,tf,x0,xf,N,tol,kmax);
xo = 1./(4 - t.*t); err = norm(x(:,1) - xo)/(N + 1)
plot(t,x(:,1),'b', t,xo,'r') %compare with true solution (6.6.4)

function dx = df661(t,x) %Eq.(6.6.5)
dx(1) = x(2); dx(2) = (2*x(1) + 4*t*x(2))*x(1);
```

The solution $x(t)$ and its derivative $x'(t)$ are known as

$$x(t) = \frac{1}{4 - t^2} \quad \text{and} \quad x'(t) = \frac{2t}{(4 - t^2)^2} = 2t x^2(t) \quad (6.6.4)$$

Note that this second-order differential equation can be written in the form of state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} x_2(t) \\ 2x_1^2(t) + 4t x_1(t)x_2(t) \end{bmatrix} \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(1) \end{bmatrix} = \begin{bmatrix} x_0 = 1/4 \\ x_f = 1/3 \end{bmatrix} \quad (6.6.5)$$

In order to apply the shooting method, we set the initial guess of $x_2(0) = x'(0)$ to

$$dx0[1] = x_2(0) = \frac{x_f - x_0}{t_f - t_0} \quad (6.6.6)$$

and solve the state equation with the initial condition $[x_1(0) \ x_2(0) = dx0[1]]$. Then, depending on the sign of the difference $e(1)$ between the final value $x_1(1)$ of the solution and the target final value x_f , we make the next guess $dx0[2]$ larger/smaller than the initial guess $dx0[1]$ and solve the state equation again with the initial condition $[x_1(0) \ dx0[2]]$. We can start up the secant method with the two initial values $dx0[1]$ and $dx0[2]$ and repeat the iteration until the difference (error) $e(k)$ becomes sufficiently small. For this job, we compose the MATLAB program “do_shoot.m”, which uses the routine “bvp2_shoot()” to get the numerical solution and compares it with the true analytical solution. Figure 6.8 shows that the numerical solution gets closer to the true analytical solution after each round of adjustment.

- (Q) Why don't we use the Newton method (Section 4.4)?
- (A) Because, in order to use the Newton method in the shooting method, we need IVP solutions instead of function evaluations to find the numerical Jacobian at every iteration, which will require much longer computation time.

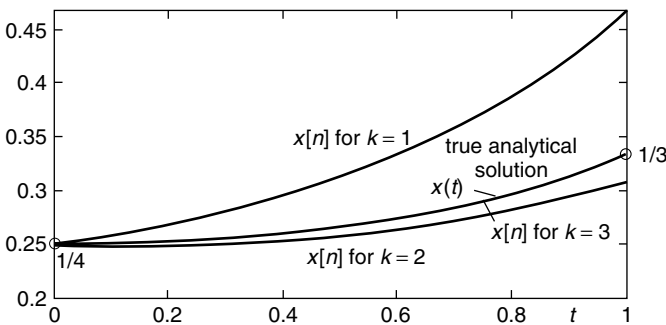


Figure 6.8 The solution of a BVP obtained by using the shooting method.

6.6.2 Finite Difference Method

The idea of this method is to divide the whole interval $[t_0, t_f]$ into N segments of width $h = (t_f - t_0)/N$ and approximate the first & second derivatives in the differential equations for each grid point by the central difference formulas. This leads to a tridiagonal system of equations with respect to $(N - 1)$ variables $\{x_i = x(t_0 + ih), i = 1, \dots, N - 1\}$. However, in order for this system of equations to be solved easily, it should be linear, implying that its coefficients may not contain any term of x .

For example, let's consider a BVP consisting of the second-order linear differential equation

$$x''(t) + a_1(t)x'(t) + a_0(t)x(t) = u(t) \quad \text{with } x(t_0) = x_0, x(t_f) = x_f \quad (6.6.7)$$

According to the finite difference method, we divide the solution interval $[t_0, t_f]$ into N segments and convert the differential equation for each grid point $t_i = t_0 + ih$ into a difference equation as

$$\frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} + a_{1i} \frac{x_{i+1} - x_{i-1}}{2h} + a_{0i}x_i = u_i$$

$$(2 - ha_{1i})x_{i-1} + (-4 + 2h^2a_{0i})x_i + (2 + ha_{1i})x_{i+1} = 2h^2u_i \quad (6.6.8)$$

Then, taking account of the boundary condition that $x_0 = x(t_0)$ and $x_N = x(t_f)$, we collect all of the $(N - 1)$ equations to construct a tridiagonal system of equations as

$$\begin{bmatrix} -4 + 2h^2a_{01} & 2 + ha_{11} & 0 & \bullet & 0 & 0 & 0 \\ 2 - ha_{12} & -4 + 2h^2a_{02} & 2 + ha_{12} & \bullet & 0 & 0 & 0 \\ 0 & 2 - ha_{13} & -4 + 2h^2a_{03} & \bullet & 0 & 0 & 0 \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ 0 & 0 & 0 & \bullet & -4 + 2h^2a_{0,N-3} & 2 + ha_{1,N-3} & 0 \\ 0 & 0 & 0 & \bullet & 2 - ha_{1,N-2} & -4 + 2h^2a_{0,N-2} & 2 + ha_{1,N-2} \\ 0 & 0 & 0 & \bullet & 0 & 2 - ha_{1,N-1} & -4 + 2h^2a_{0,N-1} \end{bmatrix}$$

$$\times \begin{bmatrix} x_1 \\ x_2 \\ x_2 \\ \bullet \\ x_{N-3} \\ x_{N-2} \\ x_{N-1} \end{bmatrix} = \begin{bmatrix} 2h^2u_1 - (2 - ha_{11})x_0 \\ 2h^2u_2 \\ 2h^2u_3 \\ \bullet \\ 2h^2u_{N-3} \\ 2h^2u_{N-2} \\ 2h^2u_{N-1} - (2 - ha_{1,N-1})x_N \end{bmatrix} \quad (6.6.9)$$

This can be solved efficiently by using the MATLAB routine “trid()”, which is dedicated to a tridiagonal system of linear equations.

The whole procedure of the finite difference method for solving a second-order linear differential equation with boundary conditions is cast into the MATLAB routine “bvp2_fdf()”. This routine is designed to accept the two coefficients a_1 and a_0 and the right-hand-side input u of Eq. (6.6.7) as its first three input arguments, where any of those three input arguments can be given as the function name in case the corresponding term is not a numeric value, but a function of time t . We make the program “do_fdf” to use this routine for solving the second-order BVP

$$x''(t) + \frac{2}{t}x'(t) - \frac{2}{t^2}x(t) = 0 \quad \text{with } x(1) = 5, x(2) = 3 \quad (6.6.10)$$

```
function [t,x] = bvp2_fdf(a1,a0,u,t0,tf,x0,xf,N)
% solve BVP2: x'' + a1*x' + a0*x = u with x(t0) = x0, x(tf) = xf
% by the finite difference method
h = (tf - t0)/N; h2 = 2*h*h;
t = t0+[0:N]'*h;
if ~isnumeric(a1), a1 = a1(t(2:N)); %if a1 = name of a function of t
elseif length(a1) == 1, a1 = a1*ones(N - 1,1);
end
if ~isnumeric(a0), a0 = a0(t(2:N)); %if a0 = name of a function of t
elseif length(a0) == 1, a0 = a0*ones(N - 1,1);
end
if ~isnumeric(u), u = u(t(2:N)); %if u = name of a function of t
elseif length(u) == 1, u = u*ones(N-1,1);
else u = u(:);
end
A = zeros(N - 1,N - 1); b = h2*u;
ha = h*a1(1); A(1,1:2) = [-4 + h2*a0(1) 2 + ha];
b(1) = b(1)+(ha - 2)*x0;
for m = 2:N - 2 %Eq.(6.6.9)
    ha = h*a1(m); A(m,m - 1:m + 1) = [2-ha -4 + h2*a0(m) 2 + ha];
end
ha = h*a1(N - 1); A(N - 1,N - 2:N - 1) = [2 - ha -4 + h2*a0(N - 1)];
b(N - 1) = b(N-1)-(ha+2)*xf;
x = [x0 trid(A,b)' xf]';
```

```
function x = trid(A,b)
% solve tridiagonal system of equations
N = size(A,2);
for m = 2:N % Upper Triangularization
    tmp = A(m,m - 1)/A(m - 1,m - 1);
    A(m,m) = A(m,m) -A(m - 1,m)*tmp; A(m,m - 1) = 0;
    b(m,:) = b(m,:) -b(m - 1,:)*tmp;
end
x(N,:) = b(N,+)/A(N,N);
for m = N - 1:-1:1 % Back Substitution
    x(m,:) = (b(m,:) -A(m,m + 1)*x(m + 1))/A(m,m);
end
```

