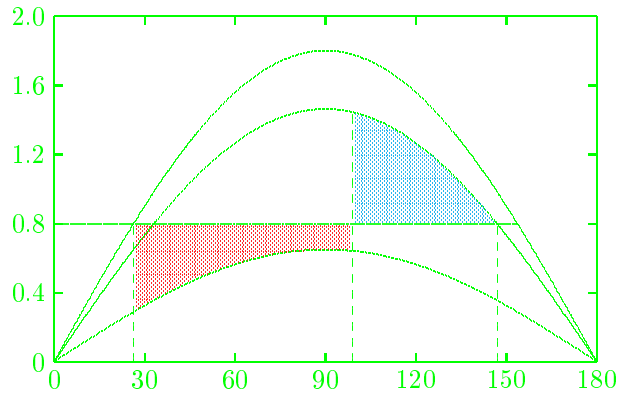


INTRODUCTION TO MATLAB

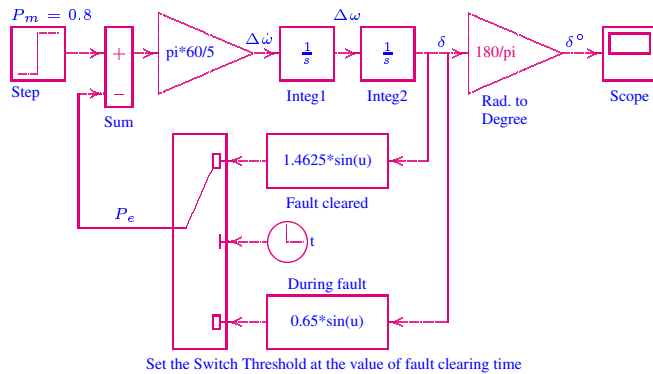
Chapter 1 [Go to Chapter 2](#)

$P_m = 0.8$; $E = 1.17$; $V = 1.0$
 $X_1 = 0.65$; $X_2 = 1.8$; $X_3 = 0.8$
`eacfault(Pm, E, V, X1, X2, X3)`

and



SIMULINK



Hadi Saadat

INTRODUCTION TO MATLAB AND SIMULINK

version v3.0

Hadi Saadat

Professor of Electrical Engineering
Milwaukee School of Engineering
Milwaukee, Wisconsin

Copyright ©2000, Hadi Saadat. All rights reserved.

Introduction to MATLAB and SIMULINK, and the accompanying m-files are distributed for the MSOE students. It may not be altered in any way, or be used as part of other documents.

CONTENTS

1	INTRODUCTION TO <i>MATLAB</i>	1
1.1	INSTALLING THE TEXT TOOLBOX	2
1.2	RUNNING <i>MATLAB</i>	2
1.3	VARIABLES	4
1.4	OUTPUT FORMAT	4
1.5	CHARACTER STRING	7
1.6	VECTOR OPERATIONS	7
1.7	ELEMENTARY MATRIX OPERATIONS	10
1.7.1	UTILITY MATRICES	12
1.7.2	EIGENVALUES	12
1.8	COMPLEX NUMBERS	13
1.9	POLYNOMIAL ROOTS AND CHARACTERISTIC POLYNOMIAL	15
1.9.1	PRODUCT AND DIVISION OF POLYNOMIALS	16
1.9.2	POLYNOMIAL CURVE FITTING	17
1.9.3	POLYNOMIAL EVALUATION	18
1.9.4	PARTIAL-FRACTION EXPANSION	18
1.10	GRAPHICS	19
1.11	GRAPHICS HARD COPY	21
1.12	THREE-DIMENSIONAL PLOTS	27
1.13	HANDLE GRAPHICS	30
1.14	LOOPS AND LOGICAL STATEMENTS	30
1.15	SIMULATION DIAGRAM	36
1.16	INTRODUCTION TO SIMULINK	38
1.16.1	SIMULATION PARAMETERS AND SOLVER	39
1.16.2	THE SIMULATION PARAMETERS DIALOG BOX	40
1.16.3	BLOCK DIAGRAM CONSTRUCTION	41
1.16.4	USING THE TO WORKSPACE BLOCK	47
1.16.5	LINEAR STATE-SPACE MODEL FROM SIMULINK DIAGRAM	47

ii CONTENTS

1.16.6 SUBSYSTEMS AND MASKING 49

CHAPTER 1

INTRODUCTION TO *MATLAB*

MATLAB, developed by Math Works Inc., is a software package for high performance numerical computation and visualization. The combination of analysis capabilities, flexibility, reliability, and powerful graphics makes *MATLAB* the premier software package for electrical engineers.

