CONTENTS

MATLAB Technical Supplement

Defining and Manipulating Matrices

MATLAB works with arrays; that is, with matrices. We illustrate some of the basic matrix operations with several examples.

Let A and B denote the (3×2) matrices

$$A=\begin{bmatrix} 1 & 2 \\ -2 & 5 \\ -3 & 4 \end{bmatrix}, \qquad B=\begin{bmatrix} 0 & -1 \\ 6 & -4 \\ -7 & 8 \end{bmatrix}.$$

To enter these matrices into MATLAB, we read in the elements "row-by-row," with rows separated by semicolons.

```
» A = [ 1,2;-2,5;-3,4]

A =

      1     2
     -2     5
     -3     4
```

Individual entries can be separated by spaces instead of commas.

```
» B = [ 0 -1;6 -4;-7 8]

B =

      0    -1
      6    -4
     -7     8
```

The result of a command line statement is displayed unless it is followed by a semicolon.

A colon notation can be used to create an n-dimensional vector.

```
» x = 1:0.25:2

x =

    1.0000    1.2500    1.5000    1.7500    2.0000
```

Matrix Addition

When two matrices have the same size, the matrix sum is defined.

```
» A+B

ans =

       1       1
       4       1
     -10      12
```

Since we did not assign a name to the sum $A + B$, MATLAB gave the default name of "ans" to the sum. The next time we make a similar unnamed assignment on the command line, ans will be overwritten with the new assignment.

Scalar Multiplication

```
» M = 3*A

M =

       3       6
      -6      15
      -9      12
```

Matrix Multiplication

Let C denote the (2×3) matrix

$$C = \begin{bmatrix} 3 & 2 & 1 \\ 0 & -4 & 2 \end{bmatrix}.$$

Enter C into the MATLAB workspace,

```
» C = [ 3 2 1;0 -4 2];
```

Note that MATLAB did not echo the assignment of C because we ended the statement with a semicolon.

We can form the (3×3) matrix product AC and the (2×2) matrix product CA:

```
» A*C

ans =

      3     -6     5
     -6    -24     8
     -9    -22     5

» C*A

ans =

     -4     20
      2    -12
```

Determinants, Inverses, and Transposes

Let Q denote the (3×3) matrix

$$Q=\begin{bmatrix} 1 & 2 & 3 \\ -1 & 0 & 2 \\ 1 & -1 & -2 \end{bmatrix}.$$

Calculating the determinant of Q in MATLAB, we find

```
» det(Q)

ans =

     5
```

Since Q has nonzero determinant, Q is invertible. Let us designate Q^{-1} by $Q1$ and calculate $Q1$ with MATLAB,

```
» Q1 = inv(Q)

Q1 =

     0.4000     0.2000     0.8000
     0.0000    -1.0000    -1.0000
     0.2000     0.6000     0.4000
```

For a matrix with real entries, such as Q, the command Q' will evaluate the transpose,

```
» Q'

ans =

     1    -1     1
     2     0    -1
     3     2    -2
```

If the matrix has complex components, however, the use of the prime actually evaluates the complex conjugate transpose of the matrix. For instance, let C denote the matrix

$$C = \begin{bmatrix} 1 & 1+2i & 3i \\ 0 & 1-2i & 2 \\ -4i & 1 & 2 \end{bmatrix}.$$

```
» C = [ 1,1+2*i,3*i;0,1-2*i,2;-4*i,1,2];

C =

     1.0000            1.0000+2.0000i    0.0000+3.0000i
     0.0000            1.0000-2.0000i    2.0000
     0.0000-4.0000i    1.0000            2.0000

» C'

ans =

     1.0000            0.0000            0.0000+4.0000i
     1.0000-2.0000i    1.0000+2.0000i    1.0000
     0.0000-3.0000i    2.0000            2.0000
```

As you can see from MATLAB's answer, the operation C' calculates the conjugate transpose of C. If we want the usual transpose, we can follow the C' command with the conjugation command:

```
» conj(C' )

ans =

     1.0000            0.0000            0.0000-4.0000i
     1.0000+2.0000i    1.0000-2.0000i    1.0000
     0.0000+3.0000i    2.0000            2.0000
```

Element-by-Element Operations

A period preceding the multiplication, division, or exponentiation symbols instructs MATLAB to perform these operations on an element-by-element basis. We illustrate this idea with some examples.

As before, consider the (3×2) matrices A and B where

$$A=\begin{bmatrix} 1 & 2 \\ -2 & 5 \\ -3 & 4 \end{bmatrix}, \qquad B=\begin{bmatrix} 0 & -1 \\ 6 & -4 \\ -7 & 8 \end{bmatrix}.$$

The matrix product AB is not defined since A and B are not compatibly dimensioned. In fact, if we try to form the product AB in MATLAB, we get an error message:

```
» M = A*B

??? Error using ==> *
Inner matrix dimensions must agree.
```

We can, however, form the element-by-element product, `A.*B`; this operation creates a (3×2) array in which each element of A is multiplied by the corresponding element of B:

```
» M = A.*B

M =

      0     -2
    -12    -20
     21     32
```

Similarly, the MATLAB command `A./B` command creates a (3×2) array whose elements are the elements of A divided by the corresponding elements of B. For our matrices A and B, the MATLAB warning and the resulting `Inf` in the $(1,1)$ position occurred because we divided by zero.

```
» A./B
Warning: Divide by zero.

ans =

       Inf    -2.0000
   -0.3333    -1.2500
    0.4286     0.5000
```

The command `B.^2` creates a (3×2) array whose elements are the squares of the elements of B:

```
» BSQ = B.^2

BSQ =

     0     1
    36    16
    49    64
```

MATLAB has one matrix operation that does not have a mathematical counterpart—the operation $A \pm a$ where a is a scalar and A is an $(m\times n)$ matrix. MATLAB "understands" the commands $A + a$ (or $A - a$) to mean "add (or subtract) the scalar a from each element of the matrix A." For example,

```
» A = [ 1  2  5  7];

» B = A + 5

B = 6    7    10    12
```

Built-in Functions

When a matrix is used as an input to a built-in function, such as $\exp(x)$ or $\sin(x)$, MATLAB "understands" that the operation is to be performed element-wise. For example, if we define the (1×4) array $x = [-1 \quad 2 \quad 0.5 \quad 1]$, then e^x is a (1×4) array as well:

```
» x = [ -1  2  0.5  1];

» y = exp(x)

y = 0.3679    7.3891    1.6487    2.7183
```

The output is the (1×4) array $y = [e^{-1} \quad e^{2} \quad e^{0.5} \quad e^{1}]$. In general, if $f(x)$ denotes one of MATLAB'S built-in functions and if x is an $(m\times n)$ matrix, then $f(x)$ is also an $(m\times n)$ matrix whose ijth entry is $f(x_{ij})$.

