

Appendix

John D. Enderle, PhD

A.1 MATLAB

MATLAB is a high-level computer program that performs technical computing and plotting based on matrices. It is a programming language that has many of the same capabilities as FORTRAN, C and other programming languages, but is much easier to use. This section introduces the reader to MATLAB in sufficient detail so that the program can be used to solve relevant problems in this book. Other MATLAB features are introduced in later chapters as needed.

A.1.1 Matrix Basics

MATLAB was originally developed as MATrix LABoratory, with most commands stated in terms of matrices. A matrix is a rectangular array consisting of n rows and m columns of elements. The array \mathbf{A} , denoted with a bold uppercase letter, is denoted as

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \quad (\text{A.1})$$

where each element of the array, a_{ij} , is a constant or a function. The order of array \mathbf{A} is referred to as $n \times m$. If $n = m$, \mathbf{A} is called a square matrix of order n . For our purposes, the matrix \mathbf{A} usually stores the parameters for a system, such as values for resistors, capacitors and inductors.

An array of one column is referred to as a column vector, that is, an array of order $n \times 1$. We typically denote a column vector with a bold lowercase letter, such as

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad (\text{A.2})$$

where each element x_i is a variable, constant or a function of time. The column vector is useful for representing variables and inputs for a system. An array of one row is referred to as a row vector, that is, an array of order $1 \times n$. The row vector is useful for representing a polynomial of the characteristic function (defined later) for a system. A matrix of order 1×1 is a scalar.

Addition and subtraction of matrices are valid only for matrices of the same order. Either operation is applied to corresponding elements term by term, that is,

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{bmatrix} = \begin{bmatrix} c_{11} = a_{11} + b_{11} & c_{12} = a_{12} + b_{12} & \cdots & c_{1m} = a_{1m} + b_{1m} \\ c_{21} = a_{21} + b_{21} & c_{22} = a_{22} + b_{22} & \cdots & c_{2m} = a_{2m} + b_{2m} \\ \vdots & \vdots & & \vdots \\ c_{n1} = a_{n1} + b_{n1} & c_{n2} = a_{n2} + b_{n2} & \cdots & c_{nm} = a_{nm} + b_{nm} \end{bmatrix} \quad (\text{A.3})$$

where, in general, $c_{ij} = a_{ij} + b_{ij}$. Subtraction follows similarly with $c_{ij} = a_{ij} - b_{ij}$.

Matrix multiplication is valid only when the number of columns in the first matrix is equal to the number of rows in the second matrix. If **A**, order $n \times m$, is multiplied by **B**, order $m \times n$, then the order of the resulting matrix **C** is $n \times n$. Each element of **C**, c_{ij} , is equal to the sum of products of the i^{th} row of **A** with the j^{th} column of **B**, that is

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj} \quad (\text{A.4})$$

for $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, m$. In general, matrix multiplication for matrices of the same order is not commutative, that is, $\mathbf{AB} \neq \mathbf{BA}$. The multiplication of a matrix by a scalar α equals the product of each element of the matrix by α .

The identity matrix **I** is a square matrix whose nondiagonal elements are zero and whose diagonal elements are one, that is

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} \quad (\text{A.5})$$

The null matrix, **0**, has 0 for all elements in the matrix.

Matrix operations of addition, subtraction, multiplication, and division follow much the same processes that these operations do with real numbers. For arbitrary and appropriately defined matrices **A**, **B**, and **C**, we have

Commutative Property: $\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$

Associative Property: $\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$

Distributive Property: $\mathbf{A}(\mathbf{B} + \mathbf{C}) = \mathbf{AB} + \mathbf{AC}$

Identities Involving I and 0:

$$\mathbf{AI} = \mathbf{A}$$

$$\mathbf{0A} = \mathbf{0}$$

$$\mathbf{A0} = \mathbf{0}$$

$$\mathbf{A} + \mathbf{0} = \mathbf{A}$$

The inverse of a square matrix \mathbf{A} , denoted, \mathbf{A}^{-1} , is defined by

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I} \quad (\text{A.6})$$

The use of the matrix inverse is important in solving a set of simultaneous equations that describe a system.

The transpose of matrix \mathbf{A} is denoted as either \mathbf{A}' or \mathbf{A}^T , where we interchange rows and columns of \mathbf{A} to form \mathbf{A}^T as follows from Equation (A.1).

$$\mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & & \vdots \\ a_{1m} & a_{2m} & \cdots & a_{nm} \end{bmatrix}$$

SIMULTANEOUS EQUATIONS AND MATRICES

A set of equations with unknown variables often results in the process of analyzing a system, after applying interconnection laws or other techniques. At other times, a set of equations with unknown parameters results in solving the coefficients of a differential equation with initial conditions.

Suppose the following set of equations describes the currents in an electric circuit, i_1 , i_2 , and i_3 , after applying Kirchhoff's voltage law.

$$\begin{aligned} 2i_1 - i_2 + 0i_3 &= 6 \\ -2i_1 + 3i_2 + i_3 &= 0 \\ -i_1 + 5i_2 - 4i_3 &= 0 \end{aligned} \quad (\text{A.7})$$

The constants multiplying the currents involve the resistors in the circuit, with the value of 6 due to the input to the circuit. In writing each equation in (A.7) we have included all variables, written in order, even if a variable is multiplied by 0.

Equation (A.7) is written in matrix form as

$$\begin{bmatrix} 2 & -1 & 0 \\ -2 & 3 & 1 \\ -1 & 5 & -4 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \\ 0 \end{bmatrix} \quad (\text{A.8})$$

or

$$\mathbf{A}\mathbf{i} = \mathbf{F} \quad (\text{A.9})$$

with appropriately defined matrices

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 \\ -2 & 3 & 1 \\ -1 & 5 & -4 \end{bmatrix}, \mathbf{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \end{bmatrix}, \text{ and } \mathbf{F} = \begin{bmatrix} 6 \\ 0 \\ 0 \end{bmatrix}$$

To solve Equation (A.8) for the current vector \mathbf{i} , we premultiply both sides of the equation by the inverse \mathbf{A}^{-1} , that is

$$\mathbf{A}^{-1}\mathbf{A}\mathbf{i} = \mathbf{A}^{-1}\mathbf{F} \quad (\text{A.10})$$

or

$$\mathbf{i} = \mathbf{A}^{-1}\mathbf{F} \quad (\text{A.11})$$

A.1.2 Using MATLAB

MATLAB is a program that evaluates mathematical expressions, plots results, and comes with a wide assortment of toolboxes (applications) for various engineering fields. The toolboxes include neural networks, imaging, digital signal processing, control theory, linear algebra, signals and systems, numerical methods, symbolic math, and SIMULINK. SIMULINK provides an iterative solution (i.e., an approximation) to very complex differential-integral equations. The plotting capabilities of MATLAB are sufficient for most work. For more sophisticated or professional plotting, output from MATLAB can be exported into Microsoft® EXCEL.

The material presented here is a very brief introduction to MATLAB. Readers interested in learning more about the program should look in the library and bookstores to find many books on this powerful program.

GETTING STARTED

To begin a MATLAB session in Microsoft Windows, click “Start”, “Programs”, the folder “MATLAB”, and then the program “MATLAB”. A command window opens with three parts as shown in [Figure A.1](#).

Our work at this stage is done in the MATLAB Command Window on the right side of [Figure A.1](#). To begin, we start typing in the Command Window after the “>>”. Text can be typed interactively with MATLAB executing the command after a carriage return (called CR hereafter) by clicking the “Enter” key. Keep in mind that MATLAB differentiates

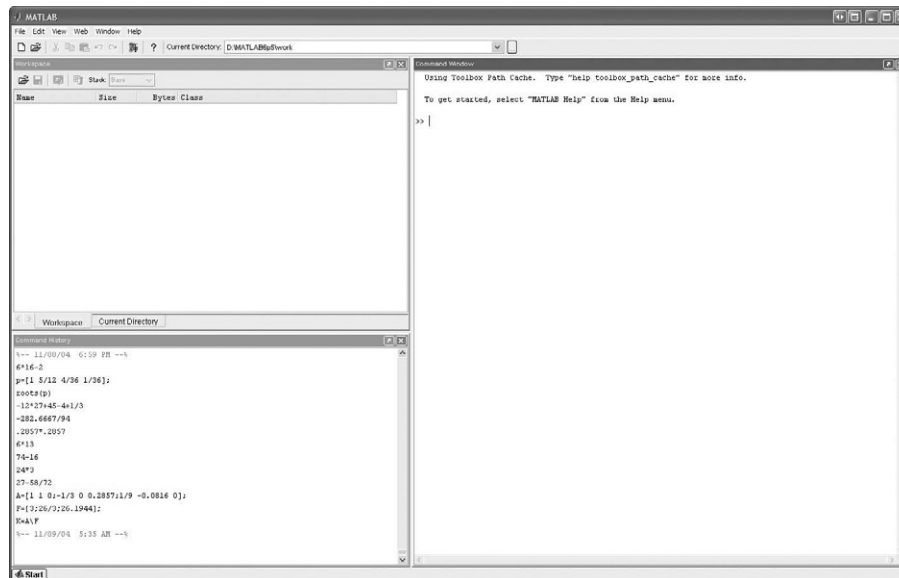


FIGURE A.1 MATLAB program with three windows. (1) Launch Pad Window, (2) Command Window, and (3) Command History.

between lower and uppercase in executing commands (so “a” is different from “A”), and it does not use italic or bold characters as defined in [Section A.1](#).

MATLAB evaluates scalars as well as matrices. For example, if we enter

```
>> A = 10/5
```

MATLAB returns with

```
A =  
    2
```

To enter a matrix, we start with the matrix name, then an equal sign followed by a left bracket, the elements of the array, and then a right bracket. Each element in a row is entered, beginning with the first row, separated by a blank space or a comma. To move from one row to the next row, a semicolon or a CR is entered. For example, matrix **A** is entered in MATLAB by writing

```
>> A = [5 9 6; 3 2 3; 5 9 1]
```

MATLAB returns with

```
A =  
    5    9    6  
    3    2    3  
    5    9    1
```

Matrix elements can also be entered as a function, such as

```
>> A = [- 5/2, sqrt(4), 3*(6 - 4)]
```

MATLAB returns with

```
A =  
-2.5000  2.0000  6.0000
```

Note that a “;” at the end of any line suppresses printing of that line. A line can also be extended by typing “...” and a CR at the end of any line so that the next line is just a continuation of the previous line.

For example,

```
>> C = [1 ...  
2 3  
4 5 6  
7 8 9]  
C =  
    1    2    3  
    4    5    6  
    7    8    9
```

Here the first row is written on two lines with “...” at the end of the first line and a CR rather than “;” separating the rows.