MATLAB provides an interactive environment with hundreds of reliable and accurate built-in mathematical functions. These functions provide solutions to a broad range of mathematical problems including matrix algebra, complex arithmetic, linear systems, differential equations, signal processing, optimization, nonlinear systems, and many other types of scientific computations. The most important feature of *MATLAB* is its programming capability, which is very easy to learn and to use, and which allows user-developed functions. It also allows access to Fortran algorithms and C codes by means of external interfaces. There are several optional toolboxes written for special applications such as signal processing, control systems design, system identification, statistics, neural networks, fuzzy logic, symbolic computations, and others. *MATLAB* has been enhanced by the very powerful *SIMULINK* program. *SIMULINK* is a graphical mouse-driven program for the simulation of dynamic systems. *SIMULINK* enables students to simulate linear, as well as nonlinear, systems easily and efficiently.

The following section describes the use of *MATLAB* and is designed to give a quick familiarization with some of the commands and capabilities of *MATLAB*. For a description of all other commands, *MATLAB* functions, and many other useful features, the reader is referred to the *MATLAB User's Guide*.

1.1 INSTALLING THE TEXT TOOLBOX

The software diskette included with the book contains all the developed functions and chapter examples. The file names for chapter examples begin with the letters **ch**. For example, the M-file for Example 2.4 is **ch2ex04**. Create a subdirectory, such as **HS**, where the *MATLABR11* toolbox resides. Copy all the files on the diskette to the subdirectory *MATLABR11 \TOOLBOX\HS*.

In the *MATLAB 5.3 Command Window* open the Path Browser by selecting **Set Path** from the **File** menu. Press to open the **Add to Path** window. Open the **toolbox** folder and double-click on the **HS** folder. Choose **Add to Back** option and click on **Save Settings** to save the new path permanently.

1.2 RUNNING MATLAB

MATLAB supports almost every computational platform. *MATLAB* for *WINDOWS* is started by clicking on the *MATLAB* icon. The **Command window** is launched, and after some messages such as intro, demo, help help, info, and others, the prompt “>>” is displayed. The program is in an interactive command mode. Typing **who** or **whos** displays a list of variable names currently in memory. Also, the **dir** command lists all the files on the default directory. *MATLAB* has an on-line help facility, and its use is highly recommended. The command **help** provides a list of files, built-in functions and operators for which on-line help is available. The command

```
help function name
```

will give information on the specified function as to its purpose and use. The command

```
help help
```

will give information as to how to use the on-line help.

MATLAB has a demonstration program that shows many of its features. The command **demo** brings up a menu of the available demonstrations. This will provide a presentation of the most important *MATLAB* facilities. Follow the instructions on the screen – it is worth trying.

MATLAB 5.3 includes a Help Desk facility that provides access to on line help topics, documentation, getting started with *MATLAB*, online reference materials, *MATLAB* functions, real-time Workshop, and several toolboxes. The online documentation is available in HTML, via either Netscape Navigator or Microsoft Internet Explorer. The command **helpdesk** launches the Help Desk, or you can use the **Help** menu to bring up the Help Desk.

If an expression with correct syntax is entered at the prompt in the Command window, it is processed immediately and the result is displayed on the screen. If an expression requires more than one line, the last character of the previous line must

contain three dots "...". Characters following the percent sign are ignored. The (%) may be used anywhere in a program to add clarifying comments. This is especially helpful when creating a program. The command **clear** erases all variables in the Command window.