Defining and Graphing Functions

When we want to graph a function on some interval of interest, we start by defining an array of input values for the function and then creating the corresponding array of output values. The plot command will graph the points associated with these two arrays.

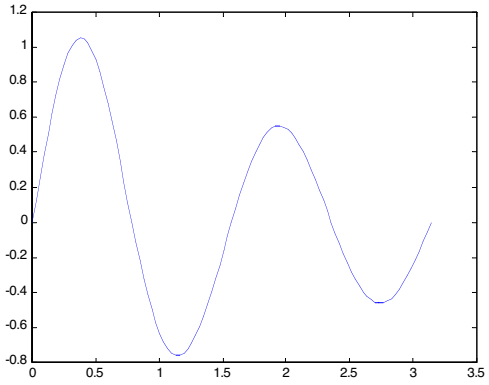
For instance, suppose we want to graph the function

$$y(t) = \frac{e^{t/2}\sin 4t}{1+t^2}$$

on the interval $0 \le t \le \pi$.

```
» t = 0:pi/100:pi;

» y = exp(t/2).*sin(4*t)./(1+t.^2);

» plot(t,y)
```

The resulting graph is shown below.



Superimposing and Labeling Graphs

Suppose we want to graph two functions, $f(t) = 2\sin t^2$ and its derivative $f'(t) = 4t\cos t^2$, on the interval $0 \le t \le 4$. We can proceed as follows:

```
» t = 0:0.01:4;

» f = 2*sin(t.^2);

» fp = 4*t.*cos(t.^2);
```

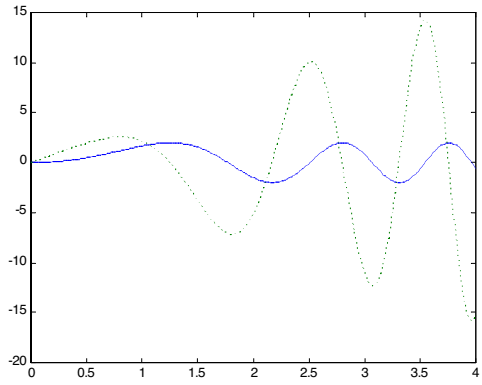
The plot command allows for several graphs to be superimposed. Also, as illustrated, we can specify different line types for different graphs. The "`help plot`" command will give you a list of the available line types.

```
» plot(t,f,t,fp,':')
```

As illustrated below, labels can be added to the graph.

```
» xlabel('t')

» ylabel('f(t) and its derivative')

» title('graphs of f(t), solid curve, and its derivative,
dashed curve')
```

The resulting graph is shown below.



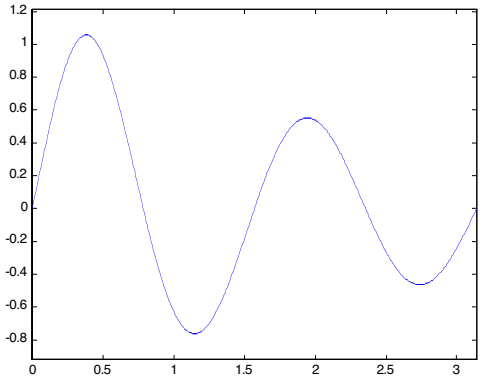
Graphing With the Symbolic Toolbox

We can also use the capabilities of the MATLAB Symbolic Toolbox to graph functions. The sequence of commands listed below displays the graph of the function $f(t)$ for $0 \le t \le \pi$, where

$$f(t) = \frac{e^{t/2}\sin 4t}{1+t^2}.$$

```
» syms t

» f = exp(t/2)*sin(4*t)/(1+t^2);

» ezplot(f,[0,pi])
```

The first command above identifies t as a symbolic variable. The second command defines the function f, and the ezplot command graphs the function. The result is shown below.



Matrix Eigenpair Computations

Since MATLAB is a matrix-oriented program, it is an especially efficient tool for eigenvalue calculations. For example, suppose we want to compute the eigenvalues of the matrix A

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

Using MATLAB, we find the eigenvalues of A:

```
» A = [ 1  2 -1;2  0 -3;-1 -3 -2];

» eig(A)

ans =

   -0.6782
   -4.1913
    3.8695
```

We can find eigenvectors as well as the eigenvalues with the command

```
» [ V,D]  = eig(A)

V =

   -0.7870    -0.0807    -0.6116
    0.4488     0.6053    -0.6574
   -0.4232     0.7919     0.4402


D =

   -0.6782          0          0
         0    -4.1913          0
         0          0     3.8695
```

The command `[ V,D]  = eig(A)` generates eigenvectors as well as eigenvalues. The eigenvectors appear as the columns of the matrix V. The corresponding eigenvalues are the diagonal entries of D. In particular, the ith column of V is an eigenvector associated with the (i,i) entry of D. As a result, the matrix equality

$$AV = VD$$

is valid.

If the columns of V are linearly independent (that is, if A has a full set of eigenvectors), then V is invertible and A is diagonalizable; therefore,

$$V^{-1}AV = D \quad \text{or} \quad A = VDV^{-1}.$$

For the (3×3) example above, we illustrate how the eigenpair calculation can be verified for the second column of V:

```
» V2 = V(:,2)

V2 =

    -0.0807
     0.6053
     0.7919

» lambda2 = D(2,2)

lambda2 =

   -4.1913

» A*V2

ans =

    0.3381
   -2.5370
   -3.3191

» lambda2*V2

ans =

    0.3381
   -2.5370
   -3.3191
```

As is seen from the calculation above, $A\mathbf{v}_2 = \lambda_2\mathbf{v}_2$, where $\mathbf{v}_2$ denotes the second column of V and λ_2 denotes the $(2,2)$ entry of D.

Recall that eigenvectors are not unique. Any constant nonzero multiple of an eigenvector is also an eigenvector. MATLAB chooses to normalize the eigenvectors so that they are unit vectors; that is, the sum of the squares of the components of an eigenvector is equal to one.

Not all matrices have a full set of eigenvectors. For instance, consider the matrix J

$$J = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}.$$

This matrix has a real repeated eigenvalue $\lambda_1 = \lambda_2 = 2$ but only one linearly independent eigenvector.

If we use MATLAB to compute the eigenpairs, we obtain

```
» J = [ 2 1;0 2];

» [ V,D] = eig(J)

V =

   1.0000   -1.0000
   0.0000    0.0000

D =

   2.0000    0.0000
   0.0000    2.0000
```

Note that the columns of V consist of the two vectors

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

In this case, MATLAB presents two scalar multiples of the same eigenvector. This example illustrates the need to scrutinize the output of software packages.