```

%do_fdf to solve BVP2 by the finite difference method
clear, clf
t0 = 1; x0 = 5; tf = 2; xf = 3; N = 100;
a1 = inline('2./t','t'); a0 = inline('-2./t./t','t'); u = 0; %Eq.(6.6.10)
[tt,x] = bvp2_fdf(a1,a0,u,t0,tf,x0,xf,N);
%use the MATLAB built-in command 'bvp4c()'
df = inline('[x(2); 2./t.*(x(1)./t - x(2))]', 't','x');
fbc = inline('[x0(1) - 5; xf(1) - 3]', 'x0','xf');
solinit = bvpinit(linspace(t0,tf,5),[1 10]); %initial solution interval
sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-4));
x_bvp = deval(sol,tt); xbv = x_bvp(1,:);
%use the symbolic computation command 'dsolve()'
xo = dsolve('D2x + 2*(Dx - x/t)/t=0','x(1) = 5, x(2) = 3')
xot = subs(xo,'t',tt); %xot=4./tt./tt +tt; %true analytical solution
err_fd = norm(x - xot)/(N+1) %error between numerical/analytical solution
err_bvp = norm(xbv - xot)/(N + 1)
plot(tt,x,'b',tt,xbv,'r',tt,xot,'k') %compare with analytical solution

```

We run it to get the result depicted in Fig. 6.9 and, additionally, use the symbolic computation command “dsolve()” and “subs()” to get the analytical solution

$$x(t) = t + \frac{4}{t^2} \quad (6.6.11)$$

and substitute the time vector into the analytical solution to obtain its numeric values for check.

Note the following things about the shooting method and the finite difference method:

- While the shooting method is applicable to linear/nonlinear BVPs, the finite difference method is suitable for linear BVPs. However, we can also apply the finite difference method in an iterative manner to solve nonlinear BVPs (see Problem 6.10).

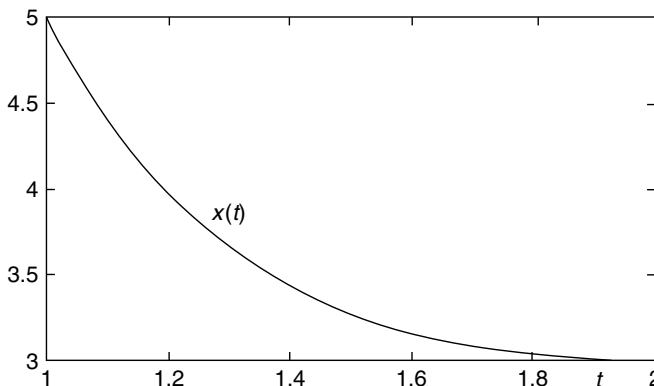


Figure 6.9 A solution of a BVP obtained by using the finite difference method.

- Both methods can be modified to solve BVPs with mixed-boundary conditions (see Problems 6.7 and 6.8).
- In MATLAB 6.x, the “bvp4c()” command is available for solving linear/nonlinear BVPs with mixed-boundary conditions (see Problems 6.7–6.10).
- The symbolic computation command “dsolve()” introduced in Section 6.5.1 can be used to solve a BVP so long as the differential equation is linear, that is, its coefficients may depend on time t , but not on the (unknown) dependent variable $x(t)$.
- The usages of “bvp4c()” and “dsolve()” are illustrated in the program “do_fdf”, where another symbolic computation command “subs()” is used to evaluate a symbolic expression at certain value(s) of the variable.

PROBLEMS

6.0 MATLAB Commands quiver() and quiver3() and Differential Equation

(a) Usage of quiver()

Type ‘help quiver’ into the MATLAB command window, and then you will see the following program showing you how to use the quiver() command for plotting gradient vectors. You can also get Fig. P6.0.1 by running the block of statements in the box below. Try it and note that the size of the gradient vector at each point is proportional to the slope at the point.

```
%do_quiver
[x,y] = meshgrid(-2:.5:2,-1:.25:1);
z = x.*exp(-x.^2 - y.^2);
[px,py] = gradient(z,.5,.25);
contour(x,y,z), hold on, quiver(x,y,px,py)
axis image %the same as AXIS EQUAL except that
           %the plot box fits tightly around the data
```

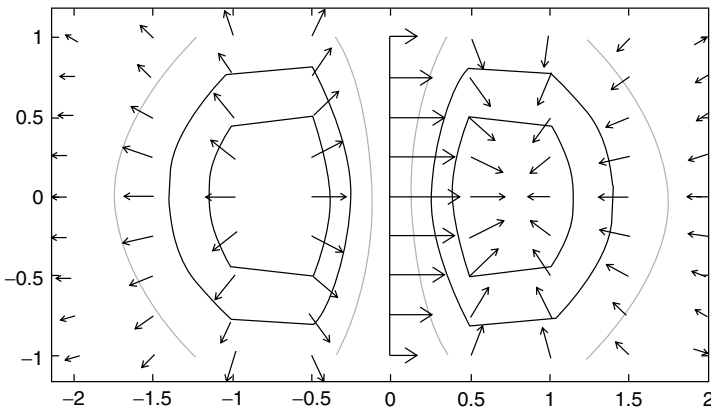


Figure P6.0.1 Graphs obtained by using gradient(), contour(), quiver().

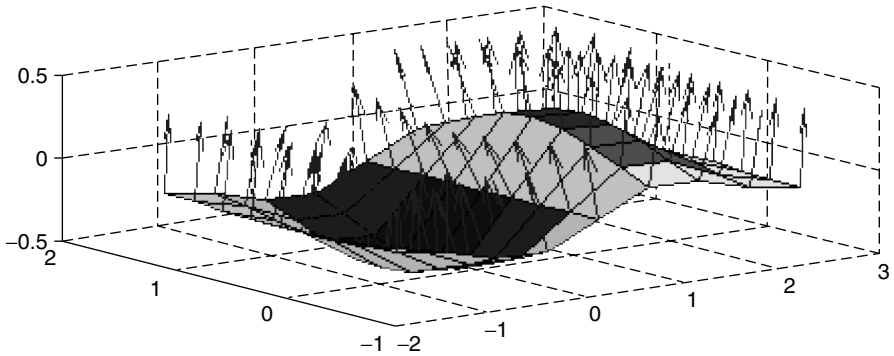


Figure P6.0.2 Graphs obtained by using `surfnorm()`, `quiver3()`, `surf()`.

(b) Usage of `quiver3()`

You can obtain Fig. P6.0.2 by running the block of statements that you see after typing ‘`help quiver3`’ into the MATLAB command window. Note that the “`surfnorm()`” command generates normal vectors at points specified by (x, y, z) on the surface drawn by “`surf()`” and the “`quiver3()`” command plots the normal vectors.

```
%do_quiver3
clear, clf
[x,y] = meshgrid(-2:.5:2,-1:.25:1);
z = x.*exp(-x.^2 - y.^2);
surf(x,y,z), hold on
[u,v,w] = surfnorm(x,y,z);
quiver3(x,y,z,u,v,w);
```

(c) Gradient Vectors and One-Variable Differential Equation

We might get the meaning of the solution of a differential equation by using the “`quiver()`” command, which is used in the following program “`do_ode.m`” for drawing the time derivatives at grid points as defined by the differential equation

$$\frac{dy(t)}{dt} = -y(t) + 1 \quad \text{with the initial condition } y(0) = 0 \quad (\text{P6.0.1})$$

The slope/direction field together with the numerical solution in Fig. P6.0.3a is obtained by running the program and it can be regarded as a set of possible solution curve segments. Starting from the initial point and moving along the slope vectors, you can get the solution curve. Modify the program and run it to plot the slope/direction

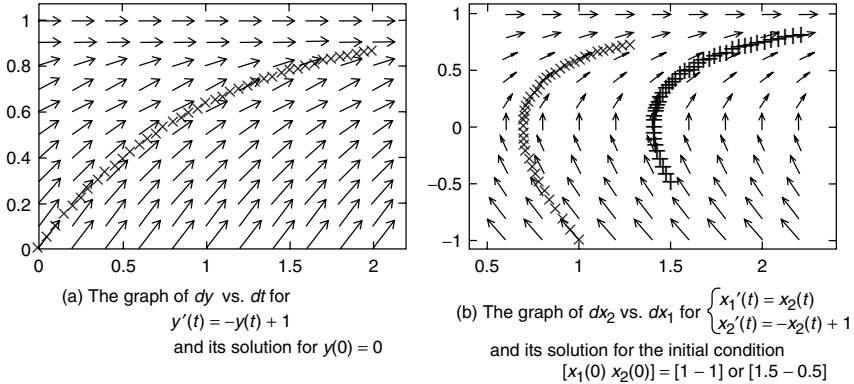


Figure P6.0.3 Possible solutions of differential equation and slope/direction field.

field $(x_2(t)$ versus $x_1(t))$ and the numerical solution for the following differential equation as depicted in Fig. P6.0.3b.

$$\begin{aligned} x_1'(t) &= x_2(t) \\ x_2'(t) &= -x_2(t) + 1 \end{aligned} \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \text{ or } \begin{bmatrix} 1.5 \\ -0.5 \end{bmatrix} \quad (\text{P6.0.2})$$

```

%do_ode.m
% This uses quiver() to plot possible solution curve segments
% called the slope/directional field for y'(t) + y = 1
clear, clf
t0 = 0; tf = 2; tspan = [t0 tf]; x0 = 0;
[t,y] = meshgrid(t0:(tf - t0)/10:tf,0:.1:1);
pt = ones(size(t)); py = (1 - y).*pt; %dy = (1 - y)dt
quiver(t,y,pt,py) %y(displacement) vs. t(time)
axis([t0 tf + .2 0 1.05]), hold on
dy=inline('-y + 1', 't', 'y');
[tR,yR] = ode_RK4(dy,tspan,x0,40);
for k = 1:length(tR), plot(tR(k),yR(k),'rx'), pause(0.001); end
    