MATLAB is also capable of executing sequences of commands that are stored in files, known as script files or *M-files*. Clicking on **File, Open M-file**, opens the **Edit window**. A program can be written and saved in ASCII format with a filename having extension .m in the directory where *MATLAB* runs. To run the program, click on the Command window and type the filename without the .m extension at the *MATLAB* command "»". You can view the text Edit window simultaneously with the Command window. That is, you can use the two windows to edit and debug a script file repeatedly and run it in the Command window without ever quitting *MATLAB*.

In addition to the Command window and Edit window are the **Graphic windows** or **Figure windows** with grey (default) background. The plots created by the graphic commands appear in these windows.

Another type of M-file is a *function file*. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. In contrast to the script file, a *function file* has a name following the word "function" at the beginning of the file. The filename must be the same as the "function" name. The first line of a function file must begin with the function statement having the following syntax

```
function [output arguments] = function name (input arguments)
```

The output argument(s) are variables returned. A function need not return a value. The input arguments are variables passed to the function. Variables generated in function files are local to the function. The use of **global** variables make defined variables common and accessible between the main script file and other function files. For example, the statement **global R S T** declares the variables *R*, *S*, and *T* to be global without the need for passing the variables through the input list. This statement goes before any executable statement in the script and function files that need to access the values of the global variables.

Normally, while an M-file is executing, the commands of the file are not displayed on the screen. The command **echo** allows M-files to be viewed as they execute. **echo off** turns off the echoing of all script files. Typing **what** lists M-files and Mat-files in the default directory.

MATLAB follows conventional Windows procedure. Information from the command screen can be printed by highlighting the desired text with the mouse and then choosing the **print Selected ...** from the **File** menu. If no text is highlighted the entire Command window is printed. Similarly, selecting **print** from the Figure window sends the selected graph to the printer. For a complete list and help on general purpose commands, type **help general**.

1.3 VARIABLES

Expressions typed without a variable name are evaluated by *MATLAB*, and the result is stored and displayed by a variable called **ans**. The result of an expression can be assigned to a variable name for further use. Variable names can have as many as 19 characters (including letters and numbers). However, the first character of a variable name must be a letter. *MATLAB* is case-sensitive. Lower and uppercase letters represent two different variables. The command **casesen** makes *MATLAB* insensitive to the case. Variables in script files are global. The expressions are composed of operators and any of the available functions. For example, if the following expression is typed

```
x = exp(-0.2696*.2)*sin(2*pi*0.2)/(0.01*sqrt(3)*log(18))
```

the result is displayed on the screen as

```
x =
    18.0001
```

and is assigned to `x`. If a variable name is not used, the result is assigned to the variable **ans**. For example, typing the expression

```
250/sin(pi/6)
```

results in

```
ans =
    500.0000
```

If the last character of a statement is a semicolon (;), the expression is executed, but the result is not displayed. However, the result is displayed upon entering the variable name. The command **disp** may be used to display a variable without printing its name. For example, **disp(x)** displays the value of the variable without printing its name. If `x` contains a text string, the string is displayed.

1.4 OUTPUT FORMAT

While all computations in *MATLAB* are done in double precision, the default format prints results with five significant digits. The format of the displayed output can be controlled by the following commands.

MATLAB Command	Display
format	Default. Same as format short
format short	Scaled fixed point format with 5 digits
format long	Scaled fixed point format with 15 digits
format short e	Floating point format with 5 digits
format long e	Floating point format with 15 digits
format short g	Best of fixed or floating point with 5 digits
format long g	Best of fixed or floating point with 15 digits
format hex	Hexadecimal format
format +	The symbols +, - and blank are printed for positive, negative, and zero elements
format bank	Fixed format for dollars and cents
format rat	Approximation by ratio of small integers
format compact	Suppress extra line feeds
format loose	Puts the extra line feeds back in

For more flexibility in the output format, the command **fprintf** displays the result with a desired format on the screen or to a specified filename. The general form of this command is the following.

```
fprintf{fstr, A,...}
```

writes the real elements of the variable or matrix A, \dots according to the specifications in the string argument of `fstr`. This string can contain format characters like *ANCI C* with certain exceptions and extensions. **fprintf** is "vectorized" for the case when A is nonscalar. The format string is recycled through the elements of A (columnwise) until all the elements are used up. It is then recycled in a similar manner through any additional matrix arguments. The characters used in the format string of the commands **fprintf** are listed in the table below.