To invert matrix **A**, given by

```
>> A = [5 9 6; 3 2 3; 5 9 1]
```

```
type
```

```
>> B = inv(A)
```

MATLAB returns with

```
B =  
   -0.2941    0.5294    0.1765  
    0.1412   -0.2941    0.0353  
    0.2000     0      -0.2000
```

There are many special functions within MATLAB like the exponential and π . For example, to use π in a calculation, we use the word “pi”; enter the following and note the response.

```
>> pi  
ans =  
    3.1416
```

MATLAB returns a numerical result for each command entered. The exponential function uses the syntax “exp (– 0.5)”, which, when entered, gives 0.6065. Most standard elementary functions in a good scientific calculator, including built-in functions, are included in MATLAB, like the sine and cosine functions. The entire list of MATLAB functions is given from the “HELP” menu: select “MATLAB HELP”, then “Functions – Alphabetical List”.

A.1.3 Arithmetic Expressions

Arithmetic operations are carried out in MATLAB with the following symbols:

\wedge $*$ $/$ \backslash $+$ $-$

for power operator, multiplication, right division, left division, addition, and subtraction, respectively. Note that $1/4$ and $4\backslash 1$ both equal 0.25, and represent 1 divided by 4 and 4 divided into 1, respectively. We often use the “ \backslash ” operation when solving simultaneous equations as in Example A.6.

MATLAB performs operations in the order previously listed, that is, \wedge first, then $*$ or $/$ or \backslash calculated in order from left to right, and then $+$ or $-$ calculated in order from left to right. For example,

```
>> 5 + 6^2*3/7*2
```

```
gives
```

```
ans =  
    35.8571
```

This operation is done by first calculating $6^2 = 36$, then multiplying 36 by 3, followed by dividing this result by 7, then multiplying the entire result by 2, and then finally adding 5. The sequence of like math operations works from left to right, with parentheses used to change the order of calculation. For example,

```
>> 3*(6 - 4)
gives
ans =
    6
```

The first calculation is $6 - 4$ and then this result is multiplied by 3.

Trigonometric functions are calculated in MATLAB with the angle in radians. For example, to evaluate $\sin(45^\circ)$, we enter

```
>> sin(pi/4)
which gives
ans =
    0.7071
```

EXAMPLE PROBLEM A.1

Solve the following set of simultaneous equations for i_1, \dots, i_5 .

$$\begin{aligned} 4i_1 - i_2 - i_3 - i_4 - i_5 &= 240 \\ -i_1 + 8i_2 - i_3 + 0 \times i_4 + 0 \times i_5 &= 0 \\ -i_1 - i_2 + 5i_3 - i_4 + 0 \times i_5 &= 0 \\ -i_1 + 0 \times i_2 - i_3 + 5i_4 - i_5 &= 0 \\ -i_1 + 0 \times i_2 + 0 \times i_3 - i_4 + 8i_5 &= 0 \end{aligned}$$

Solution

The simultaneous equations are written in matrix format as

$$\mathbf{A}\mathbf{i} = \mathbf{F}$$

with

$$\mathbf{A} = \begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 8 & -1 & 0 & 0 \\ -1 & -1 & 5 & -1 & 0 \\ -1 & 0 & -1 & 5 & -1 \\ -1 & 0 & 0 & -1 & 8 \end{bmatrix}; \quad \mathbf{i} = \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \end{bmatrix}; \quad \mathbf{F} = \begin{bmatrix} 240 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Column vector \mathbf{i} is easily solved by using the reverse division operation on the previous equation, that is,

$$\mathbf{A} \setminus \mathbf{A}\mathbf{i} = \mathbf{i} = \mathbf{A} \setminus \mathbf{F}$$

The MATLAB commands to solve for \mathbf{i} are

```
>> A = [4 -1 -1 -1 -1; -18 -100; -1 -15 -10; -10 -15 -1; -100 -18];
>> F = [240;0;0;0;0];
>> i = A \ F
```

Continued

```
i =
    77.5000
    12.5000
    22.5000
    22.5000
    12.5000
```

A.1.4 Vectors

MATLAB stores vectors as a row vector with the first element as the **i** component, the second the **j** component, and the third the **k** component, all written within brackets. For example, the vector $\mathbf{F} = 3\mathbf{i} - 4\mathbf{j} + 2\mathbf{k}$ is entered in MATLAB as

```
>> F = [3 -4 2]
```

which gives

```
F =
     3    -4     2
```

To calculate the magnitude of the vector, the MATLAB command “norm” is used. For example, the magnitude of the vector **F** is calculated as

```
>> Fmagnitude = norm(F)
```

```
Fmagnitude =
     5.3852
```

MATLAB also computes the dot product of two vectors with the command “dot(A,B)”. For example, if $\mathbf{A} = 5\mathbf{i} - 8\mathbf{j} - 2\mathbf{k}$ and $\mathbf{B} = 6\mathbf{i} + 4\mathbf{j} - 3\mathbf{k}$, the dot product is computed in MATLAB as

```
>> A = [5 -8 -2];
>> B = [6 4 -3];
>> ABdot = dot(A, B)
ABdot =
     4
```

The cross product is also a built-in function in MATLAB using the command “cross(A,B)”. For example, $\mathbf{A} \times \mathbf{B}$ is written as

```
>> A = [5 -8 -2];
>> B = [6 4 -3];
>> ABCross = cross(A, B)
ABCross =
    32     3    68
```


A.1.5 Complex Numbers

MATLAB stores all numbers as complex numbers and uses either i or j to represent the imaginary component of complex numbers. In MATLAB $z = 3 + i4$ and $z = 3 + j4$ are equivalent, that is

```
>> z = 3 + 4j
```

gives

```
z =  
    3.0000 + 4.0000i
```

and

```
>> z = 3 + 4i
```

gives

```
z =  
    3.0000 + 4.0000i
```

As mentioned previously, here we use the symbol i for current (as in Example A.1) and the symbol j for the complex numbers. MATLAB, however, defaults to the symbol i . The polar form for complex numbers is also valid in MATLAB; that is, for any arbitrary real value x and y ,

$$z = x + jy = |z|e^{j\theta}$$

where

$$|z| = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1} \frac{y}{x}$$

MATLAB has many built-in functions for working with complex numbers, with a few essential ones given in [Table A.1](#).

TABLE A.1 MATLAB Commands for Complex Numbers with $z = x + jy$

Value	Syntax
$\text{Re}(z)$	<code>real(z)</code>
$\text{Im}(z)$	<code>imag(z)</code>
$ z = \sqrt{x^2 + y^2}$	<code>abs(z)</code>
$\theta = \tan^{-1} \frac{y}{x}$	<code>angle(z)</code>

EXAMPLE PROBLEM A.2

Use MATLAB to determine $z_3 = \frac{z_1}{z_2}$ if $z_1 = 3 + j2$ and $z_2 = 2 - j$.

Solution

In MATLAB we write

```
>> z1 = 3 + j * 2
z1 =
    3.0000 + 2.0000i
>> z2 = 2-j
z2 =
    2.0000 - 1.0000i
>> z3 = z1/z2
z3 =
    0.8000 + 1.4000i
```

EXAMPLE PROBLEM A.3

Evaluate

$$z = (60 + j30) + 100e^{-j0.4887}$$

Solution

To evaluate z , we use the following MATLAB commands, mixing the rectangular and polar forms of a complex number.

```
>> z = 60 + j*30 + 100*exp(-j*0.4887)
z =
    1.4829e + 002 - 1.6948e + 001i
```

EXAMPLE PROBLEM A.4

Evaluate the magnitude and angle of the following complex number:

$$z = (5 - j4)^3$$

Solution

```
>> z = (5 - j*4)^3
z =
   -1.1500e + 002 - 2.3600e + 002i
>> r = abs(z)
r =
    262.5281
>> theta = angle(z)
theta =
   -2.0242
```

A.1.6 Polynomials and Roots

MATLAB works with polynomials using a row vector and can calculate the roots of the polynomial using a built-in function. A polynomial is written as a row vector of the polynomial coefficients in descending order, starting with the highest order. Consider the polynomial

$$x^4 - 12x^3 + 0x^2 + 25x + 116 = 0 \quad (\text{A.12})$$

which is entered in MATLAB as

```
>> p = [1  -12  0  25  116]
```

returning

```
p =  
1  -12  0  25  116
```

It is important to enter every term, even terms with a 0 coefficient as in [Equation \(A.12\)](#). To find the roots of the polynomial, we use the built-in function “roots”. To find the roots of [Equation \(A.12\)](#), we write

```
>> r = roots(p)
```

and MATLAB returns with two real roots and a pair of complex conjugate roots:

```
r =  
11.7473  
2.7028  
-1.2251 + 1.4672i  
-1.2251 - 1.4672i
```

The convention for MATLAB is that polynomials are row vectors and roots are column vectors. One can work in the opposite direction and find the polynomial given as a set of roots via the MATLAB “poly” command. Continuing from the last MATLAB calculations to find the roots of a polynomial, use the “poly” command to restore the polynomial

```
>> pp = poly(r)
```

which returns

```
pp =  
1.0000  -12.0000  -0.0000  25.0000  116.0000
```

Polynomial multiplication is carried out using the MATLAB function “conv”, which performs the convolution of the two polynomials. Consider multiplying the following two polynomials using MATLAB.

$$A(x) = x^3 + 2x^2 + 3x + 4 \quad (\text{A.13})$$

$$B(x) = x^3 + 4x^2 + 9x + 16 \quad (\text{A.14})$$

In MATLAB we type

```
>> A = [1 2 3 4];
>> B = [1 4 9 16];
>> AB = conv(A, B)
```

which returns

```
AB =
    1    6   20   50   75   84   64
```

Note that the semicolon after the matrix A and B suppresses the echo of the command. Symbolically, the polynomial product of Equation (A.13) and Equation (A.14) is given by

$$AB(x) = x^6 + 6x^5 + 20x^4 + 50x^3 + 75x^2 + 84x + 64 \quad (\text{A.15})$$

To add polynomials together, the polynomials need to be of the same order. For example, consider adding the following two polynomials:

$$S(x) = x^5 + 4x^4 + 5x^3 + 6x^2 + 7x + 2 \quad (\text{A.16})$$

and

$$T(x) = 2x^4 + 9x^3 + 8x^2 - 4x + 5 \quad (\text{A.17})$$

Since the polynomial in Equation (A.17) is 4th order, we pad the row vector with a zero in place of a x^5 coefficient. Thus the polynomials are entered in MATLAB as

```
>> S = [1 4 5 6 7 2];
>> T = [0 2 9 8 -4 5];
>> U = S + T
```

which returns

```
U =
    1    6   14   14    3    7
```

or symbolically

$$x^5 + 6x^4 + 14x^3 + 14x^2 + 3x + 7 \quad (\text{A.18})$$

To divide one polynomial by another, the MATLAB command “deconv(A,B)” is used, which represents B divided into A, or equivalently A divided by B. As there is usually a remainder from polynomial division, the syntax used is [Q, V] = deconv(A,B), where Q is the quotient polynomial and V is the remainder. Using MATLAB, the ratio of polynomials given by Equations (A.16) and (A.17), that is, $x^5 + 4x^4 + 5x^3 + 6x^2 + 7x + 2$ divided by $2x^4 + 9x^3 + 8x^2 - 4x + 5$, is written

```
>> S = [1 4 5 6 7 2];
>> T = [2 9 8 -4 5];
>> [Q, R] = deconv(S, T)
Q =
    0.5000 - 0.2500
```

```
R =
    0    0  3.2500 10.0000  3.5000  3.2500
```

Symbolically, this result is written as

$$\frac{1}{2}x - \frac{1}{4} + \frac{3.25x^3 + 10x^2 + 3.5x + 3.25}{x^5 + 2x^4 + 9x^3 + 8x^2 - 4x + 5}$$

To illustrate a simpler example with no remainder, divide $x^2 - 2x + 1$ by $x - 1$, giving

```
>> S = [1, -2, 1];
>> T = [1, -1];
>> [Q, V] = deconv(S, T)
```

```
Q =
    1   -1
```

```
V =
    0    0    0
```

The result is $x - 1$.