A real matrix that has complex eigenvalues also has complex eigenvectors. Consider

$$B = \begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}.$$

Using MATLAB,

```
» B = [ 1 2;-2 1];

» [ V,D] = eig(B)

V =

   0.0000 - 0.7071i    0.0000 + 0.7071i
   0.7071              0.7071

D =

   1.0000 + 2.0000i    0.0000
   0.0000              1.0000 - 2.0000i
```

Note that the eigenvalues of the (2×2) real matrix B form a complex conjugate pair and the corresponding eigenvectors are likewise complex conjugates of each other.

The Solution of Initial Value Problems

MATLAB provides four different approaches for solving initial value problems.

(a) A symbolic solution can be generated using the dsolve command within the Symbolic Toolbox.
(b) Solutions can be generated for problems involving linear homogeneous
constant coefficient problems by using the exponential matrix function, expm.
(c) MATLAB provides a programming environment that allows us to create
and execute numerical methods such as Euler's method.
(d) MATLAB has built-in numerical routines such as ODE45 that can be called
to solve initial value problems.

Below, we give examples of each of these approaches. In most instances, a practitioner will want to use a tested and refined program such as ODE45. There are cases, however, where using an existing program is not an option and we need to write our own program.

Symbolic Solutions Using dsolve

The dsolve command in the Symbolic Toolbox can be used to obtain symbolic solutions of differential equations and initial value problems. For instance, in order to determine the general solution of $y' + y = \sin t$, we can proceed as follows:

```
» dsolve('Dy + y = sin(t)')

ans = -1/2*cos(t)+1/2*sin(t)+exp(-t)*C1
```

The symbol *D* denotes derivative. Note that it was not necessary to declare *t* to be a symbolic variable. As a second example, consider the initial value problem

$$y'' + 4y' + 4y = 2 + t\,,\ \ y(0) = 1, y'(0) = 0\,.$$

We can solve this initial value problem using the dsolve command:

```
» y = dsolve('D2y+4*Dy+4*y = 2+t','y(0)=1,Dy(0)=0')

y = 1/4+1/4*t+3/4*exp(-2*t)+5/4*exp(-2*t)*t
```

Note that the symbol *Dk* denotes the *k*th derivative. Also note that the initial conditions are included in the dsolve function call.

Computer software packages such as dsolve have no sense of aesthetics. The answers they provide, while correct, are sometimes useless or confusing. For example, consider the relatively simple differential equation

$$y'' + 2y' + 4y = \sin t.$$

The general solution is

$$y = c_1 e^{-t}\cos\sqrt{3}\,t + c_2 e^{-t}\sin\sqrt{3}\,t + (3\sin t - 2\cos t)/13.$$

A call to dsolve, however, produces the following rather complicated form of the solution:

```
-3/26*sin(3^(1/2)*t)*cos((3^(1/2)+1)*t)+1/78*sin(3^(1/2)*t)*cos((3^(1/2)+1)*t)*3^(1/2)
+5/78*sin(3^(1/2)*t)*sin((3^(1/2)+1)*t)*3^(1/2)-
1/13*sin(3^(1/2)*t)*sin((3^(1/2)+1)*t)+3/26*sin(3^(1/2)*t)*cos((3^(1/2)-
1)*t)+1/78*sin(3^(1/2)*t)*cos((3^(1/2)-1)*t)*3^(1/2)-5/78*sin(3^(1/2)*t)*sin((3^(1/2)-1)*t)*3^(1/2)-
1/13*sin(3^(1/2)*t)*sin((3^(1/2)-1)*t)-5/78*cos(3^(1/2)*t)*cos((3^(1/2)-1)*t)*3^(1/2)-
1/13*cos(3^(1/2)*t)*cos((3^(1/2)-1)*t)-3/26*cos(3^(1/2)*t)*sin((3^(1/2)-1)*t)-
1/78*cos(3^(1/2)*t)*sin((3^(1/2)-1)*t)*3^(1/2)+5/78*cos(3^(1/2)*t)*cos((3^(1/2)+1)*t)*3^(1/2)-
1/13*cos(3^(1/2)*t)*cos((3^(1/2)+1)*t)+3/26*cos(3^(1/2)*t)*sin((3^(1/2)+1)*t)-
1/78*cos(3^(1/2)*t)*sin((3^(1/2)+1)*t)*3^(1/2)+C1*exp(-t)*sin(3^(1/2)*t)+C2*exp(-
t)*cos(3^(1/2)*t)
```

Homogeneous Linear Constant Coefficient Systems and the Exponential Matrix

Let A be a constant $(n \times n)$ matrix. In this subsection, we examine how MATLAB can be used to solve $\mathbf{y}' = A\mathbf{y}$ by computing the exponential matrix. For instance, consider the initial value problem

$$\frac{d}{dt}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} -1 & 3 \\ -3 & -1 \end{bmatrix}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}, \quad \begin{bmatrix} y_1(0) \\ y_2(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}.$$

Our objective is to plot the two solution components on the interval $0 \le t \le 4$.

Recall that the exponential matrix e^{tA} is the fundamental matrix that reduces to the identity matrix at $t = 0$. It serves to "propagate" the solution. In particular, we have

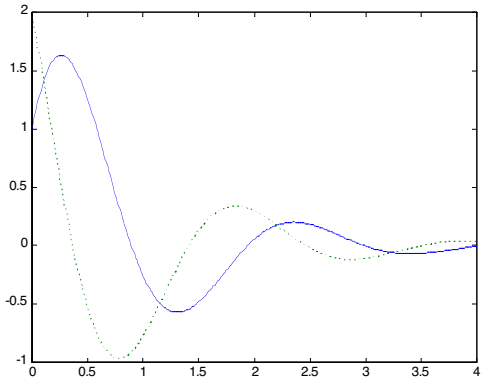
$$\begin{bmatrix} y_1(t+\Delta t) \\ y_2(t+\Delta t) \end{bmatrix} = e^{\Delta t A}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}. \qquad (1)$$

The MATLAB command `expm(A)` forms the exponential matrix e^A.

The following sequence of MATLAB commands generates graphs of the solution components. The sequence of commands begins by defining a grid on the interval $0 \le t \le 4$, using a grid spacing of $\Delta t = 0.01$. We then apply the propagation property (1), using it to evaluate the solution components $y_1(t)$ and $y_2(t)$ at the grid points.

```
» delt = 0.01;

» t = 0:delt:4;

» N = length(t);

» A = [ -1  3;-3  -1];

» C = expm(delt*A)

C =

    0.9896    0.0297
   -0.0297    0.9896

» y = zeros(2,N);

» y(:,1) = [ 1  2]';

» for j = 2:N

     y(:,j) = C*y(:,j-1);

  end

» plot(t,y(1,:),t,y(2,:),':')

» title('Solution components, y1 (solid curve),
         y2 (dashed curve)')

» xlabel('t')

» ylabel('solution components')
```

The resulting graph is shown below.