```

6.1 A System of Linear Time-Invariant Differential Equations: An LTI State Equation

Consider the following state equation:

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_s(t) \quad \text{with} \quad \begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad (\text{P6.1.1})$$

- (a) Check the procedure and the result of obtaining the analytical solution by using the Laplace transform technique.

$$\begin{aligned}
 X(s) &= [sI - A]^{-1}\{\mathbf{x}(0) + BU(s)\} \\
 &= \frac{1}{s(s+3)+2} \begin{bmatrix} s+3 & 1 \\ -2 & s \end{bmatrix} \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \frac{1}{s} \right\} \\
 &= \frac{1}{(s+1)(s+2)} \begin{bmatrix} s+3+1/s \\ -2+1 \end{bmatrix} \\
 &= \begin{bmatrix} (s^2+3s+1)/s(s+1)(s+2) \\ -1/(s+1)(s+2) \end{bmatrix}
 \end{aligned}$$

$$X_1(s) = \frac{1/2}{s} + \frac{1}{s+1} - \frac{1/2}{s+2}, \quad x_1(t) = \frac{1}{2} + e^{-t} - \frac{1}{2}e^{-2t} \quad (\text{P6.1.2a})$$

$$X_2(s) = \frac{-1}{s+1} + \frac{1}{s+2}, \quad x_2(t) = -e^{-t} + e^{-2t} \quad (\text{P6.1.2b})$$

- (b) Find the numerical solution of the above state equation by using the routine “ode_RK4()” (with the number of segments $N = 50$) and the MATLAB built-in routine “ode45()”. Compare their execution time (by using tic and toc) and closeness to the analytical solution.

6.2 A Second-Order Linear Time-Invariant Differential Equation

Consider the following second-order differential equation

$$x''(t) + 3x'(t) + 2x(t) = 1 \quad \text{with } x(0) = 1, x'(0) = 0 \quad (\text{P6.2.1})$$

- (a) Check the procedure and the result of obtaining the analytical solution by using the Laplace transform technique.

$$\begin{aligned}
 s^2X(s) - x'(0) - sx(0) + 3(sX(s) - x(0)) + 2X(s) &= \frac{1}{s} \\
 X(s) = \frac{s^2 + 3s + 1}{s(s+1)(s+2)}, \quad x(t) &= \frac{1}{2} + e^{-t} - \frac{1}{2}e^{-2t} \quad (\text{P6.2.2})
 \end{aligned}$$

- (b) Define the differential equation (P6.2.1) in an M-file so that it can be passed to the MATLAB routines like “ode_RK4()” or “ode45()” as their input argument (see Section 6.5.1).

6.3 Ordinary Differential Equation and State Equation

- (a) Van der Pol Equation

Consider a nonlinear differential equation

$$\frac{d^2}{dt^2}y(t) - \mu(1 - y^2(t))\frac{d}{dt}y(t) + y(t) = 0 \quad \text{with } \mu = 2 \quad (\text{P6.3.1})$$

Compose a program to solve this equation with the initial condition $[y(0) \ y'(0)] = [0.5 \ 0]$ and $[-1 \ 2]$ for the time interval $[0, 20]$ and plot $y'(t)$ versus $y(t)$ as well as $y(t)$ and $y'(t)$ along the t -axis.

- (b) **Lorenz Equation: Turbulent Flow and Chaos**
 Consider a nonlinear state equation.

$$\begin{aligned} x_1'(t) &= \sigma(x_2(t) - x_1(t)) & \sigma &= 10 \\ x_2'(t) &= (1 + \lambda - x_3(t))x_1(t) - x_2(t) & \text{with } \lambda &= 20 \sim 100 \quad (\text{P6.3.2}) \\ x_3'(t) &= x_1(t)x_2(t) - \gamma x_3(t) & \gamma &= 2 \end{aligned}$$

Compose a program to solve this equation with $\lambda = 20$ and 100 for the time interval $[0,10]$ and plot $x_3(t)$ versus $x_1(t)$. Let the initial condition be $[x_1(0) \ x_2(0) \ x_3(0)] = [-8 \ -16 \ 80]$.

- (c) **Chemical Reactor**

Consider a nonlinear state equation describing the concentrations of two reactants and one product in the chemical process.

$$\begin{aligned} x_1'(t) &= a(u_1 - x_1(t)) - bx_1(t)x_2(t) & a &= 5 \\ x_2'(t) &= a(u_2 - x_2(t)) - bx_1(t)x_2(t) & \text{with } b &= 2 \\ x_3'(t) &= -ax_3(t) + bx_1(t)x_2(t) & u_1 = 3, u_2 = 5 \end{aligned} \quad (\text{P6.3.3})$$

Compose a program to solve this equation for the time interval $[0, 1]$ and plot $x_1(t)$, $x_2(t)$, and $x_3(t)$. Let the initial condition be $[x_1(0) \ x_2(0) \ x_3(0)] = [1 \ 2 \ 3]$.

- (d) **Cantilever Beam: A Differential Equation w.r.t a Spatial Variable**

Consider a nonlinear state equation describing the vertical deflection of a beam due to its own weight

$$JE \frac{d^2y}{dx^2} = \rho g \left(1 + \frac{dy}{dx} \right)^2 \left\{ x \left(x - \frac{L}{2} \right) + \frac{L^2}{2} \right\} \quad (\text{P6.3.4})$$

where $JE = 2000 \text{ kg} \cdot \text{m}^3/\text{s}^2$, $\rho = 10 \text{ kg/m}$, $g = 9.8 \text{ m/s}^2$, $L = 2 \text{ m}$. Write a program to solve this equation for the interval $[0, L]$ and plot $y(t)$. Let the initial condition be $[y(0) \ y'(0)] = [0 \ 0]$. Note that the physical meaning of the independent variable for which we usually use the symbol 't' in writing the differential function is not a time, but the x -coordinate of the cantilever beam along the horizontal axis in this problem.

- (e) **Phase-Locked Loop (PLL)**

Consider a nonlinear state equation describing the behavior of a PLL circuit depicted in Fig. P6.3.1.

$$x_1'(t) = \frac{au(t) \cos(x_2(t)) - x_1(t)}{\tau} \quad \text{with} \quad \begin{aligned} a &= 1500 \\ \tau &= 0.002 \end{aligned} \quad (\text{P6.3.5a})$$

$$x_2'(t) = x_1(t) + \omega_c$$

$$y(t) = x_1(t) + \omega_c \quad (\text{P6.3.5b})$$

$$u(t) = \sin(\omega_0 t)$$

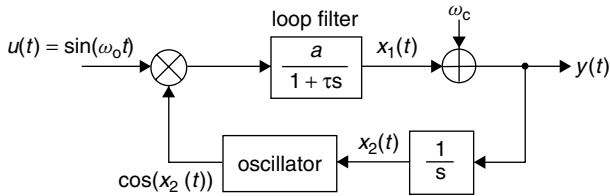


Figure P6.3.1 The block diagram of PLL circuit.

where $\omega_0 = 2100\pi$ [rad/s] and $\omega_c = 2000\pi$ [rad/s]. Compose a program to solve this equation for the time interval $[0,0.03]$ and plot $y(t)$ and ω_0 . Let the initial condition be $[x_1(0) \ x_2(0)] = [0 \ 0]$. Is the output $y(t)$ tracking the frequency ω_0 of the input $u(t)$?

(f) DC Motor

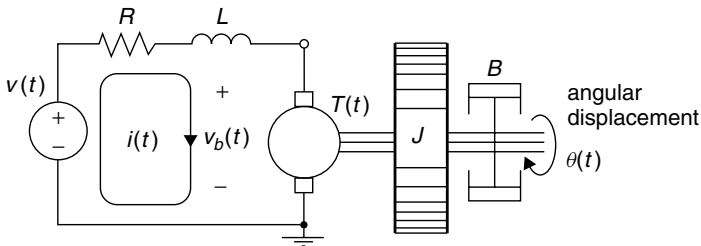
Consider a linear differential equation describing the behavior of a DC motor system (Fig. P6.3.2)

$$\begin{aligned}
 J \frac{d^2\theta(t)}{dt^2} + B \frac{d\theta(t)}{dt} &= T(t) = K_T i(t) \\
 L \frac{di(t)}{dt} + Ri(t) + K_b \frac{d\theta(t)}{dt} &= v(t)
 \end{aligned}
 \tag{P6.3.6}$$

Convert this system of equations into a first-order vector differential equation—that is, a state equation with respect to the state vector $[\theta(t) \ \theta'(t) \ i(t)]$.

(g) RC Circuit: A Stiff System

Consider a two-mesh RC circuit depicted in Fig. P6.3.3. We can write the mesh equation with respect to the two mesh currents $i_1(t)$ and $i_2(t)$ as



$$\begin{aligned}
 \text{back e.m.f. } v_b(t) &= K_b \omega(t) = K_b \theta'(t) \\
 \text{torque } T(t) &= K_T i(t)
 \end{aligned}$$

Figure P6.3.2 A DC motor system.

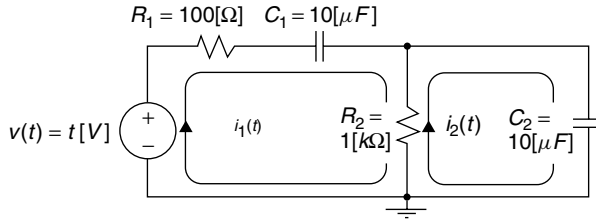


Figure P6.3.3 A two-mesh RC circuit.

$$R_1 i_1(t) + \frac{1}{C_1} \int_{-\infty}^t i_1(\tau) d\tau + R_2(i_1(t) - i_2(t)) = v(t) = t$$

$$R_2(i_2(t) - i_1(t)) + \frac{1}{C_2} \int_{-\infty}^t i_2(\tau) d\tau = 0 \quad (\text{P6.3.7a})$$

In order to convert this system of differential equations into a state equation, we differentiate both sides and rearrange them to get

$$(R_1 + R_2) \frac{di_1(t)}{dt} - R_2 \frac{di_2(t)}{dt} + \frac{1}{C_1} i_1(t) = \frac{dv(t)}{dt} = 1$$

$$-R_2 \frac{di_1(t)}{dt} + R_2 \frac{di_2(t)}{dt} + \frac{1}{C_2} i_2(t) = 0 \quad (\text{P6.3.7b})$$

$$\begin{bmatrix} R_1 + R_2 & -R_2 \\ -R_2 & R_2 \end{bmatrix} \begin{bmatrix} i_1'(t) \\ i_2'(t) \end{bmatrix} = \begin{bmatrix} 1 - i_1(t)/C_1 \\ -i_2(t)/C_2 \end{bmatrix} \quad (\text{P6.3.7c})$$

$$\begin{bmatrix} i_1'(t) \\ i_2'(t) \end{bmatrix} = \begin{bmatrix} R_1 + R_2 & -R_2 \\ -R_2 & R_2 \end{bmatrix}^{-1} \begin{bmatrix} 1 - i_1(t)/C_1 \\ -i_2(t)/C_2 \end{bmatrix} \text{ with } G_i = 1/R_i$$

$$= \begin{bmatrix} -G_1/C_1 & -G_1/C_2 \\ -G_1/C_1 & -(G_1 + G_2)/C_2 \end{bmatrix} \begin{bmatrix} i_1(t) \\ i_2(t) \end{bmatrix} + \begin{bmatrix} G_1 \\ G_1 \end{bmatrix} u_s(t) \quad (\text{P6.3.7d})$$

where $u_s(t)$ denotes the unit step function whose value is 1 (one) $\forall t \geq 0$.

- (i) After constructing an M-file function “df6p03g.m” which defines Eq. (P6.3.7d) with $R_1 = 100[\Omega]$, $C_1 = 10[\mu F]$, $R_2 = 1[k\Omega]$, $C_2 = 10[\mu F]$, use the MATLAB built-in routines “ode45()” and “ode23s()” to solve the state equation with the zero initial condition $i_1(0) = i_2(0) = 0$ and plot the numerical solution $i_2(t)$ for $0 \leq t \leq 0.05$ s. For possible change of parameters, you may declare R_1, C_1, R_2, C_2 as global variables both in the function and in the main program named, say, “nm6p03g.m”. Do you see any symptom of stiffness from the results?

- (ii) If we apply the Laplace transform technique to solve this equation with zero initial condition $\mathbf{i}(0) = \mathbf{0}$, we can get

$$\begin{aligned} \begin{bmatrix} I_1(s) \\ I_2(s) \end{bmatrix} &\stackrel{(6.5.5)}{=} [sI - A]^{-1} Bu(s) \\ &= \begin{bmatrix} s + G_1/C_1 & G_1/C_2 \\ G_1/C_2 & s + (G_1 + G_2)/C_2 \end{bmatrix}^{-1} \begin{bmatrix} G_1 \\ G_1 \end{bmatrix} \frac{1}{s} \\ I_2(s) &= \frac{G_1}{s^2 + (G_1/C_1 + (G_1 + G_2)/C_2)s + G_1G_2/C_1C_2} \\ &= \frac{1/100}{s^2 + 2100s + 100000} \\ &\cong \frac{1/100}{(s + 2051.25)(s + 48.75)} \\ &\cong \frac{1}{200250} \left(\frac{1}{s + 48.75} - \frac{1}{s + 2051.25} \right) \\ i_2(t) &\cong \frac{1}{200250} (e^{-48.75t} - e^{-2051.25t}) \end{aligned} \tag{P6.3.7e}$$

where $\lambda_1 = -2051.25$ and $\lambda_2 = -48.75$ are actually the eigenvalues of the system matrix A in Eq. (P6.3.7d). Find the measure of stiffness defined by Eq. (6.5.26).

- (iii) Using the MATLAB symbolic computation command “`dsolve()`”, find the analytical solution of the differential equation (P6.3.7b) and plot $i_2(t)$ together with (P6.3.7e) for $0 \leq t \leq 0.05$ s. Which of the two numerical solutions obtained in (i) is better? You may refer to the following code:

```

syms R1 R2 C1 C2
i = dsolve('(R1+R2)*Di1 - R2*Di2 + i1/C1 = 1',...
          '-R2*Di1 + R2*Di2 + i2/C2', 'i1(0) = 0', 'i2(0) = 0'); % (P6.3.7b)
R1 = 100; R2 = 1000; C1 = 1e-5; C2 = 1e-5;
t0 = 0; tf = 0.05; t = t0+(tf-t0)/100*[0:100];
i2t = eval(i.i2); plot(t,i2t,'m')
```

6.4 Physical Meaning of a Solution for Differential Equation and Its Animation

Suppose we are going to simulate how a vehicle vibrates when it moves with a constant speed on a rugged way, as depicted in Fig. P6.4a. Based on Newton’s second law, the situation is modeled by the differential equation (P6.4.1).

$$\begin{aligned} M \frac{d^2}{dt^2} y(t) + B \frac{d}{dt} (y(t) - u(t)) + K (y(t) - u(t)) &= 0 \tag{P6.4.1} \\ \text{with } y(0) &= 0, \quad y'(0) = 0 \end{aligned}$$

```

%do_MBK
clf
t0 = 0; tf = 10; x0 = [0 0];
[t1,x] = ode_Ham('f_MBK',[t0 tf],x0);
dt = t1(2) - t1(1);
for n = 1:length(t1)
    u(n) = udu_MBK(t1(n));
end
figure(1), clf
animation = 1;
if animation
    figure(2), clf
    draw_MBK(5,1,x(1,2),u(1))
    axis([-2 2 -1 14]), axis('equal')
    pause
    for n = 1:length(t1)
        clf, draw_MBK(5,1,x(n,2),u(n),'b')
        axis([-2 2 -1 14]), axis('equal')
        pause(dt)
        figure(1)
        plot(t1(n),u(n),'r.', t1(n),x(n,2),'b.')
        axis([0 tf -0.2 1.2]), hold on
        figure(2)
    end
    draw_MBK(5,1,x(n,2),u(n))
    axis([-2 2 -1 14]), axis('equal')
end

function [u,du] = udu_MBK(t)
i = fix(t);
if mod(i,2) == 0, u = t-i; du = 1;
    else u = 1 - t + i; du = -1;
end

function draw_MBK(n,w,y,u,color)
%n: the # of spring windings
#w: the width of each object
%y: displacement of the top of MBK
%u: displacement of the bottom of MBK
if nargin < 5, color = 'k'; end
p1 = [-w u + 4]; p2 = [-w 9 + y];
xm = 0; ym = (p1(2) + p2(2))/2;
xM = xm + w*1.2*[-1 -1 1 1 -1];
yM = p2(2) + w*[1 3 3 1 1];
plot(xM,yM,color), hold on %Mass
spring(n,p1,p2,w,color) %Spring
damper(xm + w,p1(2),p2(2),w,color) %Damper
wheel_my(xm,p1(2) - 3*w,w,color) %Wheel

function dx = f_MBK(t,x)
M = 1; B = 0.1; K = 0.1;
[u,du] = udu_MBK(t);
dx = x*[0 1; -B/M - K/M]'+[0 (K*u + B*du)/M];

```

```

function spring(n,p1,p2,w,color)
%draw a spring of n windings, width w from p1 to p2
if nargin < 5, color = 'k'; end
c = (p2(1) - p1(1))/2; d = (p2(2) - p1(2))/2;
f = (p2(1) + p1(1))/2; g = (p2(2) + p1(2))/2;
y = -1:0.01:1; t = (y+1)*pi*(n + 0.5);
x = -0.5*w*sin(t); y = y+0.15*(1 - cos(t));
a = y(1); b=y(length(x));
y = 2*(y - a)/(b - a)-1;
yyS = d*y - c*x + g; xxS = x+f; xxS1 = [f f];
yyS1 = yyS(length(yyS))+[0 w]; yyS2 = yyS(1)-[0 w];
plot(xxS,yyS,color, xxS1,yyS1,color, xxS1,yyS2,color)

function damper(xm,y1,y2,w,color)
%draws a damper in (xm-0.5 xm + 0.5 y1 y2)
if nargin < 5, color = 'k'; end
ym = (y1 + y2)/2;
xD1 = xm + w*[0.3*[0 0 -1 1]]; yD1 = [y2 + w ym ym ym];
xD2 = xm + w*[0.5*[-1 -1 1 1]]; yD2 = ym + w*[1 -1 -1
1];
xD3 = xm + [0 0]; yD3 = [y1 ym] - w;
plot(xD1,yD1,color, xD2,yD2,color, xD3,yD3,color)

function wheel_my(xm,ym,w,color)
%draws a wheel of size w at center (xm,ym)
if nargin < 5, color = 'k'; end
xW1 = xm + w*1.2*[-1 1]; yW1 = ym + w*[2 2];
xW2 = xm*[1 1]; yW2 = ym + w*[2 0];
plot(xW1,yW1,color, xW2,yW2,color)
th = [0:100]/50*pi; plot(xm + j*ym+w*exp(j*th),color)
    
```

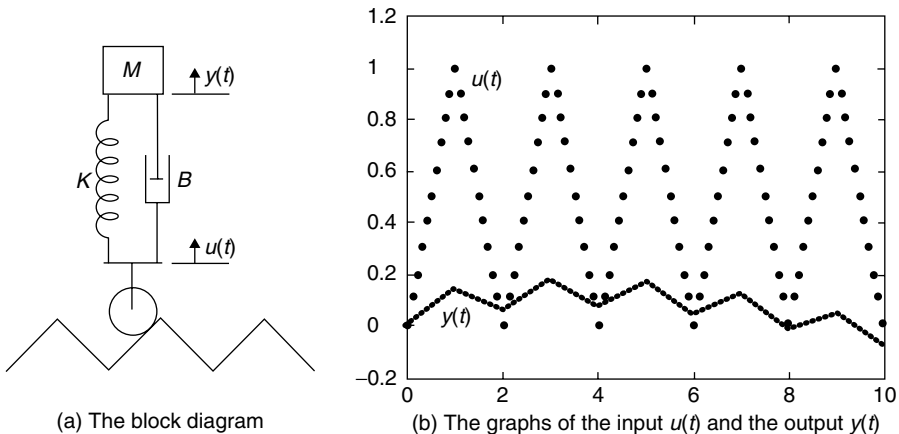


Figure P6.4 A mass–spring–damper system.

where the values of the mass, the viscous friction coefficient, and the spring constant are given as $M = 1$ kg, $B = 0.1$ N s/m, and $K = 0.1$ N/m, respectively. The input to this system is the movement $u(t)$ of the wheel part causing the movement $y(t)$ of the body as the output of the system and is approximated to a triangular wave of height 1 m, duration 1 s, and period 2 s as depicted in Fig. P6.4b. After converting this equation into a state equation as

$$\begin{bmatrix} x_1'(t) \\ x_2'(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -K/M & -B/M \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ (B/M)u'(t) + (K/M)u(t) \end{bmatrix} \tag{P6.4.2}$$

with $\begin{bmatrix} x_1(0) \\ x_2(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

we can use such routines as `ode_Ham()`, `ode45()`, ... to solve this state equation and use some graphic functions to draw not only the graphs of $y(t)$ and $u(t)$, but also the animated simulation diagram. You can run the above MATLAB program “do_MBK.m” to see the results. Does the suspension system made of a spring and a damper as depicted in Fig. P6.4a absorb effectively the shock caused by the rolling wheel so that the amplitude of vehicle body oscillation is less than 1/5 times that of wheel oscillation?

(cf) If one is interested in graphic visualization with MATLAB, he/she can refer to [N-1].

6.5 A Nonlinear Differential Equation for an Orbit of a Satellite

Consider the problem of an orbit of a satellite, whose position and velocity are obtained as the solution of the following state equation:

$$\begin{aligned} x_1'(t) &= x_3(t) \\ x_2'(t) &= x_4(t) \\ x_3'(t) &= -GM_E x_1(t)/(x_1^2(t) + x_2^2(t))^{3/2} \\ x_4'(t) &= -GM_E x_2(t)/(x_1^2(t) + x_2^2(t))^{3/2} \end{aligned} \tag{P6.5.1}$$

where $G = 6.672 \times 10^{-11}$ N m²/kg² is the gravitational constant, and $M_E = 5.97 \times 10^{24}$ kg is the mass of the earth. Note that (x_1, x_2) and (x_3, x_4) denote the position and velocity, respectively, of the satellite on the plane having the earth at its origin. This state equation is defined in the M-file ‘df_sat.m’ below.

(a) Supplement the following program “nm6p05.m” which uses the three routines `ode_RK4()`, `ode45()`, and `ode23()` to find the paths of the satellite with the following initial positions/velocities for one day.