EXAMPLE PROBLEM A.5

Consider the following equation:

$$\frac{(1600 - 3x)^3}{27x^3} = \frac{x}{500 - x}$$

Find x given that it must be between 0 and 500.

Solution

We use “format long” in this solution so that enough significant digits are carried through. This is necessary because ordinarily MATLAB evaluates expressions with “format short”. First evaluate $(1600 - 3x)^3$ as follows:

```
>> format long
>> y = [- 3 1600];
>> r = conv(y, y);
>> w = conv(y, r)
which returns
w =
    1.0e + 009*
-0.00000002700000  0.00004320000000 -0.02304000000000  4.09600000000000
```

where “w” is the expression $(1600 - 3x)^3$. Notice that “1.0e + 009” multiplies all of the coefficients in the previous polynomial.

We cross-multiply to eliminate the denominator term on the right side as follows:

```
>> z = [- 1 500];
and multiplying w*z is done by another “conv”
```

Continued

```
>> s = conv(z, w)
resulting in
s =
    1.0e + 012*

Columns 1 through 4
0.00000000002700   -0.00000005670000   0.00004464000000   -0.01561600000000
Column 5
2.048000000000000
```

The left-side denominator term $27x^3$ multiplied by the right-side numerator term x is $(27x^4)$, written in MATLAB as

```
>> q = [27 0 0 0 0]
```

giving

```
q =
    27     0     0     0     0
```

Next, we subtract the two polynomials, using the MATLAB command

```
>> t = s - q
```

resulting in

```
t =
    1.0e + 012*

Columns 1 through 4
    0 - 0.00000005670000   0.00004464000000   -0.01561600000000
Column 5
2.048000000000000
```

The polynomial evaluated by the previous steps can be written as

$$-5.67 \times 10^4 x^3 + 4.464 \times 10^7 x^2 + 1.5616 \times 10^{10} x + 2.048 \times 10^{12} = 0$$

The final step in the problem is to evaluate the roots of the resulting polynomial:

```
>> v = roots(t)
```

which yields

```
v =
    1.0e + 002*
    2.62504409150663 + 2.62297313283571i
    2.62504409150663 - 2.62297313283571i
    2.62292769000261
```

Two of the roots are complex and are eliminated because they fall outside the problem constraints, giving a solution of

$$x = 262.3$$

Table A.2 summarizes some MATLAB commands used for evaluating polynomials.

TABLE A.2 MATLAB Commands for Polynomials

Operation	Symbolic Expression	Syntax
Polynomial	$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$	$P = [a_n a_{n-1} \cdots a_1 a_0]$
Roots of a polynomial	$(x - r_1)(x - r_2) \cdots (x - r_n)$	$r = \text{roots}(P)$
Write a polynomial from the roots	$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$	$\text{poly}(r)$
Multiply two polynomials $S(x) = A(x) \times B(x)$	$S(x) = (a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0) \times (b_m x^m + b_{m-1} x^{m-1} + \cdots + b_1 x + b_0)$	$S = \text{conv}(A, B)$
Add two polynomials $S(x) = A(x) + B(x)$; note that the order of the polynomials must be the same	$S(x) = (a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0) + (0x^n + \cdots + b_m x^m + b_{m-1} x^{m-1} + \cdots + b_1 x + b_0)$ $S(x) = \frac{a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0}{b_m x^m + b_{m-1} x^{m-1} + \cdots + b_1 x + b_0} = q_p x^p + \cdots + q_1 x + q_0$	$S = A + B$
Divide two polynomials $S(x) = \frac{A(x)}{B(x)} = Q(x) + V(x)$	$+ \frac{v_l x^l + \cdots + v_1 x + v_0}{b_m x^m + b_{m-1} x^{m-1} + \cdots + b_1 x + b_0}$	$[Q, V] = \text{deconv}(A, B)$

A.1.7 Plotting with MATLAB

MATLAB has extensive plotting capabilities for two- and three-dimensional graphing of vectors and matrices for data visualization and presentation graphics. The appearance of the graphs, including line widths, color, annotations, and labeling can be customized to meet the needs of the user.

fplot

The basic plotting command in MATLAB is “fplot(‘f’, [T₁, T₂])”, which plots the function f over the interval T_1 to T_2 . Note that the expression “f” is placed within apostrophes so that MATLAB associates this as a string expression rather than a function to be evaluated at time “t”. Consider the plotting $f = 2e^{-t}$ in the interval 0 to 5 using MATLAB. We enter the following command:

```
>> fplot('2*exp(-t)', [0, 5])
```

The command “fplot” graphs the function f in the time interval from 0 to 5 in a separate window as shown in Figure A.2. To edit the properties of the plot shown in Figure A.2, select the arrow icon (next to the print icon) and then double-click on the graph. Another command window opens whereby one can change the color of the line, line styles, fonts, axes labels, axes, etc.

Graphs are easily moved from MATLAB to other applications by copying the graph to the clipboard (Edit, Copy Fig.) and then pasting it into another application such as Microsoft® Word. One has control of the graph copied to the clipboard by editing the Copy Options from the Edit menu. The “Metafile” format copies the graph without background color. The “Bitmap” option copies an exact replica of the graph including the background color.

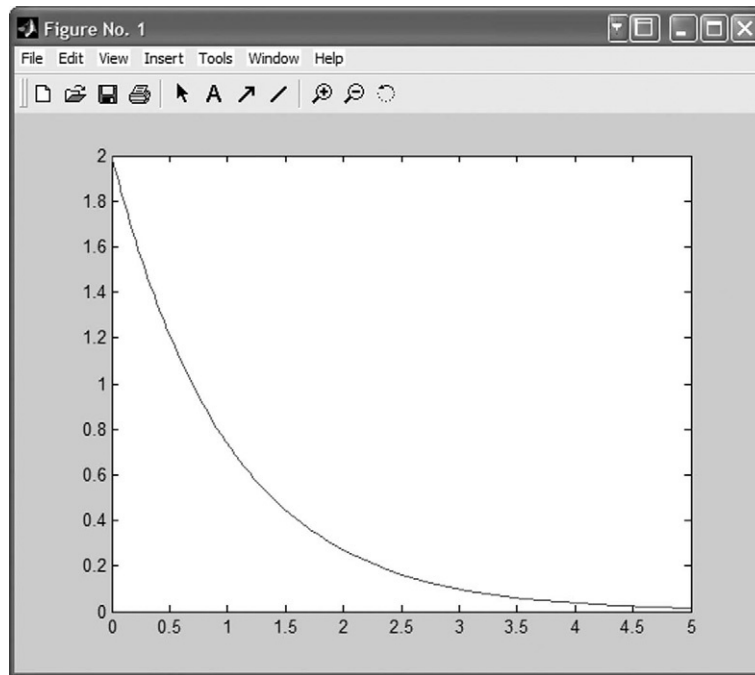


FIGURE A.2 MATLAB plot of the function $f = 2e^{-t}$ plotted using the command `fplot`.

plot

Another plotting command is “`plot(t,y)`”, which plots y versus t with points connected by straight lines. The independent variable t is easily defined using the “`linspace`” command with the syntax “`t = linspace(a,b,n)`” that generates a row vector of n linearly spaced elements over the timespan a to b .

For example, we can plot the function $y = 210\sqrt{2}e^{-2t} \sin(\sqrt{2}t)$ using the following commands:

```
>> t = linspace(0, 8, 1000);
>> y = 297*sin(1.414*t).*exp(-2*t);
>> plot(t, y)
```

The first command creates a 1000-element row vector t , with values ranging from 0 to 8. The second command creates a 1000-element row vector y by multiplying $\sin(1.414*t)$ by $\exp(-2*t)$ with the “`.*`” operation. The “`.*`” operation is the dot multiplication function that performs an element-by-element multiplication. MATLAB generates an error if one uses “`*`” instead of “`.*`” in the previous calculation, because it associates t as a 1×1000 row vector, which forces $\sin(1.414*t)$ as a 1×1000 row vector and $\exp(-2*t)$ as a 1×1000 row vector. In this case, the matrix multiplication is not valid because the order of the matrices does not agree. The command “`plot`” graphs the function y in the time interval from 0 to 8 with 1000 connected points as shown in [Figure A.3](#).

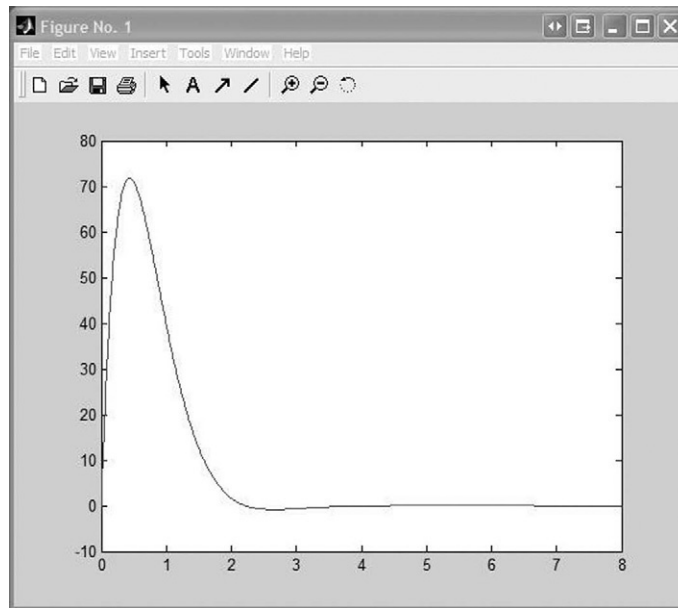


FIGURE A.3 The function $y = 210\sqrt{2}e^{-2t} \sin(\sqrt{2}t)$ graphed using the MATLAB “plot” command.