Comments:

1. The second statement in the sequence of MATLAB commands defines an equally spaced grid, t, consisting of 401 points. The third statement defines N, the length of the vectors made from the solution components, $y_1(t_i)$ and $y_2(t_i)$, $1 \le i \le N$.
2. The sixth statement, `"y = zeros(2,N)"`, initializes an array consisting of two rows and N columns. The first row will hold the vector $y_1(t_i), i = 1, 2, \ldots, N$ and the second row will hold the vector $y_2(t_i), i = 1, 2, \ldots, N$.
3. The seventh statement, `y(:,1) = [ 1 2]'`, stores the initial condition vector in the first column of the solution array; the colon stands for "all rows." [Similarly, a statement such as `B(3,:) = [ 1 2]` would mean that the row vector `[ 1 2]` is stored in the third row of B; in this case, the colon would stand for "all columns."]
4. The "for loop" following statement seven uses (1) to propagate the solution from one grid point to the next. As the index j takes on the values 2, 3, 4, . . . , N, the columns of the solution array are generated.

Note that this computational approach also can be applied to the solution of initial value problems involving nth order linear homogeneous constant coefficient scalar equations. For such problems, we would rewrite the equation as an equivalent first order system.

Programming in MATLAB; Scripts and Functions

When we have a complicated computational task to perform, we may want to program the calculations. While the steps of a program could be entered on the command line one by one, it is normally more efficient to store the commands in a file and then execute the program by invoking the file. The previous example, solving an initial value problem using the exponential matrix, is a case in point. It would have been more efficient to type the sequence of fourteen steps into a single file rather than entering them one by one on the command line.

MATLAB program files are called m-files. They are typically created in the MATLAB editor and saved as files with an extension denoted .m. There are two types of m-files: script m-files and function m-files.

To illustrate the concept of a script m-file, suppose we type the fourteen steps of the exponential matrix example in the MATLAB editor and save the resulting file as example.m. If we now type the word "example" in the command line, the file example.m will execute and the plot shown above will appear. Thus, a script m-file is nothing more than a sequence of MATLAB statements entered and saved in a file. Among the many advantages of stored programs (such as scripts) are that they can be invoked over and over with a single statement, they can be modified easily to handle different circumstances, they can be executed a few lines at a time to debug the program, etc.

The second type of m-file is a function m-file. As the name suggests, a function m-file has many of the same properties as a mathematical function; a function m-file has a set of inputs and a set of rules that transforms the input variables into a set of output variables. The output variables are then returned to

the user who invoked the function. In MATLAB, the first line of a function m-file must begin with the keyword "function." The following simple example illustrates the syntax.

```
function c = triple(u,v,w)
x = cross(v,w);
c = dot(u,x);
```

The simple function above calculates the triple product of three vectors, $\mathbf{c} = \mathbf{u} \bullet (\mathbf{v} \times \mathbf{w})$, using the built-in MATLAB dot product and cross product functions. Once we save the function file as triple.m, we can call it from the command line and use it to form the triple product of three vectors. For example,

```
» a = [ 1  2  4]; b = [ 2  1  3]; c = [ 7  0  -3];

» trip = triple(a,b,c)

» trip = 23
```

In a function m-file, all the variables are local. In a script m-file, however, the variables are global. For that reason, we need to be careful in using scripts.

Solving Initial Value Problems Using Numerical Methods

Euler's method as well as the numerical methods described in Chapter 9 can be coded and run in MATLAB. As an example, we list an implementation of Euler's method in the form of a function m-file. We then use the program to solve three different initial value problems.

Our implementation is designed to use Euler's method to numerically solve initial value problems of the form

$$y' = f(t,y),\ y(a) = y_0,\ a \le t \le b. \qquad (2)$$

The program is flexible enough so that it can be used for first-order systems as well as for scalar problems. The program assumes that n steps of Euler's method will be executed, using a constant step size $h = (b-a)/n$. The program calls another function m-file, named f.m, that evaluates $f(t,y)$. In order to use the Euler's method program below, the function f.m needs to be available.

```
function results = euler(a,b,y0,n)
%
% Set the starting values for Euler's method
%
t = a;
h = (b-a)/n;
y = y0;
results = [ t,y];
%
% Execute n steps of Euler's method
%
for i = 1:n
   y = y + h*f(t,y);
   t = t + h;
   results = [ results;[ t,y]];
end
```

Comments on the program:

1. The fifth statement, `results = [ t,y]` , initializes an array that will hold the results. For a scalar problem, the first row of this results array is $[a, y_0]$. As additional points $[t_i, y_i]$ are generated, they will be appended to the array.
2. The for loop generates the iterates of Euler's method. The statement `results = [ results;[ t,y]]` , appends the latest iterate to the array. For a scalar problem, the results array will ultimately be an $((n+1)\times 2)$ matrix.
3. As can be seen from the first line of the program, the input variables for the function euler.m are the interval $[a,b]$, the initial value y_0, and the number of steps to be executed, n. The output is the array named results.

We illustrate euler.m by using it to estimate the solution of

$$y' = -ty^3 + 1,\ y(0) = 1,\ 0 \le t \le 2.$$

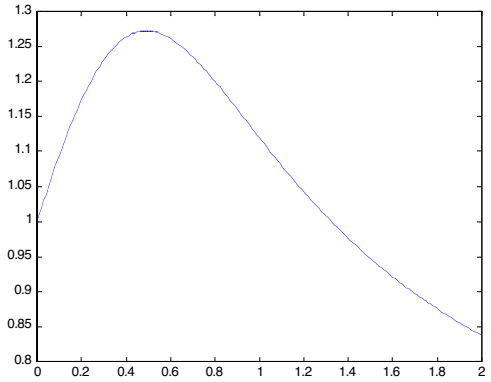
For this example, the following function m-file, f.m, must be prescribed:

```
function yp = f(t,y)
yp = -t*y^3+1;
```

For this illustration, we use $n = 200$ leading to a step size of $h = 0.01$. After calling euler.m, we graph the numerical solution.

```
» a = 0;

» b = 2;

» y0 = 1;

» n = 200;

» results = euler(a,b,y0,n);

» t = results(:,1);

» y = results(:,2);

» plot(t,y)

» xlabel('t')

» ylabel('y')
```

The resulting graph is shown below.



In the next example, we use the function euler.m to solve the nonlinear second order initial value problem

$$y'' + \sin y = e^{-t}\sin t\,,\ \ y(0) = 0, y'(0) = 2, 0 \le t \le 4\,.$$

The first step in solving this problem is to convert it into a first order system,

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -\sin y_1 + e^{-t}\sin t \\ y_1(0) &= 0, y_2(0) = 2\ \ . \end{aligned} \tag{3}$$

For the system $\mathbf{y}' = \mathbf{f}(t,\mathbf{y})$, Euler's method reduces to the vector iteration

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{f}(t_i,\mathbf{y}_i)\,.$$

As can be seen from the listing of the function euler.m, this particular program can be used for first order systems as well as for scalar equations. The only thing we need to do is code the function f.m so that $\mathbf{y}' = \mathbf{f}(t,\mathbf{y})$ is a row vector. Below, we list the function f.m for system (3).