```

function dx = df_sat(t,x)
global G Me Re
dx = zeros(size(x));
r = sqrt(sum(x(1:2).^2));
if r <= Re, return; end % when colliding against the earth surface
GMr3 = G*Me/r^3;
dx(1) = x(3); dx(2) = x(4); dx(3) = -GMr3*x(1); dx(4) = -GMr3*x(2);

%nm6p05.m to solve a nonlinear d.e. on the orbit of a satellite
clear, clf
global G Me Re
G = 6.67e-11; Me = 5.97e24; Re = 64e5;
f = 'df_sat'; ;
t0 = 0; T = 24*60*60; tf = T; N = 2000;
R = 4.223e7;
v20s = [3071 3500 2000];
for iter = 1:length(v20s)
    x10 = R; x20 = 0; v10 = 0; v20 = v20s(iter);
    x0 = [x10 x20 v10 v20]; tol = 1e-6;
    [tR,xR] = ode_RK4(f,[t0 tf],x0,N);
    [t45,x45] = ode45('df_sat');
    [t23s,x23s] = ode23s(f,[t0 tf],x0);
    plot(xR(:,1),xR(:,2),'b', x45(:,1),x45(:,2),'k.', 'df_sat');
    [t45,x45] = ode45(f,[t0 tf],x0,odeset('RelTol',tol));
    [t23s,x23s] = ode23s('df_sat');
    plot(xR(:,1),xR(:,2),'b', x45(:,1),x45(:,2),'k.', 'df_sat');
end
    
```

- (i) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)[m]$ and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0, 3071)[m/s]$.
- (ii) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)[m]$ and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0, 3500)[m/s]$.
- (iii) $(x_{10}, x_{20}) = (4.223 \times 10^7, 0)[m]$ and $(x_{30}, x_{40}) = (v_{10}, v_{20}) = (0, 2000)[m/s]$.

Run the program and check if the plotting results are as depicted in Fig. P6.5.

- (b) In Fig. P6.5, we see that the “ode23s()” solution path differs from the others for case (ii) and the “ode45()” and “ode23s()” paths differ from the “ode_RK4()” path for case (iii). But, we do not know which one is more accurate. In order to find which one is the closest to the true solution, apply the two routines “ode45()” and “ode23s()” with smaller relative error tolerance of $tol = 1e-6$ to find the paths for the three cases. Which one do you think is the closest to the true solution among the paths obtained in (a)?
- (cf) The purpose of this problem is not to compare the several MATLAB routines, but to warn the users of the danger of abusing them. With smaller number of steps (N) (i.e., larger step size), the routine “ode_RK4()” will also deviate much from the true solution. The MATLAB built-in routines have too many good features to be mentioned here. Note that setting the parameters such as

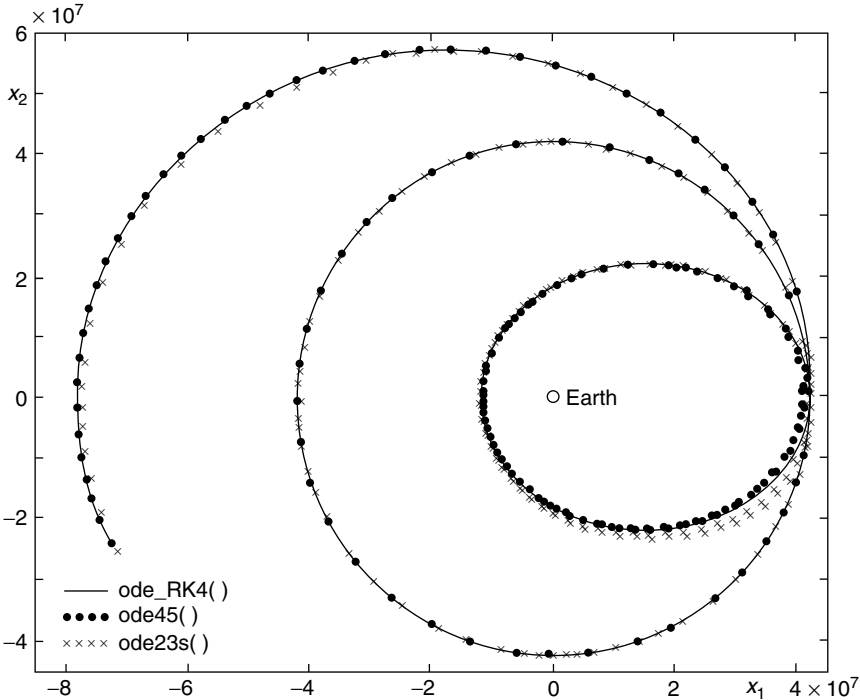


Figure P6.5 The paths of a satellite with the same initial position and different initial velocities.

the relative error tolerance (`RelTol`) is sometimes very important for obtaining a reasonably accurate solution.

6.6 Shooting Method for BVP with Adjustable Position and Fixed Angle

Suppose the boundary condition for a second-order BVP is given as

$$x'(t_0) = x_{20}, \quad x(t_f) = x_{1f} \tag{P6.6.1}$$

Consider how to modify the MATLAB routines “`bvp2_shoot()`” and “`bvp2_fdf()`” so that they can accommodate this kind of problem.

- (a) As for “`bvp2_shootp()`” that you should make, the variable quantity to adjust for improving the approximate solution is not the derivative $x'(t_0)$, but the position $x(t_0)$ and what should be made close to zero is still $f(x(t_0)) = x(t_f) - x_f$. Modify the routine in such a way that $x(t_0)$ is adjusted to make this quantity close to zero and make its declaration part have the initial derivative (`dx0`) instead of the initial position (`x0`) as the fourth input argument as follows.

```
function [t,x] = bvp2_shootp(f,t0,tf,dx0,xf,N,tol,kmax)
```

Noting that the initial derivative of the true solution for Eq. (6.6.3) is zero, apply this routine to solve the BVP by inserting the following statement into the program “do_shoot.m”.

```
[t,x1] = bvp2_shootp('df661',t0,tf,0,xf,N,tol,kmax);
```

and plot the result to check if it conforms with that (Fig. 6.8) obtained by “bvp2_shoot()”.

- (b) As for “bvp2_fdfp()” implementing the finite difference method, you have to approximate the boundary condition as

$$x'(t_0) = x_{20} \rightarrow \frac{x_1 - x_{-1}}{2h} = x_{20}, \quad x_{-1} = x_1 - 2hx_{20}, \quad (\text{P6.6.2})$$

substitute this into the finite difference equation corresponding to the initial time as

$$\frac{x_1 - 2x_0 + x_{-1}}{h^2} + a_{10} \frac{x_1 - x_{-1}}{2h} + a_{00}x_0 = u_0 \quad (\text{P6.6.3})$$

$$\frac{x_1 - 2x_0 + x_1 - 2hx_{20}}{h^2} + a_{10}x_{20} + a_{00}x_0 = u_0$$

$$(a_{00}h^2 - 2)x_0 + 2x_1 = h^2u_0 + h(2 - ha_{10})x_{20} \quad (\text{P6.6.4})$$

and augment the matrix–vector equation with this equation. Also, make its declaration part have the initial derivative (dx0) instead of the initial position (x0) as the sixth input argument as follows:

```
function [t,x] = bvp2_fdfp(a1,a0,u,t0,tf,dx0,xf,N)
```

Noting that the initial derivative of the true solution for Eq. (6.6.10) is -7 , apply this routine to solve the BVP by inserting the following statement into the program “do_fdf.m”.

```
[t,x1] = bvp2_fdfp(a1,a0,u,t0,tf,-7,xf,N);
```

and plot the result to check if it conforms with that obtained by using “bvp2_fdf()” and depicted in Fig. 6.9.

6.7 BVP with Mixed-Boundary Conditions I

Suppose the boundary condition for a second-order BVP is given as

$$x(t_0) = x_{10}, \quad c_1x(t_f) + c_2x'(t_f) = c_3 \quad (\text{P6.7.1})$$

Consider how to modify the MATLAB routines “bvp2_shoot()” and “bvp2_fdf()” so that they can accommodate this kind of problem.

- (a) As for “bvp2_shoot()” that you should modify, the variable quantity to adjust for improving the approximate solution is still the derivative $x'(t_0)$, but what should be made close to zero is

$$f(x'(t_0)) = c_1x(t_f) + c_2x'(t_f) - c_3 \quad (\text{P6.7.2})$$

If you don't know where to begin, modify the routine “bvp2_shoot()” in such a way that $x'(t_0)$ is adjusted to make this quantity close to zero. Regarding the quantity (P6.7.2) as a function of $x'(t_0)$, you may feel as if you were going to solve a nonlinear equation $f(x'(t_0)) = 0$. Here are a few hints for this job:

- Make the declaration part have the boundary coefficient vector $cf = [c_1 \ c_2 \ c_3]$ instead of the final position (xf) as the fifth input argument as follows.

```
function [t,x] = bvp2m_shoot(f,t0,tf,x0,cf,N,tol,kmax)
```

- Pick up the first two guesses of $x'(t_0)$ arbitrarily.
- You may need to replace a couple of statements in “bvp2_shoot()” by

```
e(1) = cf*[x(end,:)';-1];
e(k) = cf*[x(end,:)';-1];
```

Now that you have the routine “bvp2m_shoot()” of your own making, don't hesitate to try using the weapon to attack the following problem:

$$x''(t) - 4t x(t)x'(t) + 2x^2(t) = 0 \quad \text{with } x(0) = \frac{1}{4}, 2x(1) - 3x'(1) = 0 \quad (\text{P6.7.3})$$

For this job, you only have to modify one statement of the program “do_shoot” (Section 6.6.1) into

```
[t,x] = bvp2m_shoot('df661',t0,tf,x0,[2 -3 0],N,tol,kmax);
```

If you run it to obtain the same solution as depicted in Fig. 6.8, you deserve to be proud of yourself having this book as well as MATLAB; otherwise, just keep trying until you succeed.

- (b) As for “bvp2_fdf()” that you should modify, you have only to augment the matrix–vector equation with one row corresponding to the approximate version of the boundary condition $c_1x(t_f) + c_2x'(t_f) = c_3$, that is,

$$c_1x_N + c_2 \frac{x_N - x_{N-1}}{h} = c_3; \quad -c_2x_{N-1} + (c_1h + c_2)x_N = c_3h \quad (\text{P6.7.4})$$

Needless to say, you should increase the dimension of the matrix A to N and move the x_N term on the right-hand side of the $(N - 1)$ th row back to the left-hand side by incorporating the corresponding statement into the `for` loop. What you have to do with “`bvp2m_fdf()`” for this job is as follows:

- Make the declaration part have the boundary coefficient vector `cf = [c1 c2 c3]` instead of the final position (`xf`) as the seventh input argument.