It is also possible to plot two functions on a graph using the “plot” command. To plot $f = 100e^{-t}$ and $y = 210\sqrt{2}e^{-2t} \sin(\sqrt{2}t)$ on the same graph, enter the following:

```
>> t = linspace(0, 8, 1000);
>> y = 297*sin(1.414*t).*exp(-2*t);
>> f = 100*exp(-t);
>> plot(t, y, t, f)
```

which generates the graph in [Figure A.4](#). As before, the graph is fully customizable, with the line color, line styles, fonts, axes labels, axes, etc., changed to the user’s needs.

There are many more plotting functions available within MATLAB, including three-dimensional graphs, histograms, pie charts, etc. The reader is directed to the Help function in MATLAB to discover all of the plotting capabilities.

Plotting with Microsoft Excel

While MATLAB provides very good plotting capabilities, exporting the data to other applications such as Microsoft® Excel gives graphics a more professional appearance. Before plotting the data in Excel, we need to first export the data from MATLAB and then import it into Excel. All data in MATLAB is stored under variable names in a common workspace accessible to all toolboxes, including SIMULINK. All variables used in the MATLAB workspace are given by the “who” or “whos” command. The “who” command simply lists the variables and the “whos” command lists the variables along with their sizes, number of bytes used, and class (i.e., logical character, integer array, floating point number array).

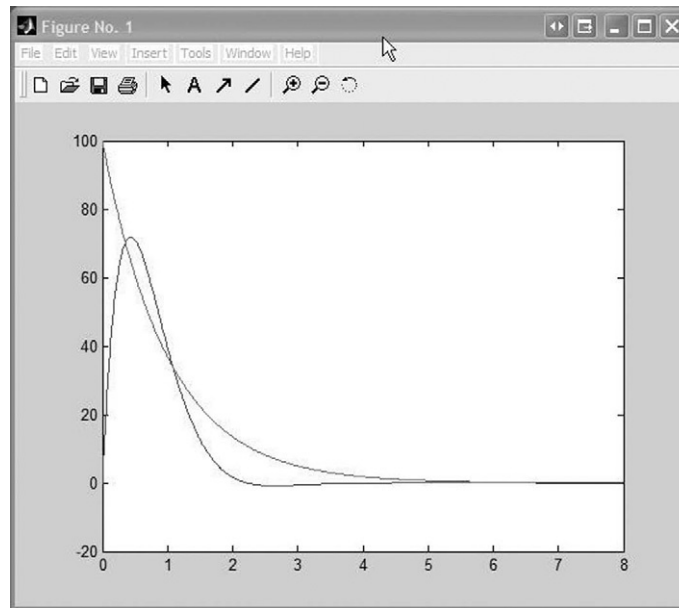


FIGURE A.4 The functions $f = 100e^{-t}$ and $y = 210\sqrt{2}e^{-2t} \sin(\sqrt{2}t)$ graphed using the MATLAB “plot” command.

Before exporting the data, it is easiest to collect the data in a single matrix with the independent variable listed first, along with the dependent variables—each evaluated at the points listed for the independent variable. Consider exporting the function shown in [Figure A.3](#), where t and y are given by

```
>> t = linspace(0, 8, 1000);
>> y = 297*sin(1.414*t).*exp(-2*t);
```

Keep in mind that t and y are stored as 1×1000 row vectors, so we must take the transpose of each row vector when concatenating to form the new matrix *data* by

```
>> data = [t', y'];
```

Data is stored in the matrix *data* with order 1000×2 . If there are additional vectors to be exported as in [Figure A.4](#), we concatenate all the data with the command “data = [t', y', f'];”.

To export the data to an arbitrary ASCII file “output.txt”, with tabs separating each variable, we use the following command:

```
>> save output.txt data - ascii - tabs
```

This file is stored in the MATLAB subfolder “work” by default. Next, open the file “output.txt” in Excel. At this time, the data are stored in the first two columns in the workspace. From here one can use the step-by-step “chart wizard” to create a “scatter chart” like that shown in [Figure A.5](#).

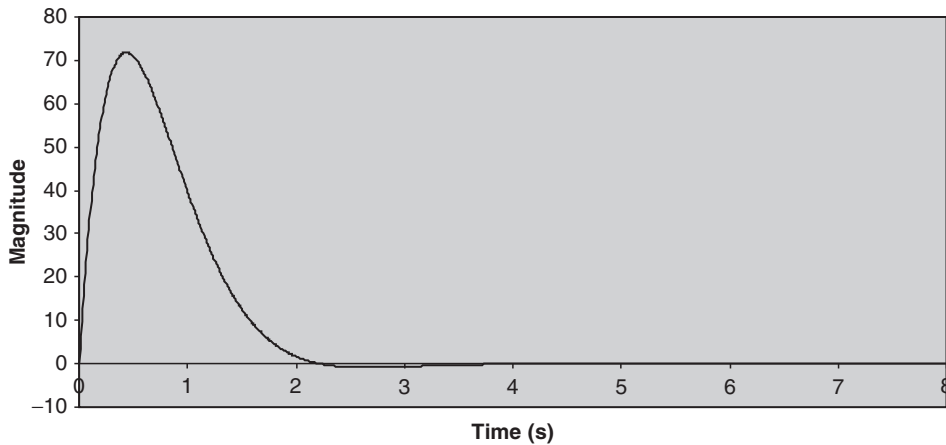


FIGURE A.5 The function $y = 210\sqrt{2}e^{-2t} \sin(\sqrt{2}t)$ drawn using Microsoft Excel.

A.1.8 Loading, Quitting and Saving the Workspace

Most problems can be solved interactively using MATLAB by entering information on the command line, pressing Enter, and having MATLAB evaluate the expression. For more challenging and lengthy projects, one can use m-files to store work to be retrieved later for further analysis. As a user becomes more proficient in MATLAB, m-files are used more frequently to store MATLAB commands, data, and input-output. The default extension for MATLAB is .mat. To create an m-file, one can save the workspace, or write the commands in Windows Notepad and load the commands in MATLAB with the load command. All of the expressions in the m-file are computed when loaded.

To open an m-file, type

```
>> load "filename.mat"
```

The command "save" is used to save the workspace to an m-file. If no filename is given, MATLAB defaults to "matlab.mat."

A.2 SOLVING DIFFERENTIAL EQUATIONS USING MATLAB

MATLAB provides the Symbolic Math Toolbox for symbolically solving many calculus and algebraic problems. The Symbolic Math Toolbox is a collection of commands that operate on functions based on a program called Maple[®] developed at the University of Waterloo in Ontario, Canada. Unlike previous exercises in which the answers MATLAB provided were numbers, using this toolbox results in answers that are symbolic functions. Examples of symbolic functions are

$$\tan(t^2), \quad \frac{d}{dt}(4y^2 + 2y + 4), \quad \int \sin(y)e^{-2y}dy, \quad \text{and} \quad 4\ddot{y} + 2\dot{y} + 6 = e^{-2t}.$$

MATLAB has commands among its features in this toolbox that determine the exact symbolic solution for a derivative or an integral of a function and for differential equations. For example, if we want to compute the derivative of $\tan(t^2)$, MATLAB provides the answer $2t(1 + \tan(t^2)^2)$. In this section, we introduce a few of these features, which are appropriate for solving problems in this chapter.

Before performing symbolic computations, we need to declare all of the symbolic variables. To write a symbolic variable we enclose it between single quotation marks using the “sym” MATLAB command or use the “syms” command. For example, the command

```
>> y = sym('y')
```

creates a symbolic variable “y”. Instead of defining a symbolic variable using the “sym” command, we can write it directly by writing the function within single quotes as follows:

```
>> f = 'tan(t^2)'
f =
    tan(t^2)
```

which creates a symbolic variable “f” that is stored in memory as “tan (t^2)”. To declare more than one symbolic variable, use the “syms” command. For example,

```
>> syms x y w z
```

or as

```
>> syms('w', 'x', 'y', 'z')
```

declares two four symbolic variables. This command is equivalent to the “sym” command for each variable separately. Whenever a symbolic variable is used after it is declared as a symbolic variable, any function using it is also a symbolic function. For example,

```
>> f = tan(y^2)
f =
    tan (y^2)
```

Notice that in the previous MATLAB command, we did not need to write $f = \tan(t^2)$ (with single quotes) because “y” is already declared as a symbolic variable. These variables are now available to be used with any MATLAB command to create symbolic results. With “w”, “x”, “y”, and “z” declared as symbolic variables, we can create a symbolic determinant by using a symbolic matrix using the MATLAB “det” command as follows:

```
>> m = [w, x;y, z]
m =
    [w, x]
    [y, z]
>> det(m)
ans =
    w*z - x*y
```

The MATLAB commands “diff” and “int” are the symbolic derivative and the indefinite integral functions, respectively. The following illustrates these two symbolic functions with some common expressions.

```
>> x = diff('cos(y)')
      x =
      -sin(y)
and
>> x = int('sin(y)')
      x =
      -cos(y)
```

Sometimes the result of a MATLAB operation gives an answer that is not easily recognizable. The MATLAB command “simplify” algebraically simplifies the result to a form that is more readily recognized. For example,

```
>> simplify((x^2 + 7*x + 12)/(x + 4) )
=
x + 3
```

As we will see, this command is very useful during integration and differentiating operations.

MATLAB also calculates definite integrals by including the limits of integration as arguments in the “int” command. For example, to calculate $\int_0^{\pi} \cos(y)dy$, we use the following MATLAB command

```
>> x = int('sin(y)', '0', 'pi')
      x =
      2
```

The limits of integration do not have to be numbers, but can be symbolic variables. For example,

```
>> x = int('sin(y)', 'a', 'b')
      x =
      -cos(b) + cos(a)
```

The “solve” command is convenient for solving symbolic algebraic expressions. For example, to solve for the roots of the polynomial $x^4 + 14x^3 + 71x^2 + 154x + 120 = 0$, we use

```
>> solve('x^4 + 14*x^3 + 71*x^2 + 154*x + 120 = 0')
ans =
[-5]
[-4]
[-3]
[-2]
```

Recall from the previous chapter that we could calculate the roots of a polynomial using the “roots” command, that is

```
>> p = [1 14 71 154 120];
>> r = roots(p)
    r =
    -5.0000
    -4.0000
    -3.0000
    -2.0000
```

The difference between the two commands is that “solve” allows us to symbolically solve for the roots without specifying the coefficients of the polynomial such as the well-known quadratic equation:

```
>> syms a b c y x
>> solve('a*x^2 + b*x + c = 0')
ans =
    [1/2/a*(-b + (b^2 - 4*a*c)^(1/2))]
    [1/2/a*(-b - (b^2 - 4*a*c)^(1/2))]
```

In solving for the response in many dynamic systems we need to solve a system of equations for several unknown variables such as the coefficients in the natural response. Suppose we needed to solve $K_1 + K_2 = 3$ and $-5K_1 - 3K_2 = 299$, which we did using a matrix approach. We can also solve a system of equations using the “solve” command by typing

```
>> syms k1 k2
>> [k1, k2] = solve('k1 + k2 = 3', '-5*k1 -3*k2 = 299')
    k1 =
        -154
    k2 =
        157
```

There are many other syntaxes available using the “solve” command; use the MATLAB help command to learn about them or consult other references.