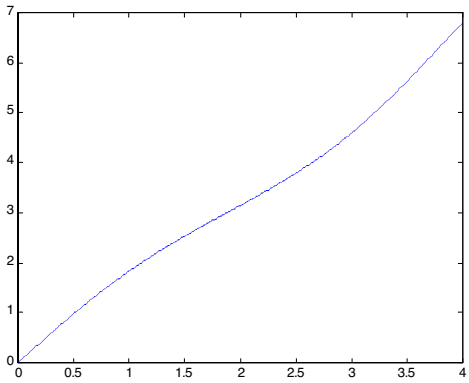
```
function yp = f(t,y)
yp = zeros(1,2);
yp(1) = y(2);
yp(2) = -sin(y(1)) + exp(-t)*sin(t);
```

Even though this new problem is a first order system, we can call the function euler.m just as we did in our first example. Of course, a, b, y_0, and n are different. In particular, y_0 is a row vector.

```
» a = 0;

» b = 4;

» y0 = [ 0,1];

» n = 400;

» results = euler(a,b,y0,n);

» t = results(:,1);

» y = results(:,2);
```

For this example, the output array "results" is a (401×3) matrix. The first column holds the 401 grid points $t_0, t_1, \ldots, t_{400}$, the second column holds the numerical solution values $y_0, y_1, \ldots, y_{400}$, and the third column has the numerical derivative values $y_0', y_1', \ldots, y_{400}'$.

The graph of the numerical solution of system (3) is shown in the figure:



The function euler.m is easily modified so as to implement other one-step methods such as Heun's method or the modified Euler's method. For example, the following listing is an implementation of Heun's method:

```
function results = heun(a,b,y0,n)
%
% Set the starting values for Heun's method
%
t = a;
h = (b-a)/n;
y = y0;
results = [t,y];
%
% Execute n steps of Heun's method
%
for i = 1:n
   y = y + (h/2)*(f(t,y) + f(t+h;y+h*f(t,y)));
   t = t + h;
   results = [results;[t,y]];
end
```

As can be seen from this listing, euler.m and heun.m are identical except for the first line in the for loop; this line reflects the difference between the Euler's method iteration and the Heun's method iteration.

As an example, we use heun.m to solve the system

$$\begin{aligned} x' &= -(1-y)x + e^{-t} \\ y' &= -(1+x)y + 1 \\ x(0) &= 1,\ y(0) = 2,\ 0 \le t \le 3\,. \end{aligned} \tag{4}$$

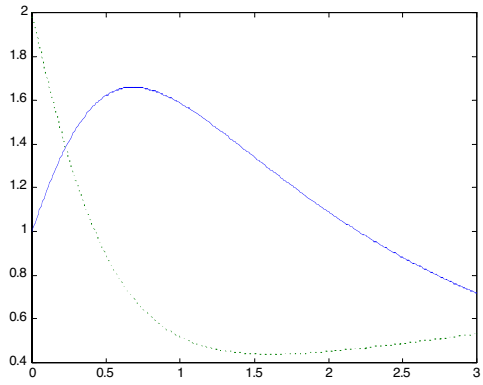
The function f.m is listed below, followed by the MATLAB statements invoking heun.m for the initial value problem (4).

```
function yp = f(t,y)
yp = zeros(1,2);
yp(1) = -(1 - y(2))*y(1) + exp(-t);
yp(2) = -(1 + y(1))*y(2) + 1;
```

The MATLAB commands calling heun.m are:

```
» a = 0;

» b = 3;

» y0 = [1,2];

» n = 300;

» results = heun(a,b,y0,n);

» t = results(:,1);

» x = results(:,2);

» y = results(:,3);
```

A graph of the solutions of system (4) is shown below.



Solving Initial Value Problems Using MATLAB's Built-in Numerical Methods

MATLAB has several built in numerical routines for initial value problems. An example is the program ODE45. The program ODE45 is a variable-step Runge-Kutta method, one that is fairly easy to use and quite reliable. We illustrate ODE45 by using it to solve numerically the initial value problem (4) that was featured in the previous example. This time, however, we solve the problem over the interval [0,10].

The call to ODE45 is

```
» [t,y] = ode45('f',tspan,y0).
```
(5)

In (5), we are trying to solve the system $\mathbf{y}' = \mathbf{f}(t,\mathbf{y}), \mathbf{y}(t_0) = \mathbf{y}_0, a \le t \le b$. In (5), `f` is the name of a function m-file that evaluates $\mathbf{f}(t,\mathbf{y})$, `tspan` is a row vector, and `y0` denotes the vector $\mathbf{y}_0$ of initial conditions. The output consists of times $t_0, t_1, \ldots, t_N$, and the numerical solution vectors $\mathbf{y}(t_0), \mathbf{y}(t_1), \ldots, \mathbf{y}(t_N)$. If `tspan` is the vector $[a,b]$, then the output values $t_0, t_1, \ldots, t_N$ are the points used by the variable-step method. If `tspan` has points between a and b and if ODE45 is able to solve the initial value problem successfully, then the output times are those specified by the input vector `tspan`. Unlike our implementation of Euler's method and Heun's method, the function m-file `f` must return a column vector.

In order to illustrate the use of ODE45 we apply it to solve the initial value problem (4). In this case, the function m-file has the form

```
function yp = f(t,y)
yp = zeros(2,1);
yp(1) = -(1 - y(2))*y(1) + exp(-t);
yp(2) = -(1 + y(1))*y(2) + 1;
```

As you can see, the only difference between f.m in these two examples is that the initial specification defines the output vector `yp` to be a row vector in the previous example, but a column vector in this example.

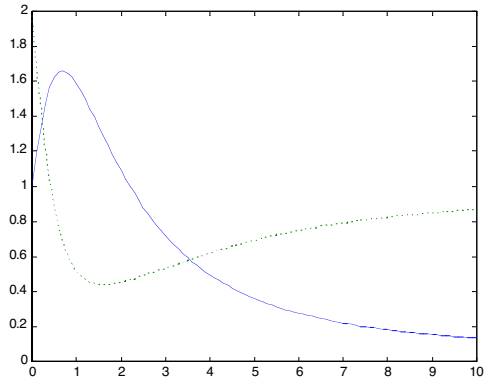
Unlike our implementation of Euler's method and Heun's method, we are not restricted to naming the function file f.m; the call to ODE45 passes the name of the function file as a string. For instance, instead of naming the function f.m as we did above, we could have called it fex4.m:

```
function yp = fex4(t,y)
yp = zeros(2,1);
yp(1) = -(1 - y(2))*y(1) + exp(-t);
yp(2) = -(1 + y(1))*y(2) + 1;
```

After saving this m-file as fex4.m, we move to the MATLAB command line and define the input variables `tspan` and `y0`. We then call ODE45 and graph the results.

```
» tspan = 0:0.1:10;

» y0 = [1;2];

» [t,y] = ode45('fex4',tspan,y0);

» x = y(:,1);

» y = y(:,2);

» plot(t,x,t,y,':')
```

The resulting graph is shown below. We generated a numerical solution of system (4) earlier, using Heun's method. This time, we graph the numerical solution over the longer interval $0 \le t \le 10$.