```
function [t,x] = bvp2m_fdf(a1,a0,u,t0,tf,x0,cf,N)
```

- Replace some statement by `A = zeros(N,N)`.
- Increase the last index of the `for` loop to `N-1`.
- Replace the statements corresponding to the $(N - 1)$ th row equation by

```
A(N,N-1:N) = [-cf(2) cf(1)*h + cf(2)];    b(N) = cf(3)*h;
```

which implements Eq. (P6.7.4).

- Modify the last statement arranging the solution as

```
x = [x0 trid(A,b)']';
```

Now that you have the routine “`bvp2m_fdf()`” of your own making, don’t hesitate to try it on the following problem:

$$x''(t) + \frac{2}{t}x'(t) - \frac{2}{t^2}x(t) = 0 \quad \text{with } x(1) = 5, x(2) + x'(2) = 3 \tag{P6.7.5}$$

For this job, you only have to modify one statement of the program “`do_fdf.m`” (Section 6.6.2) into

```
[t,x] = bvp2m_fdf(a1,a0,u,t0,tf,x0,[1 1 3],N);
```

You might need to increase the number of segments N to improve the accuracy of the numerical solution. If you run it to obtain the same solution as depicted in Fig. 6.9, be happy with it.

6.8 BVP with Mixed-Boundary Conditions II

Suppose the boundary condition for a second-order BVP is given as

$$c_{01}x(t_0) + c_{02}x'(t_0) = c_{03} \tag{P6.8.1a}$$

$$c_{f1}x(t_f) + c_{f2}x'(t_f) = c_{f3} \tag{P6.8.1b}$$

Consider how to modify the MATLAB routines “`bvp2m_shoot()`” and “`bvp2m_fdf()`” so that they can accommodate this kind of problems.

- (a) As for “bvp2mm_shoot ()” that you should make, the variable quantity to be adjusted for improving the approximate solution is $x'(t_0)$ or $x(t_0)$ depending on whether or not $c_{01} \neq 0$, while the quantity to be made close to zero is still

$$f(x(t_0), x'(t_0)) = c_{f1}x(t_f) + c_{f2}x'(t_f) - c_{f3} \quad (\text{P6.8.2})$$

If you don't have your own idea, modify the routine “bvp2m_shoot ()” in such a way that $x'(t_0)$ or $x(t_0)$ is adjusted to make this quantity close to zero and $x(t_0)$ or $x'(t_0)$ is set by (P6.8.1a), making its declaration as

```
function [t,x] = bvp2mm_shoot(f,t0,tf,c0,cf,N,tol,kmax)
```

where the boundary coefficient vectors $c_0 = [c_{01} \ c_{02} \ c_{03}]$ and $c_f = [c_{f1} \ c_{f2} \ c_{f3}]$ are supposed to be given as the fourth and fifth input arguments, respectively.

Now that you get the routine “bvp2mm_shoot ()” of your own making, try it on the following problem:

$$x''(t) - \frac{2t}{t^2 + 1}x'(t) + \frac{2}{t^2 + 1}x(t) = t^2 + 1 \quad (\text{P6.8.3})$$

$$\text{with } x(0) + 6x'(0) = 0, \ x(1) + x'(1) = 0$$

- (b) As for “bvp2_fdf ()” implementing the finite difference method, you only have to augment the matrix–vector equation with two rows corresponding to the approximate versions of the boundary conditions $c_{01}x(t_0) + c_{02}x'(t_0) = c_{03}$ and $c_{f1}x(t_f) + c_{f2}x'(t_f) = c_{f3}$, that is,

$$c_{01}x_0 + c_{02}\frac{x_1 - x_0}{h} = c_{03}, \quad (c_{01}h - c_{02})x_0 + c_{02}x_1 = c_{03}h \quad (\text{P6.8.4a})$$

$$c_{f1}x_N + c_{f2}\frac{x_N - x_{N-1}}{h} = c_{f3}; \quad -c_{f2}x_{N-1} + (c_{f1}h + c_{f2})x_N = c_{f3}h \quad (\text{P6.8.4b})$$

Now that you have the routine “bvp2mm_fdf ()” of your own making, try it on the problem described by Eq. (P6.8.3).

- (c) Overall, you will need to make the main programs like “nm6p08a.m” and “nm6p08b.m” that apply the routines “bvp2mm_shoot ()” and “bvp2mm_fdf ()” to get the numerical solutions of Eq. (P6.8.3) and plot them. Additionally, use the MATLAB routine “bvp4c ()” to get another solution and plot it together for cross-check.

6.9 Shooting Method and Finite Difference Method for Linear BVPs

Apply the routines “bvp2_shoot ()”, “bvp2_fdf ()”, and “bvp4c ()” to solve the following BVPs.

```
%nm6p08a.m: to solve BVP2 with mixed boundary conditions
%x" = (2t/t^2 + 1)*x' -2/(t^2+1)*x +t^2+1
% with x(0)+6x'(0) = 0, x'(1) + x(1) = 0
%shooting method
f = inline('[x(2); 2*(t*x(2) - x(1))./(t.^2 + 1)+(t.^2 + 1)]','t','x');
t0 = 0; tf = 1; N = 100; tol = 1e-8; kmax = 10;
c0 = [1 6 0]; cf = [1 1 0]; %coefficient vectors of boundary condition
[tt,x_sh] = bvp2mm_shoot(f,t0,tf,c0,cf,N,tol,kmax);
plot(tt,x_sh(:,1),'b')
```

```
%nm6p08b.m: finite difference method
a1 = inline('-2*t./(t.^2+1)','t'); a0 = inline('2./(t.^2+1)','t');
u = inline('t.^2+1','t');
t0 = 0; tf = 1; N = 500;
c0 = [1 6 0]; cf = [1 1 0]; %coefficient vectors of boundary condition
[tt,x_fd] = bvp2mm_fdf(a1,a0,u,t0,tf,c0,cf,N);
plot(tt,x_fd,'r')
```

$$y''(x) = f(y'(x), y(x), u(x)) \quad \text{with } y(x_0) = y_0, y(x_f) = y_f \quad (\text{P6.9.0a})$$

Plot the solutions and fill in Table P6.9 with the mismatching errors (of the numerical solutions) that are defined as

```
function err = err_of_sol_de(df,t,x,varargin)
% evaluate the error of solutions of differential equation
[Nt,Nx] = size(x); if Nt < Nx, x = x.'; [Nt,Nx] = size(x); end
n1 = 2:Nt - 1; t=t(:); h2s = t(n1 + 1)-t(n1-1);
dx = (x(n1 + 1,:) - x(n1 - 1,:))./(h2s*ones(1,Nx));
num = x(n1 + 1,:)-2*x(n1,:) + x(n1 - 1,:); den = (h2s/2).^2*ones(1,Nx);
d2x = num./den;
for m = 1:Nx
    for n = n1(1):n1(end)
        dfx = feval(df,t(n),[x(n,m) dx(n - 1,m)],varargin{:});
        errm(n - 1,m) = d2x(n - 1,m) - dfx(end);
    end
end
err=sum(errm.^2)/(Nt - 2);
```

```
%nm6p09_1.m
%y"-y'+y = 3*e^2t-2sin(t) with y(0) = 5 & y(2)=-10
t0 = 0; tf = 2; y0 = 5; yf = -10; N = 100; tol = 1e-6; kmax = 10;
df = inline('[y(2); y(2) - y(1)+3*exp(2*t)-2*sin(t)]','t','y');
a1 = -1; a0 = 1; u = inline('3*exp(2*t) - 2*sin(t)','t');
solinit = bvpinit(linspace(t0,tf,5),[-10 5]); % [1 9]
fbc = inline('[y0(1) - 5; yf(1) + 10]','y0','yf');
% Shooting method
tic, [tt,y_sh] = bvp2_shoot(df,t0,tf,y0,yf,N,tol,kmax); times(1) = toc;
% Finite difference method
tic, [tt,y_fd] = bvp2_fdf(a1,a0,u,t0,tf,y0,yf,N); times(2) = toc;
% MATLAB built-in function bvp4c
sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-6));
tic, y_bvp = deval(sol,tt); times(3) = toc
% Error evaluation
ys=[y_sh(:,1) y_fd y_bvp(1,:)']; plot(tt,ys)
err=err_of_sol_de(df,tt,ys)
```

Table P6.9 Comparison of the BVP Solver Routines `bvp2_shoot()`/`bvp2_fdf()`

BVP	Routine	Mismatching Error (P6.9.0b)	Times
(P6.9.1) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>	1.5×10^{-6}	
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	2.9×10^{-6}	
(P6.9.2) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	1.6×10^{-23}	
	<code>bvp4c()</code>		
(P6.9.3) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>	1.7×10^{-17}	
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	7.8×10^{-14}	
(P6.9.4) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	4.4×10^{-27}	
	<code>bvp4c()</code>		
(P6.9.5) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>	8.9×10^{-9}	
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	8.9×10^{-7}	
(P6.9.6) N = 100, tol = 1e-6, kmax = 10	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	4.4×10^{-25}	
	<code>bvp4c()</code>		

$$\text{err} = \frac{1}{N-1} \sum_{i=1}^{N-1} \{D^{(2)}y(x_i) - f(Dy(x_i), y(x_i), u(x_i))\}^2 \quad (\text{P6.9.0b})$$

with

$$D^{(2)}y(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1}))}{h^2}, \quad Dy(x_i) = \frac{y(x_{i+1}) - y(x_{i-1}))}{2h} \quad (\text{P6.9.0c})$$

$$x_i = x_0 + ih, \quad h = \frac{x_f - x_0}{N} \quad (\text{P6.9.0d})$$

and can be computed by using the following routine “`err_of_sol_de()`”.

Overall, which routine works the best for linear BVPs among the three routines?

$$(a) \quad y''(x) = y'(x) - y(x) + 3e^{2x} - 2 \sin x \quad \text{with } y(0) = 5, y(2) = -10 \quad (\text{P6.9.1})$$

$$(b) \quad y''(x) = -4y(x) \quad \text{with } y(0) = 5, y(1) = -5 \quad (\text{P6.9.2})$$

$$(c) \quad y''(t) = 10^{-6}y(t) + 10^{-7}(t^2 - 50t) \quad \text{with } y(0) = 0, y(50) = 0 \quad (\text{P6.9.3})$$

$$(d) \quad y''(t) = -2y(t) + \sin t \quad \text{with } y(0) = 0, y(1) = 0 \quad (\text{P6.9.4})$$

$$(e) \quad y''(x) = y'(x) + y(x) + e^x(1 - 2x) \quad \text{with } y(0) = 1, y(1) = 3e \quad (\text{P6.9.5})$$

$$(f) \quad \frac{d^2y(r)}{dr^2} + \frac{1}{r} \frac{dy(r)}{dr} = 0 \quad \text{with } y(1) = \ln 1, y(2) = \ln 2 \quad (\text{P6.9.6})$$

6.10 Shooting Method and Finite Difference Method for Nonlinear BVPs

(a) Consider a nonlinear boundary value problem of solving

$$\frac{d^2T}{dx^2} = 1.9 \times 10^{-9}(T^4 - T_a^4), \quad T_a = 400 \quad (\text{P6.10.1})$$

with the boundary condition $T(x_0) = T_0, T(x_f) = T_f$

to find the temperature distribution $T(x)$ [$^{\circ}\text{K}$] in a rod 4 m long, where $[x_0, x_f] = [0, 4]$.

Apply the routines “bvp2_shoot()”, “bvp2_fdf()”, and “bvp4c()” to solve this differential equation for the two sets of boundary conditions $\{T(0) = 500, T(4) = 300\}$ and $\{T(0) = 550, T(4) = 300\}$ as listed in Table P6.10. Fill in the table with the mismatching errors defined by Eq. (P6.9.0b) for the three numerical solutions