The MATLAB command for solving ordinary differential equations is “dsolve”. The syntax involves using the capital letter “D” to denote a derivative, “D2” to denote the second derivative, “D3” to denote the third derivative, and so on. The syntax for the argument of this command involves writing the given differential equation and the initial conditions, each separated by a comma and enclosed in a single quote. For example, to solve the following differential equation for $t \geq 0$

$$\ddot{y} + 4\dot{y} + 3y = 0$$

with initial conditions $y(0) = 1$ and $\dot{y}(0) = 0$ using MATLAB’s “dsolve”, we have

```
>> dsolve('D2y + 4*Dy + 3*y = 0', 'Dy(0) = 0', 'y(0) = 1')
ans =
    -1/2*exp(-3*t) + 3/2*exp(-t)
```

To plot these results, we use the MATLAB plotting function called “ezplot”, an easy way to plot functions defined by symbolic functions. To plot the previous result, we execute

```
>> ezplot(y, [0, 5])
```

which gives the results shown in [Figure A.6](#).

The argument enclosed in the brackets of the “ezplot” command is the initial and final points.

If the initial conditions are not entered in the “dsolve” command, the solution is calculated in terms of unknown coefficients. To illustrate, consider entering the command “dsolve” without initial conditions,

```
>> dsolve('D2y + 4*Dy + 3*y = 0')
```

which gives us

$$C1 * \exp(-t) + C2 * \exp(-3*t)$$

The values for C1 and C2 are determined from the initial conditions.

To solve

$$\ddot{y} + 16,000\dot{y} + 10^8 y = 5 \times 10^8 t$$

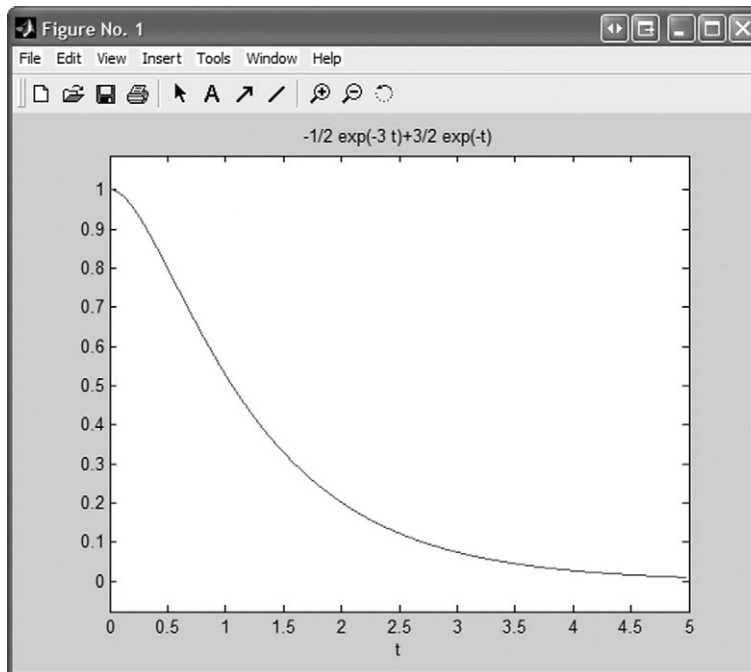


FIGURE A.6 An illustration of the MATLAB plotting function, “ezplot.”

with $\dot{y}(0) = 300$ and $y(0) = -45 \times 10^{-3}$ using MATLAB, we have

```
>> dsolve('D2y + 16000*Dy + 10^8*y = 5*10^8*t', 'Dy(0) = 300', 'y(0) = -45*10^(-3)')
ans =
```

$$-1/1250 + 5*t - 293/30000 * \exp(-8000*t) * \sin(6000*t) - 221/5000 * \exp(-8000*t) * \cos(6000*t)$$

To solve a set of simultaneous differential equations, we use the command “dsolve” with the argument syntax consisting of the given differential equations and initial conditions, each separated by a comma and enclosed in a single quote. For example, suppose we wish to solve:

$$\begin{aligned} 5\dot{y}_2 + 10\dot{y}_1 + 60y_1 &= 300u(t) \\ 5\dot{y}_2 + 40y_2 + 5\dot{y}_1 &= 0 \end{aligned}$$

with zero initial conditions at $t = 0$. Using MATLAB, we write

```
>> [x, y] = dsolve('5*Dy + 10*Dx + 60*y = 300',
    '5*Dy + 40*y + 5*Dx = 0', 'x(0) = 0', 'y(0) = 0')
x =
    -45 + 45*exp(12/17*t)
y =
    -5*exp(12/17*t) + 5
```

On occasion, MATLAB produces incorrect or misleading results to the solution of differential equations, so care should be taken in verifying the solution by plotting, checking the initial conditions and final conditions, or simulating the solution with SIMULINK.

A.3 BLOCK DIAGRAMS AND SIMULINK

As discussed earlier, solving a differential equation or a set of simultaneous differential equations involves a considerable amount of work. In the previous section, MATLAB’s symbolic math toolbox provided an easier solution method when it worked, but it sometimes failed to produce the correct solution. In this section, we use another MATLAB toolbox called SIMULINK to numerically calculate the solution for linear and nonlinear differential equations. For nonlinear differential equations, SIMULINK is usually the only solution option. To implement a numerical solution in SIMULINK, we draw the system of equations in a block diagram.

A.3.1 Block Diagrams

A block diagram is a graphical representation of the system’s differential equations using basic mathematical operations such as constants, gains, summing junctions or summers, and integrators. To deal with nonlinear elements, we can create special blocks that contain mathematical functions or standard functions such as the square root, exponential, and

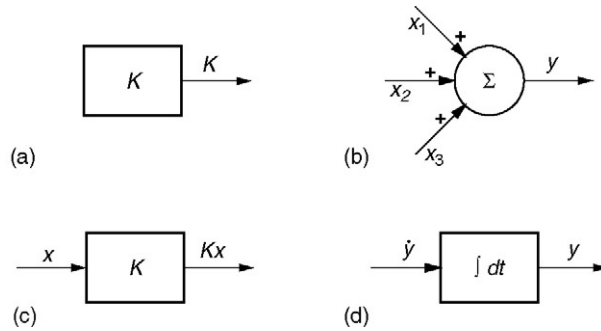


FIGURE A.7 Basic drawing elements in a block diagram: (a) constant, (b) summer, (c) gain, and (d) integrator.

logarithmic functions. The basic drawing elements for the block diagram are shown in Figure A.7.

To create a block diagram, we first take the highest derivative term output variable and put it on the left-hand side of the equal sign and the other terms on the right-hand side. Next, the block diagram is formed by connecting elements on the right-hand side to a summing junction representing the equation side, with the output being the left-hand side of the equation. From the summer, we include as many integrator blocks as necessary to reduce the highest order derivative to the output variable (i.e., $\frac{d^3y}{dt^3}$ requires three integrators). Finally, we use a gain block and connect the output variable and lower-order derivatives to appropriate gain blocks back to the summer. If there are any inputs, these are added to the summer using a constant block for a unit step function, or an appropriate function block such as t , or e^{-t} .

EXAMPLE PROBLEM A.6

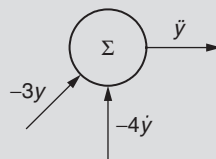
Draw a block diagram for $\ddot{y} + 4\dot{y} + 3y = 0$.

Solution

To begin the process of drawing a block diagram, the differential equation is rearranged as

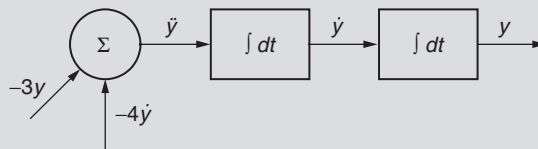
$$\ddot{y} = -4\dot{y} - 3y$$

and the block diagram is started by drawing the summer as shown in the following figure.

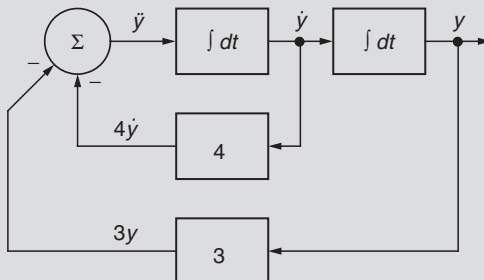


Next, we insert two integrator blocks to the output of the summer to obtain \dot{y} and y as illustrated in the following diagram.

Continued



Finally, the block diagram is completed by connecting \dot{y} and y to the summer via a gain block as shown in the following figure.



A block diagram can also be drawn for a set of simultaneous differential equations using the same approach as in Example A.6, drawing a block diagram for each equation and connecting the output variables and its derivatives to the appropriate summer.

EXAMPLE PROBLEM A.7

Draw a block diagram for the following set of simultaneous differential equations.

$$3\ddot{y}_1 + 10\dot{y}_1 + 60y_1 + 5\dot{y}_2 = f(t)$$

$$5y_1 + 4\ddot{y}_2 + 5\dot{y}_2 + 40y_2 = 0$$

where $f(t)$ is an unspecified input function.

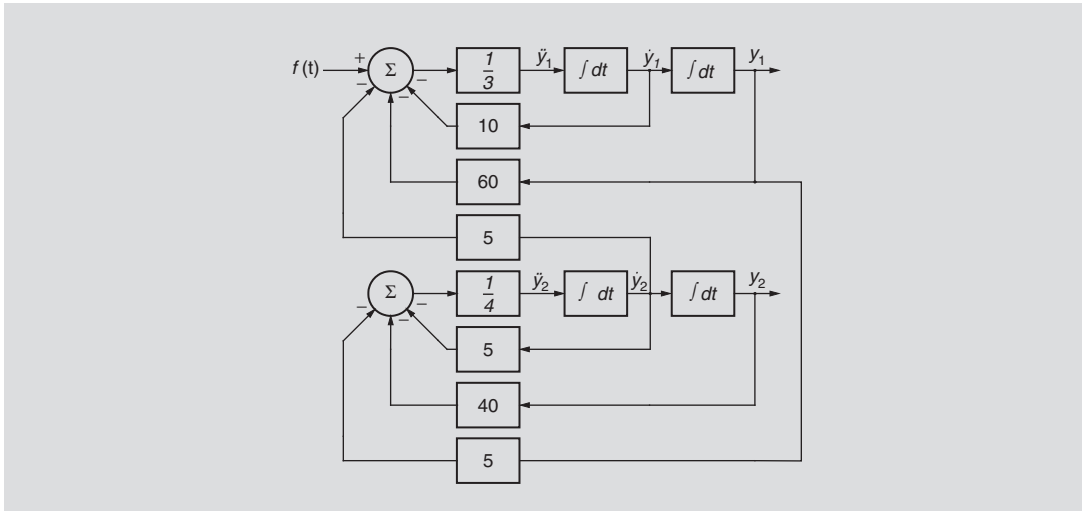
Solution

We begin by solving for the highest order derivative in each equation.

$$\ddot{y}_1 = \frac{1}{3} (-10\dot{y}_1 - 60y_1 - 5\dot{y}_2 + f(t))$$

$$\ddot{y}_2 = \frac{1}{4} (-5y_1 - 5\dot{y}_2 - 40y_2)$$

The block diagram is drawn by first using two summers, with \ddot{y}_1 and \ddot{y}_2 as the output of each summer, followed by two integrators for each output variable. Next, we complete the diagram by including the terms on the right-hand side of the previous equations as the input to each summer with appropriate gain elements, giving us the following figure.