If we wish to do so, we can create a table of values for $x(t)$ and $y(t)$:

```
» tabval = [ t,x,y]

tabval =

    0.0000    1.0000    2.0000
    0.1000    1.1883    1.7119
    0.2000    1.3475    1.4537
    0.3000    1.4731    1.2307
    0.4000    1.5644    1.0444
                 .
                 .
                 .

    9.8000    0.1371    0.8636
    9.9000    0.1352    0.8654
   10.0000    0.1334    0.8671
```

The routine ODE45 has many options that ease the job of the user; consult the help file for ODE45. In addition to ODE45, MATLAB has a number of other numerical routines for estimating the solution of an initial value problem. Some of these routines have options for special situations such as dense output, stiff equations, differential-algebraic systems, etc. These routines include ODE113, ODE15S, ODE23S, ODE23T, and ODE23TB.

Flowfields for First Order Scalar Equations

We list below a MATLAB program, flowfield.m, that can be used to construct a flowfield for the first order scalar equation, $y' = f(t,y)$. We illustrate the use of this program by constructing a flowfield for the equation $y' = y(2-y)$ on the rectangle $0 \le t \le 8, -3 \le y \le 4$.

The following two m-files need to be entered into MATLAB. The m-file f.m defines the right side of the particular differential equation being studied. The m-file flowfield.m calls f.m and displays the graph of the flowfield.

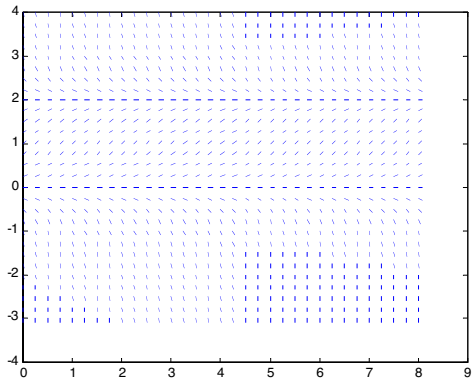
```
function yp = f(t,y)
yp = y*(2-y);
```

```
function flowfield(a,b,c,d,tinc,yinc,scafac)
tval = a:tinc:b;
yval = c:yinc:d;
nt = length(tval);
ny = length(yval);
t = zeros(ny,nt);
y = zeros(ny,nt);
u = zeros(ny,nt);
v = zeros(ny,nt);

for i = 1:ny
   for j = 1:nt
      dy = f(tval(j),yval(i));
      t(i,j) = tval(j);
      y(i,j) = yval(i);
      u(i,j) = 1/sqrt(1+dy*dy);
      v(i,j) = dy/sqrt(1+dy*dy);
   end
end
%
% The quiver command plots the flow field.
%
quiver(t,y,u,v,scafac,'.')
xlabel('t')
ylabel('y')
```

Once these two files are entered into MATLAB, the single command

```
» flowfield(0,8,-3,4,0.25,0.25,0.3)
```

produces the following output:



The call to flowfield has the form

```
» flowfield(a,b,c,d,tinc,yinc,scafac)
```

The program draws the flowfield over the rectangle $a \le t \le b, c \le y \le d$. The flowfield filaments are placed *tinc* units apart in the *t* direction and *yinc* units apart in the *y* direction. The parameter *scafac* determines the length of the filaments. The MATLAB quiver command plots the filaments; the last parameter in the quiver function, `'.'`, suppresses the placement of an arrowhead on the filaments. Eliminating this input causes the filaments to become small arrows.

Direction Fields for Two-Dimensional Autonomous Systems

Below, we list a MATLAB program, dirfield.m, that constructs the direction field for a two-dimensional autonomous system of the form

$$x' = f(x,y)$$
$$y' = g(x,y)\,.$$

We illustrate the program by constructing the direction field for the system

$$x' = (2 - x - y)x/4$$
$$y' = (3 - x - 2y)y/12 \qquad (6)$$

on the square $0 \le x \le 3, 0 \le y \le 3$.

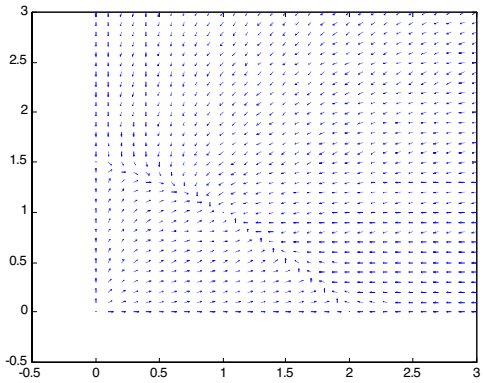
The following two m-files need to be entered into MATLAB. The m-file fdir.m defines the right side of the particular system being studied. The m-file dirfield.m calls fdir.m and displays the graph of the direction field.

```
function [ xp,yp]  = fdir(x,y)
xp = (2-x-y)*x/4;
yp = (3-x-2*y)*y/12;


function dirfield(a,b,c,d,xinc,yinc,scafac)
xval=a:xinc:b;
yval=c:yinc:d;
nx=length(xval);
ny=length(yval);
x=zeros(ny,nx);
y=zeros(ny,nx);
xp=zeros(ny,nx);
yp=zeros(ny,nx);

for i=1:ny
   for j=1:nx
%
% dx = f(x,y) and dy = g(x,y). A function m-file, named
%  fdir.m, must be available to calculate dx and dy.
%
         [ dx,dy] =fdir(xval(j),yval(i));
         x(i,j)=xval(j);
         y(i,j)=yval(i);
         mag=sqrt(dx*dx+dy*dy);
         xp(i,j)=dx;
         yp(i,j)=dy;

         if mag > 0
            xp(i,j)=xp(i,j)/mag;
            yp(i,j)=yp(i,j)/mag;
         end

   end
end
%
% The quiver command plots the direction field
%
quiver(x,y,xp,yp,scafac)
xlabel('x')
ylabel('y')
```

Once these two files are entered into MATLAB, the single command

```
» dirfield(0,3,0,3,0.1,0.1,0.3)
```

produces the following direction field for system (6):



The call to dirfield has the form

```
» dirfield(a,b,c,d,xinc,yinc,scafac)
```

The program draws the direction field over the rectangle $a \le x \le b, c \le y \le d$. The filaments are placed *xinc* units apart in the *x* direction and *yinc* units apart in the *y* direction. The parameter *scafac* determines the length of the filaments. The MATLAB quiver command plots the filaments.