```
%nm6p10a
clear, clf
K = 1.9e-9; Ta = 400; Ta4 = Ta^4;
df = inline('[T(2); 1.9e-9*(T(1).^4-256e8)]','t','T');
x0 = 0; xf = 4; T0 = 500; Tf = 300; N = 500; tol = 1e-5; kmax = 10;
% Shooting method
[xx,T_sh] = bvp2_shoot(df,x0,xf,T0,Tf,N,tol,kmax);
% Iterative finite difference method
a1 = 0; a0 = 0; u = T0 + [1:N - 1]*(Tf - T0)/N;
for i = 1:100
    [xx,T_fd] = bvp2_fdf(a1,a0,u,x0,xf,T0,Tf,N);
    u = K*(T_fd(2:N).^4 - Ta4); %RHS of (P6.10.1)
    if i > 1 & norm(T_fd - T_fd0)/norm(T_fd0) < tol, i, break; end
    T_fd0 = T_fd;
end
% MATLAB built-in function bvp4c
solinit = bvpinit(linspace(x0,xf,5),[Tf T0]);
fbc = inline('[Ta(1)-500; Tb(1)-300]','Ta','Tb');
tic, sol = bvp4c(df,fbc,solinit,bvpset('RelTol',1e-6));
T_bvp = deval(sol,xx); time_bvp = toc;
% The set of three solutions
Ts = [T_sh(:,1) T_fd T_bvp(1,:)'];
% Evaluates the errors and plot the graphs of the solutions
err = err_of_sol_de(df,xx,ys)
subplot(321), plot(xx,Ts)
```

Table P6.10 Comparison of the BVP routines `bvp2_shoot()`/`bvp2_fdf()`

Boundary Condition	Routine	Mismatching Error (P6.9.0b)	Time (seconds)
(P6.10.1) with $T_a = 400$ $T(0) = 500, T(4) = 300$	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	3.6×10^{-6}	
	<code>bvp4c()</code>		
(P6.10.1) with $T_a = 400$ $T(0) = 550, T(4) = 300$	<code>bvp2_shoot()</code>	NaN (divergent)	N/A
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	30×10^{-5}	
(P6.10.2) with $y(0) = 0, y(1) = 0$	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	3.2×10^{-13}	
	<code>bvp4c()</code>		
(P6.10.3) with $y(1) = 4, y(2) = 8$	<code>bvp2_shoot()</code>	NaN (divergent)	N/A
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	3.5×10^{-6}	
(P6.10.4) with $y(1) = 1/3, y(4) = 20/3$	<code>bvp2_shoot()</code>		
	<code>bvp4c()</code>	3.4×10^{-10}	
	<code>bvp2_fdf(c)</code>		
(P6.10.5) with $y(0) = \pi/2, y(2) = \pi/4$	<code>bvp2_shoot()</code>	3.7×10^{-14}	
	<code>bvp2_fdf()</code>		
	<code>bvp4c()</code>	2.2×10^{-9}	
(P6.10-6) with $y(2) = 2, y'(8) = 1/4$	<code>bvp2_shoot()</code>		
	<code>bvp2_fdf()</code>	5.0×10^{-14}	
	<code>bvp4c()</code>		

$$\{T(x_i), i = 0 : N\} \quad (x_i = x_0 + ih = x_0 + i \frac{x_f - x_0}{N} \text{ with } N = 500$$

Note that the routine “`bvp2_fdf()`” should be applied in an iterative way to solve a nonlinear BVP, because it has been fabricated to accommodate only linear BVPs. You may start with the following program “`nm6p10a.m`”. Which routine works the best for the first case and the second case, respectively?

- (b) Apply the routines “bvp2_shoot()”, “bvp2_fdf()”, and “bvp4c()” to solve the following BVPs. Fill in Table P6.10 with the mismatching errors defined by Eq. (P6.9.0b) for the three numerical solutions and plot the solution graphs if they are reasonable solutions.

$$(i) \quad y'' - e^y = 0 \quad \text{with } y(0) = 0, y(1) = 0 \quad (\text{P6.10.2})$$

$$(ii) \quad y'' - \frac{1}{t}y' - \frac{2}{y}(y')^2 = 0 \quad \text{with } y(1) = 4, y(2) = 8 \quad (\text{P6.10.3})$$

$$(iii) \quad y'' - \frac{2}{y' + 1} = 0 \quad \text{with } y(1) = \frac{1}{3}, y(4) = \frac{20}{3} \quad (\text{P6.10.4})$$

$$(iv) \quad y'' = t(y')^2 \quad \text{with } y(0) = \pi/2, y(2) = \pi/4 \quad (\text{P6.10.5})$$

$$(v) \quad y'' + \frac{1}{y^2}y' = 0 \quad \text{with } y(2) = 2, y'(8) = 1/4 \quad (\text{P6.10.6})$$

Especially for the BVP (P6.10.6), the routine “bvp2m_shoot()” or “bvp2mm_shoot()” developed in Problems 6.7 and 6.8 should be used instead of “bvp2_shoot()”, since it has a mixed-boundary condition I.

- (cf) Originally, the shooting method was developed for solving nonlinear BVPs, while the finite difference method is designed as a one-shot method for solving linear BVPs. But the finite difference method can also be applied in an iterative way to handle nonlinear BVPs, producing more accurate solutions in less computation time.

6.11 Eigenvalue BVPs

- (a) A Homogeneous Second-Order BVP to an Eigenvalue Problem
Consider an eigenvalue boundary value problem of solving

$$y''(x) + \omega^2 y = 0 \quad (\text{P6.11.1})$$

$$\text{with } c_{01}y(x_0) + c_{02}y'(x_0) = 0, \quad c_{f1}y(x_f) + c_{f2}y'(x_f) = 0$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω .

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$ and then, replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations (5.3.1) and (5.1.8) to write

$$\frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + \omega^2 y_i = 0$$

$$y_{i-1} - (2 - \lambda)y_i + y_{i+1} = 0 \quad \text{with } \lambda = h^2 \omega^2 \quad (\text{P6.11.2})$$

with

$$c_{01}y_0 + c_{02} \frac{y_1 - y_{-1}}{2h} = 0 \rightarrow y_{-1} = 2h \frac{c_{01}}{c_{02}} y_0 + y_1 \quad (\text{P6.11.3a})$$

$$c_{f1}y_N + c_{f2} \frac{y_{N+1} - y_{N-1}}{2h} = 0 \rightarrow y_{N+1} = y_{N-1} - 2h \frac{c_{f1}}{c_{f2}} y_N \quad (\text{P6.11.3b})$$

Substituting the discretized boundary condition (P6.11.3) into (P6.11.2) yields

$$y_{-1} - 2y_0 + y_1 = -\lambda y_0 \xrightarrow{(\text{P6.11.3a})}$$

$$\left(2 - 2h \frac{c_{01}}{c_{02}}\right) y_0 - 2y_1 = \lambda y_0 \quad (\text{P6.11.4a})$$

$$y_{i-1} - 2y_i + y_{i+1} = -\lambda y_i \rightarrow -y_{i-1} + 2y_i - y_{i+1} = \lambda y_i$$

$$\text{for } i = 1 : N - 1 \quad (\text{P6.11.4b})$$

$$y_{N-1} - 2y_N + y_{N+1} = -\lambda y_N \xrightarrow{(\text{P6.11.3b})}$$

$$-2y_{N-1} + \left(2 + 2h \frac{c_{f1}}{c_{f2}}\right) y_N = \lambda y_N \quad (\text{P6.11.4c})$$

which can be formulated in a compact form as

$$\begin{bmatrix} 2 - 2hc_{01}/c_{02} & -2 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -2 & 2 + 2hc_{f1}/c_{f2} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ y_{N-1} \\ y_N \end{bmatrix}$$

$$= \lambda \begin{bmatrix} y_0 \\ y_1 \\ \cdot \\ y_{N-1} \\ y_N \end{bmatrix}$$

$$\mathbf{Ay} = \lambda \mathbf{y}; \quad [A - \lambda I] \mathbf{y} = \mathbf{0} \quad (\text{P6.11.5})$$

For this equation to have a nontrivial solution $\mathbf{y} \neq \mathbf{0}$, λ must be one of the eigenvalues of the matrix A and the corresponding eigenvectors are possible solutions.