A.4 SIMULINK

SIMULINK is a toolbox in MATLAB that simulates linear and nonlinear continuous and discrete dynamic systems. We program a system from the block diagram as described in the previous section using SIMULINK by dragging and dropping blocks and connecting the blocks together according to the system equation or equations. After building the model we run the program to plot results or export the results to MATLAB.

To start SIMULINK, start MATLAB as before, then type “simulink” in lowercase letters in the command window. This brings us to another window containing the icons for implementing a SIMULINK simulation as shown in [Figure A.8](#). Next, select “File” on the menu bar, then “New” and “Model”. This brings up another window, a workspace, that is used to create the SIMULINK model as shown in [Figure A.9](#). The icons shown make up the basic subsystem of blocks used to create a simulation, with each providing a set of blocks arranged into libraries. The libraries are a collection of blocks grouped into similar functions.

Double-clicking on any of the icons opens up the library of blocks. For example, double-clicking on the “Math Operations” library opens the window shown in [Figure A.10](#) with common operations such as absolute value (Abs), gain, math function, etc. The library “Continuous” contains the blocks for the derivative and integrator among other blocks. The library “Sources” contains the blocks for a constant, clock, step, etc., all blocks that produce an output without an input. The block “clock” outputs the current simulation time as a variable, useful in decision blocks and outputting variables to MATLAB. The library “Sinks” contains the blocks associated that have an input but no output, such as the scope that displays a variable as a function of time as if on an oscilloscope. Another important block in this library is the block “To Workspace”, which allows exporting a variable to MATLAB for further analysis and plotting.

Blocks are copied from the “SIMULINK Library Browser” window to the workspace window in [Figure A.9](#) by simply dragging them across from one to the other. Each block

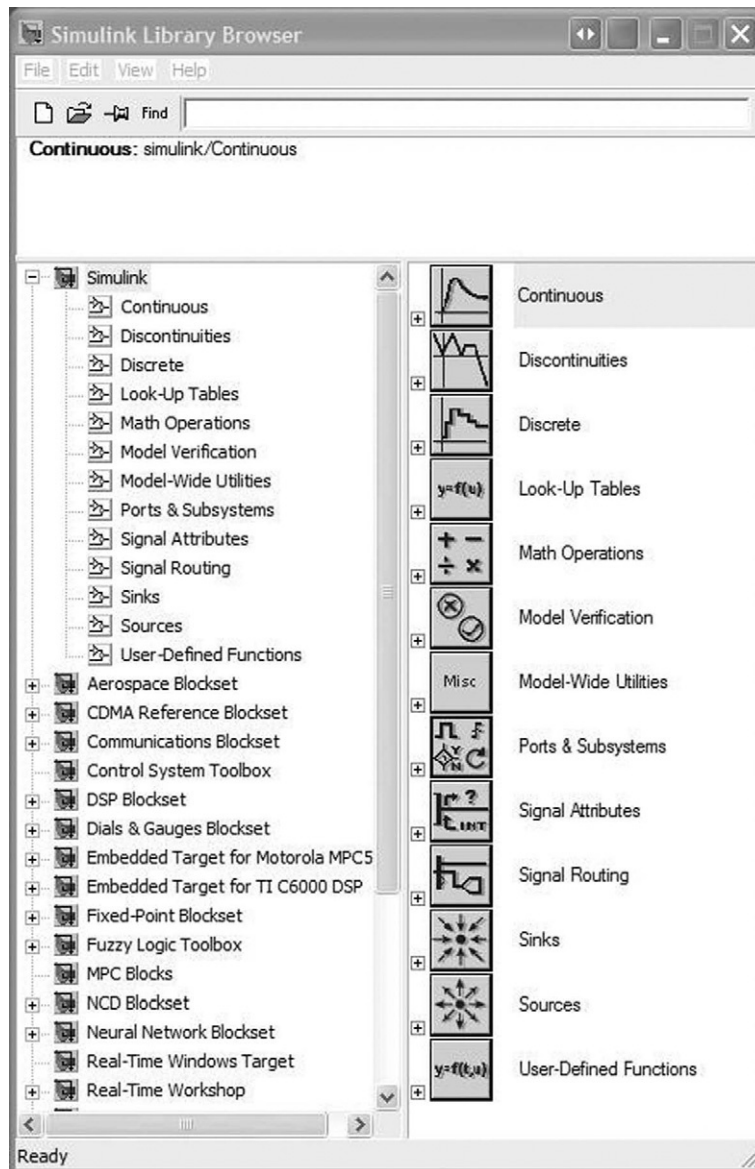


FIGURE A.8 SIMULINK Library of blocks on the right. Other MATLAB toolboxes are shown on the left.

is given a unique name below the block once it is placed in the workspace. If a block is repeated, its name is the generic block name followed by an integer. There is also a block identifier placed inside the block. The block name can be edited by placing the mouse pointer on the name, clicking the left mouse button, and then editing the name. Once in the workspace, the block's behavior is set by double-clicking the block, which opens

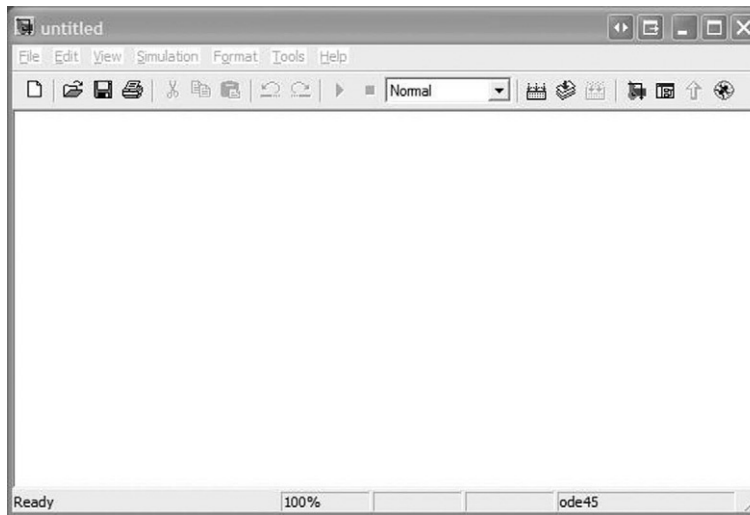


FIGURE A.9 SIMULINK workspace used to create the model.

another window where parameters are specified. The block size can be changed by selecting the block with the mouse pointer, and then, while the left mouse button is held down on the handle, drag the handle to its new size. To remove a block, simply select and delete it. For convenience, one can use the copy and paste commands to reproduce a block.

Blocks are connected together with line segments. To connect two blocks, move the mouse pointer to the angle bracket ($>$) on the output of one block, hold the left mouse button down, and then drag the pointer to the input $<$ of another block. If the line segment is drawn incorrectly, use the mouse to select, and then delete it. The line segment can be manipulated by selecting it, and while the left mouse button is held down, drag the line segment to its desired shape. One can also break into a line segment by holding the mouse pointer over the line segment with the “CTRL” key pressed down, clicking the left mouse button, and dragging the pointer to the desired location.

Shown in Figure A.11 (Top) is the workspace that consists of a sine wave, integrator, and scope, connected together as the output of the sine wave to the input of the integrator to the output of the integrator to the input of the scope. Double-clicking on the sine wave reveals its properties as shown in Figure A.11 (Middle). Double-clicking on the scope block results in a graph shown in Figure A.11 (Bottom). The scope is an excellent debugging tool to verify all simulation outputs are correct, but other graphing capabilities in MATLAB and EXCEL provide far superior plots for professional reports.

Once a SIMULINK model is defined, a simulation of the model is run whereby all outputs and inputs are computed from the simulation start time to the end time for each time period. Before running the simulation, the simulation parameters need to be specified by clicking “Simulation” on the menu and then selecting the pull down “Simulation Parameters”. Shown in Figure A.12 is the “Simulation Parameter” window with default parameters. The default values are usually acceptable for many simple applications. Obviously, the “Stop time” should be set to be at least 5 time constants (dominant time constant).

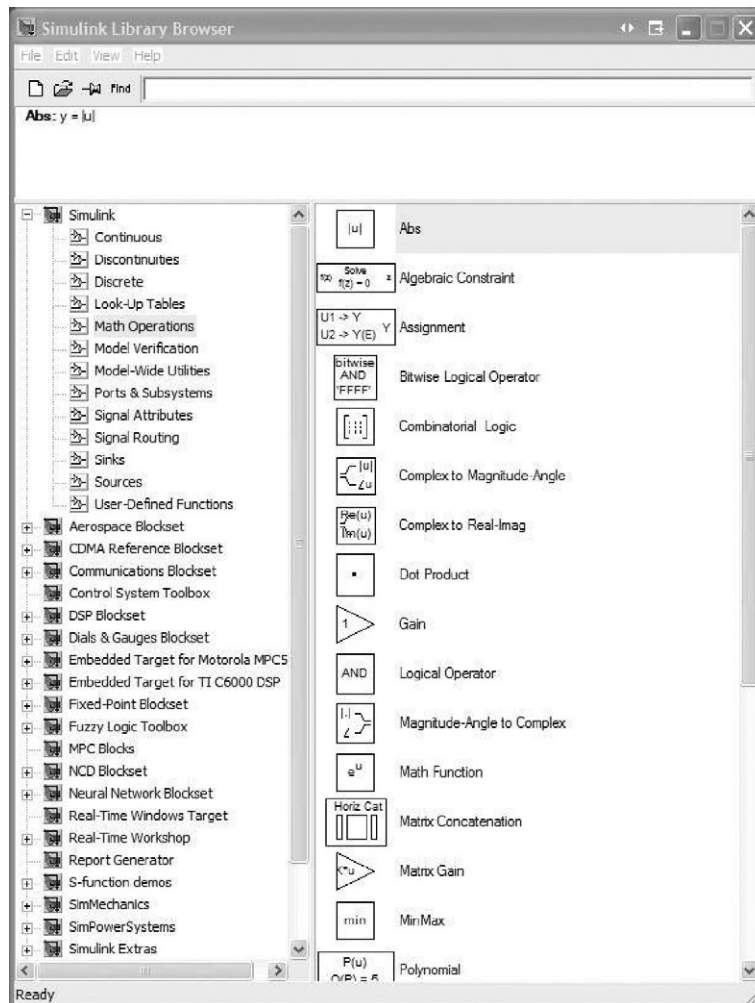


FIGURE A.10 The “Math Operations” library.