Laplace Transforms

We present several examples showing how the Symbolic Toolbox can be used to compute Laplace transforms and their inverses. In the following examples, the variables *s* and *t*, as well as the parameter *w*, are first declared to be symbolic variables.

```
» syms s t w;

» laplace(sin(w*t))

ans = w/(s^2+w^2)

» F = laplace(exp(-2*t))

F = 1/(s+2)
```

When a function of *t* is transformed, the default variable for the transform is *s*. Similarly, when the inverse transform operation, `ilaplace`, is applied to a function of *s*, the default variable for the inverse transform is *t*.

```
» ilaplace(2/(s^2+4))

ans = 1/2*4^(1/2)*sin(4^(1/2)*t)

» simplify(ans)

ans = sin(2*t)
```

Two-Point Boundary Value Problems

Just as we saw with initial value problems, two-point boundary value problems can be solved using the Symbolic Toolbox, the exponential matrix, or numerical methods.

Solutions using the Symbolic Toolbox: We begin by illustrating a symbolic solution of the two-point boundary value problem

$$y'' - y = 0,\ \ y(0) = 1, y'(1) = 0\,.$$

Using the dsolve command, we obtain

```
» dsolve('D2y-y=0','y(0)=1,Dy(1)=0')

ans = -sinh(1)/cosh(1)*sinh(t)+cosh(t).
```

Solutions using the exponential matrix: The exponential matrix command can be used to solve two-point boundary value problems that involve linear constant coefficient first order systems. For example, consider

$$y^{(4)} - y = 0,\ \ y(0) = 1, y'(0) = 2, y(\pi) = 1, y'(\pi) = -2\,.$$

We first recast this problem as a first order system by defining the change of dependent variables $y = z_1, y' = z_2, y'' = z_3, y''' = z_4$. With this, the fourth order scalar problem becomes $\mathbf{z}' = A\mathbf{z},\quad P_1\mathbf{z}(0) + P_2\mathbf{z}(\pi) = \boldsymbol{\alpha}$, where

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix},\quad P_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},\quad P_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix},\quad \boldsymbol{\alpha} = \begin{bmatrix} 1 \\ 2 \\ 1 \\ -2 \end{bmatrix}.$$

The general solution of the differential equation is

$$\mathbf{z} = e^{tA}\mathbf{c} \qquad (7)$$

where $\mathbf{c}$ is a (4×1) vector of arbitrary coefficients. The vector $\mathbf{c}$ is determined by imposing the boundary condition

$$(P_1 + P_2 e^{\pi A})\mathbf{c} = \boldsymbol{\alpha}\,.$$

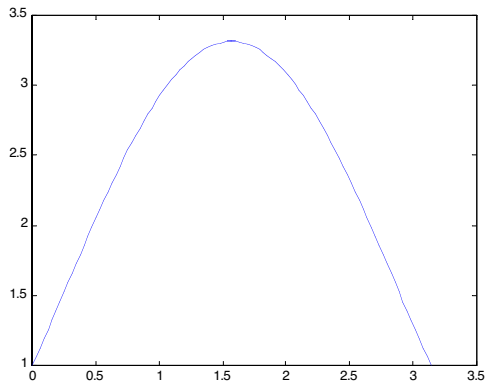
Once we have determined $\mathbf{c}$, we also know $\mathbf{z}(0)$ since, see (7), $\mathbf{z}(0) = \mathbf{c}$. We now use the propagator property of the exponential matrix, $\mathbf{z}(t + \Delta t) = e^{\Delta t A}\mathbf{z}(t)$, to determine the solution at grid points spanning the interval $0 \le t \le \pi$. The following MATLAB script carries out these steps and graphs the solution, $y(t) = z_1(t)$.

```
%
% The integer N defines the grid spacing and needs to be
% defined before invoking this script, bvpexample.m
%
dt = pi/N;
t = 0:dt:pi;
z = zeros(4,N+1);
A = [ 0 1 0 0;0 0 1 0;0 0 0 1;1 0 0 0];
P1 = [ 1 0 0 0;0 1 0 0;0 0 0 0;0 0 0 0];
P2 = [ 0 0 0 0;0 0 0 0;1 0 0 0;0 1 0 0];
alpha = [ 1 2 1 -2] ';
expdt = expm(dt*A);
c = inv(P1+P2*expm(pi*A))*alpha;
z(:,1) = c;

for i = 1:N
   z(:,i+1) = expdt*z(:,i);
end

plot(t,z(1,:))
xlabel('t')
ylabel('y')
```

To invoke the script bvpexample.m, we define *N*

```
» N = 100;

» bvpexample
```

The resulting graph is shown below.



Solutions of linear two-point boundary value problems using numerical methods:

For two-point boundary value problems involving linear differential equations and separated boundary conditions, the principle of superposition applies; the general solution of a nonhomogeneous differential equation can be represented as the sum of the general solution of the homogeneous equation and a particular solution. Numerical methods can be used to construct these constituents. Then, imposing the boundary conditions, we obtain the solution of the two-point boundary value problem.

As an illustration, consider the boundary value problem

$$y'' - t^2 y = e^{-t}, \quad y(0) + y'(0) = 2, y(2) - 2y'(2) = 0. \tag{8}$$

We can rewrite this problem as

$$\mathbf{z}' = A\mathbf{z} + \mathbf{g}(t), \quad P_1\mathbf{z}(0) + P_2\mathbf{z}(2) = \boldsymbol{\alpha}$$

where

$$A = \begin{bmatrix} 0 & 1 \\ t^2 & 0 \end{bmatrix}, \quad \mathbf{g}(t) = \begin{bmatrix} 0 \\ e^{-t} \end{bmatrix}, \quad P_1 = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}, \quad P_2 = \begin{bmatrix} 0 & 0 \\ 1 & -2 \end{bmatrix}, \quad \boldsymbol{\alpha} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

The general solution of the nonhomogeneous system has the form

$$\mathbf{z}(t) = \Psi(t)\mathbf{c} + \mathbf{z}_P(t)$$

where $\Psi(t)$ is any fundamental matrix, $\mathbf{c}$ is a (2×1) vector of arbitrary constants, and $\mathbf{z}_P(t)$ is any particular solution of the nonhomogeneous system.