```

function [x,Y,ws,eigvals] = bvp2_eig(x0,xf,c0,cf,N)
% use the finite difference method to solve an eigenvalue BVP4:
% y''+w^2*y = 0 with c01y(x0) + c02y'(x0) = 0, cf1y(xf) + cf2y'(xf) = 0
%input: x0/xf = the initial/final boundaries
%      c0/cf = the initial/final boundary condition coefficients
%      N - 1 = the number of internal grid points.
%output: x = the vector of grid points
%      Y = the matrix composed of the eigenvector solutions
%      ws = angular frequencies corresponding to eigenvalues
%      eigvals = the eigenvalues
if nargin < 5 | N < 3, N = 3; end
h = (xf - x0)/N; h2 = h*h; x = x0+[0:N]*h;
N1 = N + 1;
if abs(c0(2)) < eps, N1 = N1 - 1; A(1,1:2) = [2 -1];
else A(1,1:2) = [2*(1-c0(1)/c0(2)*h) -2]; % (P6.11.4a)
end
if abs(cf(2)) < eps, N1 = N1 - 1; A(N1,N1 - 1:N1) = [-1 2];
else A(N1,N1 - 1:N1) = [-2 2*(1 + cf(1)/cf(2)*h)]; % (P6.11.4c)
end
if N1 > 2
for m = 2:ceil(N1/2), A(m,m - 1:m + 1) = [-1 2 -1]; end % (P6.11.4b)
end
for m=ceil(N1/2) + 1:N1 - 1, A(m,:) = fliplr(A(N1 + 1 - m,:)); end
[V,LAMBDA] = eig(A); eigvals = diag(LAMBDA)';
[eigvals,I] = sort(eigvals); % sorting in the ascending order
V = V(:,I);
ws = sqrt(eigvals)/h;
if abs(c0(2)) < eps, Y = zeros(1,N1); else Y = []; end
Y = [Y; V];
if abs(cf(2)) < eps, Y = [Y; zeros(1,N1)]; end

```

Note the following things:

- The angular frequency corresponding to the eigenvalue λ can be obtained as

$$\omega = \sqrt{\lambda/a_0}/h \quad (\text{P6.11.6})$$

- The eigenvalues and the eigenvectors of a matrix A can be obtained by using the MATLAB command '[V,D] = eig(A)'.
- The above routine "bvp2_eig()" implements the above-mentioned scheme to solve the second-order eigenvalue problem (P6.11.1).
- In particular, a second-order eigenvalue BVP

$$y''(x) + \omega^2 y = 0 \quad \text{with } y(x_0) = 0, y(x_f) = 0 \quad (\text{P6.11.7})$$

corresponds to (P6.11.1) with $\mathbf{c}_0 = [c_{01} \ c_{02}] = [1 \ 0]$ and $\mathbf{c}_f = [c_{f1} \ c_{f2}] = [1 \ 0]$ and has the following analytical solutions:

$$y(x) = a \sin \omega x \quad \text{with } \omega = \frac{k\pi}{x_f - x_0}, k = 1, 2, \dots \quad (\text{P6.11.8})$$

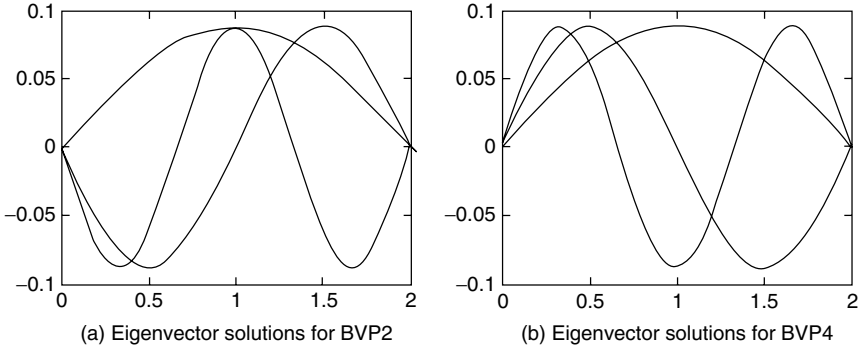


Figure P6.11 The eigenvector solutions of homogeneous second-order and fourth-order BVPs.

Now, use the routine “bvp2_eig ()” with the number of grid points $N = 256$ to solve the BVP2 (P6.11.7) with $x_0 = 0$ and $x_f = 2$, find the lowest three angular frequencies (ω_i 's) and plot the corresponding eigenvector solutions as depicted in Fig. P6.11a.

- (b) **A Homogeneous Fourth-Order BVP to an Eigenvalue Problem**
 Consider an eigenvalue boundary value problem of solving

$$\frac{d^4 y}{dx^4} - \omega^4 y = 0 \tag{P6.11.9}$$

$$\text{with } y(x_0) = 0, \frac{d^2 y}{dx^2}(x_0) = 0, y(x_f) = 0, \frac{d^2 y}{dx^2}(x_f) = 0$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω .

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$ and then, replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations to write

$$\frac{y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}}{h^4} - \omega^4 y_i = 0$$

$$y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2} = \lambda y_i (\lambda = h^4 \omega^4) \tag{P6.11.10}$$

with

$$y_0 = 0, \frac{y_{-1} - 2y_0 + y_1}{h^2} = 0 \rightarrow y_{-1} = -y_1 \tag{P6.11.11a}$$

$$y_N = 0, \frac{y_{N-1} - 2y_N + y_{N+1}}{h^2} = 0 \rightarrow y_{N+1} = -y_{N-1} \tag{P6.11.11b}$$

Substituting the discretized boundary condition (P6.11.11) into (P6.11.10) yields

$$\begin{aligned}
 y_{-1} - 4y_0 + 6y_1 - 4y_2 + y_3 &= \lambda y_1 \xrightarrow{\text{(P6.11.11a)}} \\
 5y_1 - 4y_2 + y_3 &= \lambda y_1 \\
 y_0 - 4y_1 + 6y_2 - 4y_3 + y_4 &= \lambda y_2 \xrightarrow{\text{(P6.11.11a)}} \\
 -4y_1 + 6y_2 - 4y_3 + y_4 &= \lambda y_2 \\
 y_i - 4y_{i+1} + 6y_{i+2} - 4y_{i+3} + y_{i+4} &= \lambda y_{i+2} \\
 \text{for } i = 1 : N - 5 & \qquad \qquad \qquad \text{(P6.11.12)} \\
 y_{N-4} - 4y_{N-3} + 6y_{N-2} - 4y_{N-1} + y_N &= \lambda y_{N-2} \xrightarrow{\text{(P6.11.11b)}} \\
 y_{N-4} - 4y_{N-3} + 6y_{N-2} - 4y_{N-1} &= \lambda y_{N-2} \\
 y_{N-3} - 4y_{N-2} + 6y_{N-1} - 4y_N + y_{N+1} &= \lambda y_{N-1} \xrightarrow{\text{(P6.11.11b)}} \\
 y_{N-3} - 4y_{N-2} + 5y_{N-1} &= \lambda y_{N-1}
 \end{aligned}$$

which can be formulated in a compact form as

$$\begin{aligned}
 \begin{bmatrix} 5 & -4 & 1 & 0 & 0 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 \\ 0 & \cdot & \cdot & \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & 0 & 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 0 & 0 & 1 & -4 & 5 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ y_{N-3} \\ y_{N-2} \\ y_{N-1} \end{bmatrix} &= \lambda \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \cdot \\ y_{N-3} \\ y_{N-2} \\ y_{N-1} \end{bmatrix} \\
 \mathbf{A}\mathbf{y} = \lambda\mathbf{y}, \quad [\mathbf{A} - \lambda\mathbf{I}]\mathbf{y} = \mathbf{0} & \qquad \qquad \qquad \text{(P6.11.13)}
 \end{aligned}$$

For this equation to have a nontrivial solution $\mathbf{y} \neq \mathbf{0}$, λ must be one of the eigenvalues of the matrix \mathbf{A} and the corresponding eigenvectors are possible solutions. Note that the angular frequency corresponding to the eigenvalue λ can be obtained as

$$\omega = \sqrt[4]{\lambda}/h \qquad \qquad \qquad \text{(P6.11.14)}$$

- (i) Compose a routine “bvp4_eig()” which implements the above-mentioned scheme to solve the fourth-order eigenvalue problem (P6.11.9).

```
function [x,Y,ws,eigvals] = bvp4_eig(x0,xf,N)
```

- (ii) Use the routine “bvp4_eig()” with the number of grid points $N = 256$ to solve the BVP4 (P6.11.9) with $x_0 = 0$ and $x_f = 2$, find the lowest three angular frequencies (ω_i 's) and plot the corresponding eigenvector solutions as depicted in Fig. P6.11b.
- (c) The Sturm–Liouville Equation

Consider an eigenvalue boundary value problem of solving

$$\frac{d}{dx}(f(x)y') + r(x)y = \lambda q(x)y \quad \text{with } y(x_0) = 0, y(x_f) = 0 \tag{P6.11.15}$$

to find $y(x)$ for $x \in [x_0, x_f]$ with the (possible) angular frequency ω .

In order to use the finite difference method, we divide the solution interval $[x_0, x_f]$ into N subintervals to have the grid points $x_i = x_0 + ih = x_0 + i(x_f - x_0)/N$, and then we replace the derivatives in the differential equation and the boundary conditions by their finite difference approximations (with the step size $h/2$) to write

$$\begin{aligned} & \frac{f(x_i + h/2)y'(x_i + h/2) - f(x_i - h/2)y'(x_i - h/2)}{2(h/2)} + r(x_i)y_i = \lambda q(x_i)y(x_i) \\ & \frac{1}{h} \left\{ f\left(x_i + \frac{h}{2}\right) \frac{y_{i+1} - y_i}{h} - f\left(x_i - \frac{h}{2}\right) \frac{y_i - y_{i-1}}{h} \right\} + r(x_i)y_i = \lambda q(x_i)y(x_i) \\ & a_i y_{i-1} + b_i y_i + c_i y_{i+1} = \lambda y_i \quad \text{for } i = 1, 2, \dots, N - 1 \end{aligned} \tag{P6.11.16}$$

with

$$a_i = \frac{f(x_i - h/2)}{h^2 q(x_i)}, \quad c_i = \frac{f(x_i + h/2)}{h^2 q(x_i)}, \quad \text{and} \quad b_i = \frac{r(x_i)}{q(x_i)} - a_i - c_i \tag{P6.11.17}$$

- (i) Compose a routine “sturm()” which implements the above-mentioned scheme to solve the Sturm–Liouville BVP (P6.11.15).

```
function [x,Y,ws,eigvals] = sturm(f,r,q,x0,xf,N)
```

- (ii) Use the routine “sturm()” with the number of grid points $N = 256$ to solve the following BVP2:

$$\frac{d}{dx}((1 + x^2)y') = -2\lambda y \quad \text{with } y(x_0) = 0, y(x_f) = 0 \tag{P6.11.18}$$

Plot the eigenvector solutions corresponding to the lowest three angular frequencies (ω_i 's).