The start time can be set to a value appropriate for the simulation. The default simulation algorithm (ode45, the Dormand-Prince formula) is usually adequate for most applications. This algorithm is the best first choice as it involves calculating the output value at the next state based only on the previous output value.

Simulations can be run with either a fixed-step solver or a variable-step solver. The fixed-step solver calculates the model at equally spaced time steps, beginning at the start time. The step size must be small enough so that the approximations are valid, but not too small so that the program takes too long to run. A good rule of thumb is to start at

$$\text{step size} = \frac{\text{Total simulation time}}{500 \text{ to } 5000}$$

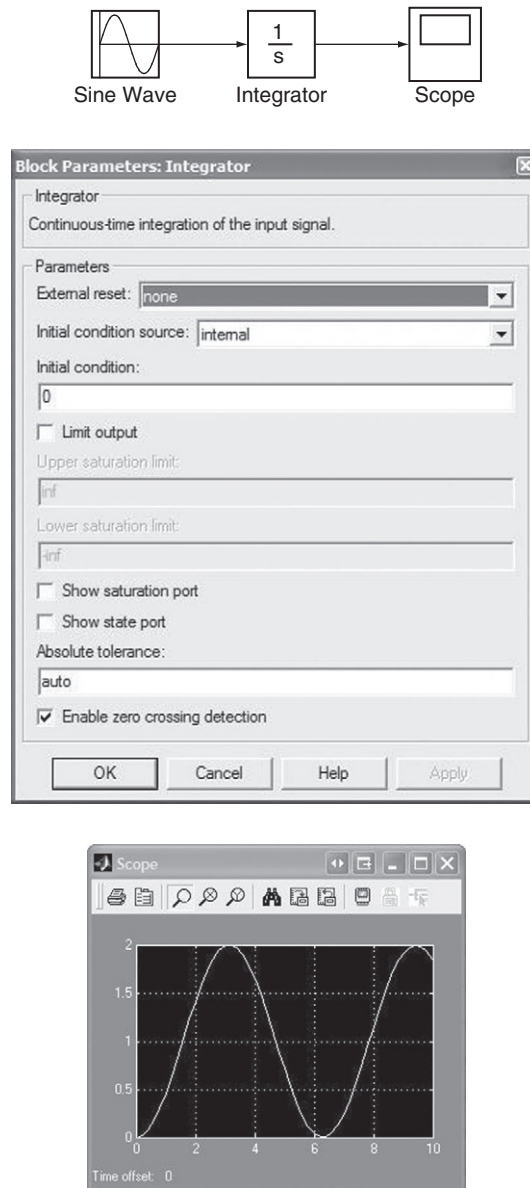


FIGURE A.11 (Top) SIMULINK blocks. (Middle) Block parameters for the integrator block. (Bottom) The output of the scope block.

and then to reduce the step size by 10. If the simulation output does not change, retain the step size, otherwise continue to reduce the step size by 10 until the output stabilizes. The variable-step size adjusts the step size, decreasing the size when the outputs are changing rapidly and increasing the step size when the outputs are changing slowly. The

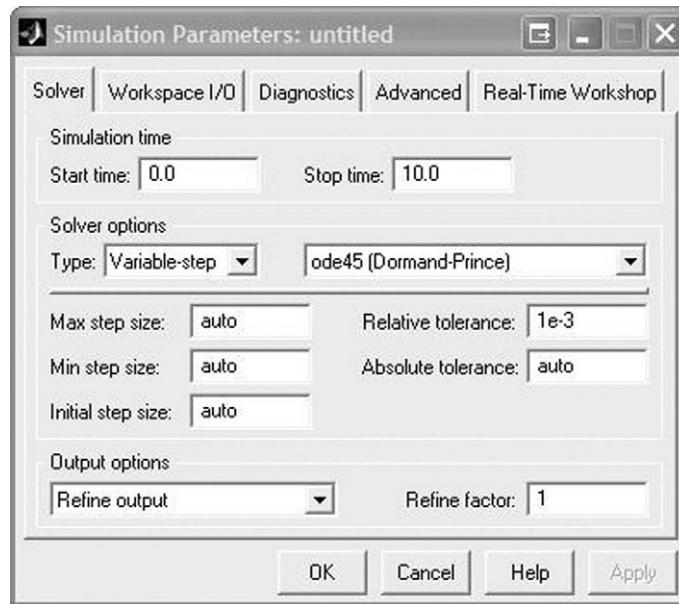


FIGURE A.12 Simulation Parameters window.

variable-step size simulation often saves simulation calculation time, resulting in a faster simulation. To run the simulation, select the command “Start” from the “Simulation” menu or the run command icon.

After creating a SIMULINK model, use “Save” or “Save As” from the “File” menu to save the model. Many simulation models have a number of parameters, some of which are changed from simulation to simulation. For convenience, it makes sense to define the parameters in MATLAB (i.e., $K=5$, $M=3$, etc.), with the parameter values saved in an m-file. The m-file can then be opened in MATLAB and these parameters can then be used in the SIMULINK model. To print the SIMULINK model, select the “Print” command from the “File” menu, or select the “Copy Model to Clipboard” command from the “Edit” menu and paste it into another application like Microsoft Word.

EXAMPLE PROBLEM A.8

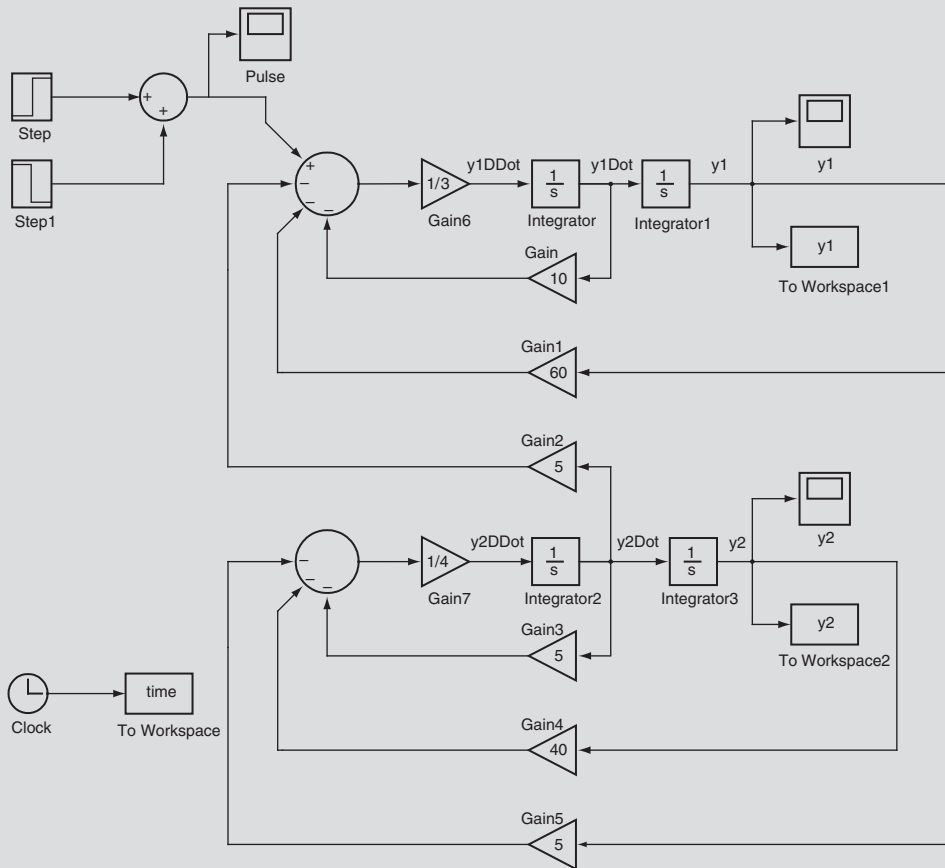
Simulate the response of the system described in Example A.7 with zero initial conditions if the input $f(t)$ is a pulse with magnitude 120 and duration 10 applied at $t = 0$.


Solution

Whenever we create a SIMULINK simulation, it is best to begin by drawing the block diagram for the system. In Example A.7, we already created the block diagram where the highest derivative was first solved in each equation and all of the other terms were placed on the right-hand side

of the equation. The block diagram was then constructed using a summer for the equal sign, the integrator to remove the derivative terms, and gains for the feedback elements. The SIMULINK model is essentially a reproduction of the block diagram.

To create the SIMULINK model, we open a SIMULINK session, open a SIMULINK workspace, and then open the appropriate library to begin dragging and dropping the icons to create the model. The final SIMULINK model is shown in the following figure. Details on how to create the model are provided in the remainder of the example.





To begin drawing the model, first drag the summer, which is located in the “Math Operations” library. The summer icon looks like . This block can be changed to a rectangle from the default circle by double-clicking the block, which opens the “Block Parameters Sum” window. Also, the number of inputs can be increased by adding additional “pluses” or “minuses” in the

Continued

list of signs. For the first equation, we have four elements connected to the summer (a positive for the input and three negative feedback elements), so the summer is modified by removing one positive sign and adding three negative signs, giving



The input is a pulse with magnitude 120 and duration of 10. To create a pulse, we use two step inputs from the Library “Sources”. The step input block looks like  to enter the magnitude and duration, double-click on the block to open the “Block Parameters Step” and enter for the first block 0 for “Step Time”, 10 for the “Initial value”, and 0 for the “Final value”. The second step function uses the values 120 for the “Step Time”, 0 for the “Initial value”, and -120 for the “Final value”. Another summer with line segments from the output of the step to the input of the summer is used to connect the two inputs. The output of the input summer is connected to the “+” input of the first summer.

After the summer, a gain of $1/3$ is needed. The gain block looks like , which is found in the “Math Operations” library. After dragging it from the library, the magnitude is set by double-clicking and entering a value of $1/3$. The number inside the gain block is the value of the gain if it can fit, otherwise “-k-” is used. A line segment is used to connect the summer to the gain.

Next, two integrators are dragged across to the workspace from the “Continuous” library. The integrator has the symbol $\frac{1}{s}$ in it that comes from the Laplace transform representation of integration. While not necessary in this example, integrators sometimes have nonzero initial conditions. To enter the initial condition for an integrator, double-click on the integrator block, which opens the “Block Parameters Integrator” window, where the initial condition can be entered. By default, the initial conditions for the integrator are zero. The output of the summer is connected with a line segment to the input of the first integrator, the output of the first integrator is connected to the input of the second integrator, and the output of the second integrator is connected to the input of the third integrator. It is generally good practice to label the output of each integrator. A label can be entered by clicking anywhere in the open workspace. A convenient label for \ddot{y}_1 is “y1DDot” (borrowed from the D-operator), \dot{y}_1 is “y1Dot”, and so on.