In particular, let $\Phi(t)$ denote the fundamental matrix that reduces to the identity at $t = 0$,

$$\Phi(t) = \begin{bmatrix} \phi_{11}(t) & \phi_{12}(t) \\ \phi_{21}(t) & \phi_{22}(t) \end{bmatrix}.$$

Likewise, let $\mathbf{z}_P(t)$ denote the particular solution of the nonhomogeneous equation that satisfies the initial condition $\mathbf{z}_P(0) = \mathbf{0}$.

We construct $\Phi(t)$ and $\mathbf{z}_P(t)$ numerically, using ODE45 to solve the following initial value problems

$$\frac{d}{dt}\begin{bmatrix} \phi_{11}(t) \\ \phi_{21}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ t^2 & 0 \end{bmatrix}\begin{bmatrix} \phi_{11}(t) \\ \phi_{21}(t) \end{bmatrix}, \begin{bmatrix} \phi_{11}(0) \\ \phi_{21}(0) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{9a}$$

$$\frac{d}{dt}\begin{bmatrix} \phi_{12}(t) \\ \phi_{22}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ t^2 & 0 \end{bmatrix}\begin{bmatrix} \phi_{12}(t) \\ \phi_{22}(t) \end{bmatrix}, \begin{bmatrix} \phi_{12}(0) \\ \phi_{22}(0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{9b}$$

$$\mathbf{z}' = A\mathbf{z} + \mathbf{g}(t), \quad \mathbf{z}(0) = \mathbf{0}. \tag{9c}$$

Finally, imposing the boundary condition on the general solution $\mathbf{z}(t) = \Phi(t)\mathbf{c} + \mathbf{z}_P(t)$, determines the vector **c** as the solution of

$$[P_1 + P_2\Phi(2)]\mathbf{c} = \boldsymbol{\alpha} - P_2\mathbf{z}_P(2). \qquad (9d)$$

Having **c** we can evaluate the solution

$$\mathbf{z}(t) = \Phi(t)\mathbf{c} + \mathbf{z}_P(t) \qquad (9e)$$

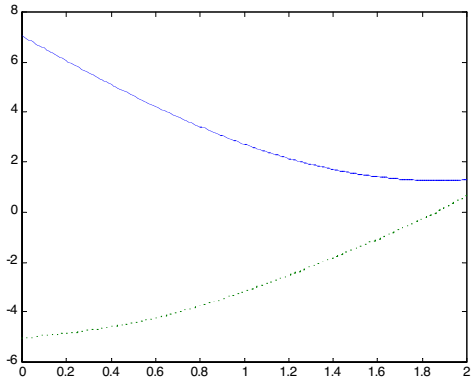
at the points t_k returned by ODE45. As evaluation points for ODE45, we chose 201 points equally spaced in $0 \le t \le 2$; that is, the evaluation points had a spacing of $\Delta t = 0.01$.

We implemented these ideas in a script, bvpexample2.m. In general, there are many different ways to code a particular solution procedure. In this case, since the procedure is somewhat complicated, we have opted for clarity rather than trying for the most efficient program. In particular, our script translates as literally as possible the five steps listed in equations (9a) - (9e). The system designated `'sys1'` in the script represents the system in equations (9a) and (9b). The system designated `'sys2'` represents equation (9c). The matrix denoted `PHI2` in the tenth statement is the matrix $\Phi(2)$ in equation (9d). The matrix denoted `PHI` in the seventeenth statement is the matrix $\Phi(t)$ in equation (9e).

```
% This script, bvpexample2, solves the two-point
% boundary value problem (8), using equations
%  (9a)(9c).
%
tspan = 0:.01:2;
y0 = [1,0]';
[t,phi1] = ode45('sys1',tspan,y0);
y0 = [0,1]';
[t,phi2] = ode45('sys1',tspan,y0);
y0 = [0,0]';
[t,z] = ode45('sys2',tspan,y0);
p1 = [1 1;0 0];
p2 = [0 0;1 -2];
PHI2 = [phi1(201,:)',phi2(201,:)'];
M = p1+p2*PHI2;
alpha = [2,0]';
c = inv(M)*(alpha-p2*z(201,:)');
soln = zeros(2,201);

for k = 1:201
   A = [0 1;t(k)^2 0];
   PHI = [phi1(k,:)',phi2(k,:)'];
   soln(:,k) = PHI*c+z(k,:)';
end
```

```
plot(t,soln(1,:),t,soln(2,:),':')
title('solution is solid, derivative is dotted')
xlabel('t')
ylabel('solution and derivative')
```

The graph of the resulting solution is displayed below.



The two-point boundary value problem (8) can also be solved using the shooting method described in Exercises 29-30 in Section 13.1. As is described in Exercises 29-30, we begin by constructing numerical solutions for the following two initial value problems:

$$y_c'' - t^2 y_c = 0, \quad y_c(0) = 1, y_c'(0) = -1 \qquad \text{(10a)}$$

$$y_p'' - t^2 y_p = e^{-t}, \quad y_p(0) = 1, y_p'(0) = 1. \qquad \text{(10b)}$$

In the notation of Exercise 29, $c_0 = -1/2$ and $c_1 = 1/2$. In particular, the initial conditions have been chosen so that the solution of the homogeneous differential equation (10a) satisfies the homogeneous boundary condition

$$y_c(0) + y_c'(0) = 0$$

while the solution of the nonhomogeneous differential equation (10b) satisfies the given boundary condition of (8),

$$y_p(0) + y_p'(0) = 2.$$

The function $y(t) = sy_c(t) + y_p(t)$ is a solution of the nonhomogeneous differential equation in (8) that likewise satisfies the given boundary condition at $t = 0$ for all choices of the shooting parameter s. Therefore, we solve problems (10a)-(10b) and then impose the boundary condition at $t = 2$ in order to determine s:

$$s = -\frac{y_p(2) - 2y_p'(2)}{y_c(2) - 2y_c'(2)}.$$

Note that the Fredholm alternative theorem guarantees that the denominator of the above quotient is nonzero. The MATLAB code needed to implement the shooting method is similar to the earlier program bvpexample2.m, except that it is substantially shorter. The graphical results are identical and we do not include them. The system designated `'sysh'` in the script represents the homogeneous system in equation (10a). The system designated `'sysnh'` represents the nonhomogeneous system in equation (10b).

```
%
% This script, shooting.m, solves the two-point
% boundary value problem (8) using the shooting
%  method.
%
tspan = 0:.01:2;
y0 = [ 1,-1] ';
[ t,yc] = ode45('sysh',tspan,y0);
y0 = [ 1,1] ';
[ t,yp] = ode45('sysnh',tspan,y0);
s = -(yp(201,1)-2*yp(201,2))/(yc(201,1)-2*yc(201,2));
soln = zeros(2,201);

for k = 1:201
   soln(:,k) = s*yc(k,:)'+yp(k,:)';
end

plot(t,soln(1,:),t,soln(2,:),':')
title('solution is solid, derivative is dotted')
xlabel('t')
ylabel('solution and derivative')
```