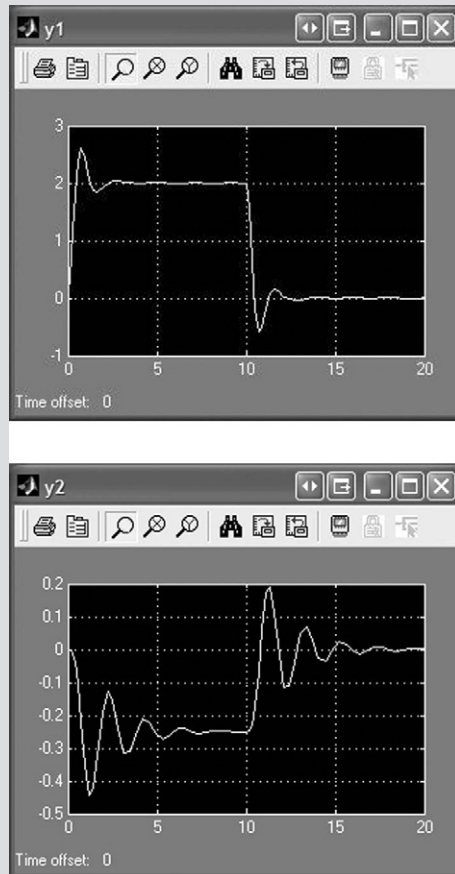
Three gain blocks are then dragged across to the workspace from the “Math Operations” library. To rotate the gain block so that it is in the correct orientation, select it and press the “CTRL and R” keys together twice for a 180° counterclockwise rotation (each click rotates 90°). The value for each gain block is entered by double-clicking and entering 10, 60, and 5. Line segments are used to connect the output of the integrators to the gain block by moving the mouse pointer over the integrator output line segment, pressing “CTRL” and left-clicking the mouse (to break into the line), and then moving the mouse pointer to the input of the gain block. The output of the gain blocks are then connected to the summer. Note that the term \dot{y}_2 is not connected to the gain block until the second equation is drawn.

The same steps are used to create the SIMULINK model for the second equation. Notice that the input for the Gain2 block is \dot{y}_2 , and the input for Gain5 is y_1 .

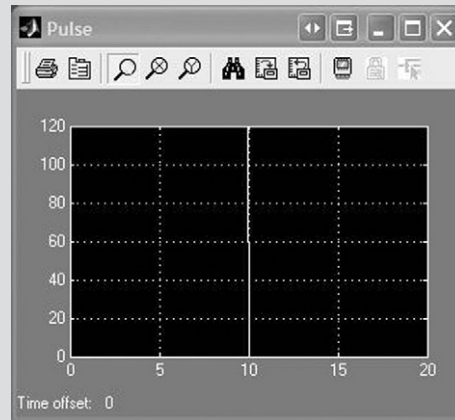
To view the output, two scopes are used from the “Sinks” library. For clarity the labels on each scope are renamed to “y1” and “y2” by double-clicking on the labels below the scope and making

the change. Double-clicking on the scope icon opens a new window that contains a graph for the output variable. A scope is also drawn for the pulse input as good practice to ensure that the input is correctly specified. The binocular icon on the menu of the scope graph auto scales the graph.

The “Clock” block from the “Sources” library is used to output the simulation time at each simulation step. To move the simulation data from SIMULINK to the MATLAB workspace for further analysis and plotting, the “To Workspace” block is used from the “Sinks” library. Three “To Workspace” blocks are used in the example to output data for the variables y_1 , y_2 , and $Time$. The “To Workspace” writes the data to an array specified by the block’s “Variable name”. By default, the “To Workspace” uses the “Variable name” “simout” for the first, “simout2” for the second, and so on. These names are easily renamed by double-clicking on the block and entering meaningful names, such as y_1 , y_2 , and $Time$ in this example. While the block is open, the “Save format” should be changed to “Array” and the “Sample Time” to a number that provides enough data points. In this example, the simulation time is 20, so the sample time is set at .2 to provide 100 data points for each variable in the MATLAB workspace. Once in the workspace, one can manipulate or plot the variables, i.e., using the MATLAB command “plot(Time, y_1)”.



Continued



The final step is to open the “Simulation Parameters” to change the simulation “Stop time” to 20 and click the run button. The preceding figure displays the output of the three scopes.

In some simulations, a model’s performance is evaluated by changing parameter values. For the model in this example, the block parameters would need to be changed individually. For a model with hundreds of parameters, this would entail a considerable effort. It is far easier to enter the parameter values in MATLAB and use the parameter names in the blocks rather than values. For instance, enter the following in MATLAB

```
>> PulseGain = 120;
>> PulseDuration = 10;
>> Gain = 10;
>> Gain1 = 60;
>> Gain2 = 5;
>> Gain3 = 5;
>> Gain4 = 40;
>> Gain5 = 5;
>> Gain6 = 1/3;
>> Gain7 = 1/4;
```

Then in each gain block, use the parameter name rather than the value. Any changes in the MATLAB values are automatically updated in SIMULINK. Defining initial conditions for integrator blocks can also be done in MATLAB. To avoid entering the parameter values each time MATLAB is opened, store the values in an m-file.

It is often convenient to use subsystems to collect blocks together when drawing a SIMULINK model to make a model more readable. In the previous example, for instance, the pulse input is easily drawn as a subsystem by selecting the blocks by collecting them in a bounding box with the mouse pointer (i.e., use the mouse to draw a box around the elements), right-clicking the mouse and then selecting “Create subsystem”. The inputs and the outputs to the subsystem are those of the elements in the selected blocks. Shown in [Figure A.13](#) is part of the model from the previous example

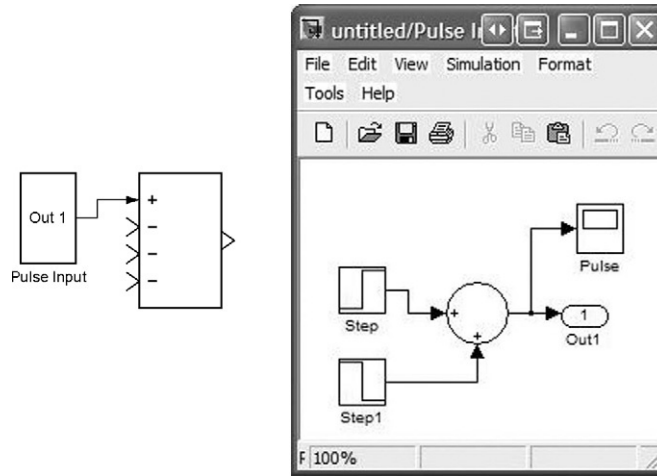


FIGURE A.13 (Left) A subsystem for the pulse input from the previous example. (Right) Double-clicking the subsystem block opens a new window that displays the blocks within the subsystem.

using a subsystem for the pulse. Double-clicking the subsystem block opens a new window with the blocks in the subsystem with the output from the subsystem indicated by the Out1 block.

Two blocks that are useful in simulations are the “Math Function” and “Fcn”. The “Math Function” from the “Math Operations” library is convenient for entering mathematical functions such as the exponential, logarithm, and power functions from a drop down list. For example, when the input variable name is “u” and the function “exp”, the output is “exp (u)”. The label within the block is the math function. The “Fcn” from the “User-Defined Functions” library is convenient for entering general expressions. Once again, the input variable name is “u”. Any arithmetic operator (+ − * /) and mathematical function (cos, sin, tan, exp, power, ...) can be used in the block, as well as any parameter from MATLAB. For instance, in writing the potassium conductance for a membrane, the expression $\bar{g}_k n^4$ is needed, where \bar{g}_k is a constant and n is a voltage and time dependent rate constant. In SIMULINK, this expression is written using the “Fcn” block with the parameter “Gkbar*u^4” where “Gkbar” is defined in MATLAB and “u” the input to the block representing n .

In modeling systems, some parameters may change based on threshold conditions. For example, in the saccadic eye movement system, the input to the agonist muscle, F_{ag} , is a low-pass filtered plus-step waveform given by the following equation.

$$\dot{E}_{ag} = \frac{N_{ag} - F_{ag}}{\tau_{ag}}$$

where N_{ag} is the neural control input (pulse-step waveform), and

$$\tau_{ag} = \tau_{ac} (u(t) - u(t - t_1)) + \tau_{de} u(t - t_1)$$

where τ_{ag} is the time-varying time constant made up of τ_{ac} and τ_{de} , the activation and deactivation time constants. To simulate this subsystem we use the “Switch” block in the “Signal

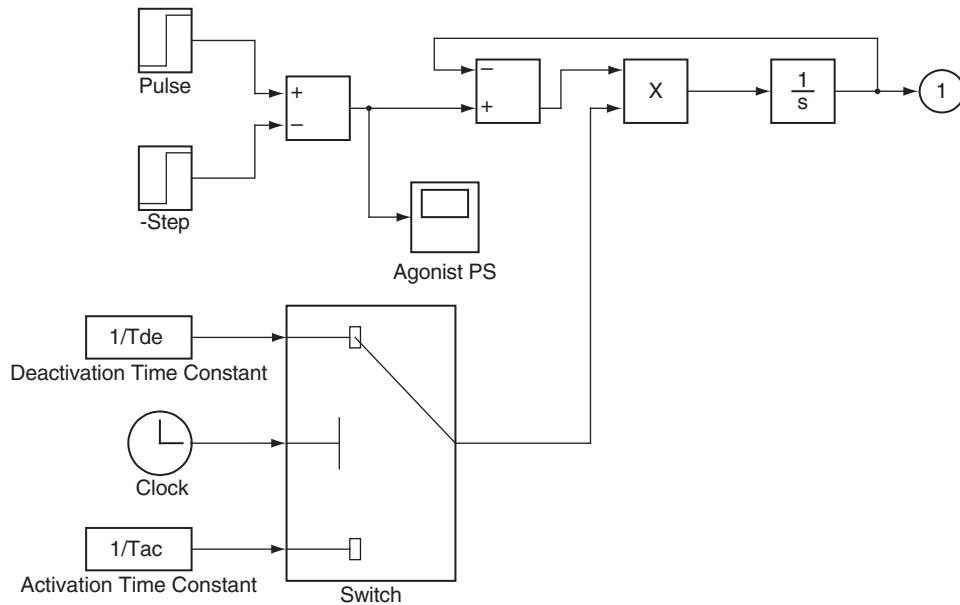


FIGURE A.14 A low-pass filter with a time-varying time constant.

Routing” library as shown in Figure A.14. The “clock” block is used to monitor the simulation time. At the start of the simulation, the time constant τ_{ac} is used. When the simulation time is greater than t_1 , threshold is reached and the switch changes to the time constant τ_{de} . The remainder of the blocks in Figure A.13 solves for F_{ag} .

For information on the other operations in SIMULINK, the reader is encouraged to consult the help menu.