Format codes		Control characters	
%e	scientific format, lower case e	\n	new line
%E	scientific format, upper case E	\r	beginning of the line
%f	decimal format	\b	back space
%s	string	\t	tab
%u	integer	\g	new page
%i	follows the type	//	apostrophe
%x	hexadecimal, lower case	\\	back slash
%X	hexadecimal, upper case	\a	bell

A simple example of the **fprintf** is

```
fprintf('Area = %7.3f Square meters \n', pi*4.5^2)
```

The results is

```
Area = 63.617 Square meters
```

The `%7.3f` prints a floating point number seven characters wide, with three digits after the decimal point. The sequence `\n` advances the output to the left margin on the next line.

The following command displays a formatted table of the natural logarithmic for numbers 10, 20, 40, 60, and 80

```
x = [10; 20; 40; 60; 80];
y = [x, log(x)];
fprintf('\n Number    Natural log\n')
fprintf('%4i \t %8.3f\n',y')
```

The result is

Number	Natural log
10	2.303
20	2.996
40	3.689
60	4.094
80	4.382

An M-file can prompt for input from the keyboard. The command **input** causes the computer to request data from the keyboard. For example, the command

```
R = input('Enter radius in meter ')
```

displays the text string

```
Enter radius in meter
```

and waits for a number to be entered. If a number, say 4.5 is entered, it is assigned to variable `R` and displayed as

```
R =
4.5000
```

The command **keyboard** placed in an M-file will stop the execution of the file and permit the user to examine and change variables in the file. Pressing **ctrl-z** terminates the keyboard mode and returns to the invoking file. Another useful command is **diary A:filename**. This command creates a file on drive A, and all output displayed on the screen is sent to that file. **diary off** turns off the diary. The contents of this file can be edited and used for merging with a word processor file. Finally, the command **save filename** can be used to save the expressions on the screen to a file named *filename.mat*, and the statement **load filename** can be used to load the file *filename.mat*.

MATLAB has a useful collection of transcendental functions, such as exponential, logarithm, trigonometric, and hyperbolic functions. For a complete list and help on operators, type **help ops**, and for elementary math functions, type **help elfun**.

1.5 CHARACTER STRING

A sequence of characters in single quotes is called a *character string* or *text variable*.

```
c = 'Good'
```

results in

```
c = Good
```

A text variable can be augmented with more text variables, for example,

```
cs = [c, ' luck']
```

produces

```
cs =
    Good luck
```

1.6 VECTOR OPERATIONS

An n vector is a row or a column array of n numbers. In *MATLAB*, elements enclosed by brackets and separated by semicolons generate a column vector.

For example, the statement

```
X = [ 2; -4; 8]
```

results in

```
X =
     2
    -4
     8
```

If elements are separated by blanks or commas, a row vector is produced. Elements may be any expression. The statement

```
R = [tan(pi/4) sqrt(9) -5]
```

results in the output

```
R =
    1.0000    3.0000   -5.0000
```

The transpose of a column vector results in a row vector, and vice versa. For example

```
Y=R'
```

will produce

```
Y =
    1.0000
    3.0000
   -5.0000
```

MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element-by-element. The period character (.) distinguishes the array operations from the matrix operations. However, since the matrix and array operations are the same for addition and subtraction, the character pairs .+ and .- are not used.

Vectors of the same size can be added or subtracted, where addition is performed componentwise. However, for multiplication, specific rules must be followed in order to obtain the correct resulting values. The operation of multiplying a vector X with a scalar k (scalar multiplication) is performed componentwise. For example $P = 5 * R$ produces the output

```
P =
    5.0000   15.0000  -25.0000
```

The inner product or the *dot product* of two vectors X and Y denoted by $\langle X, Y \rangle$ is a scalar quantity defined by $\sum_{i=1}^n x_i y_i$. If X and Y are both column vectors defined above, the inner product is given by

```
S = X' * Y
```

and results in

```
S =
   -50
```

The operator (.* performs element-by-element operation. For example, for the previously defined vectors, X and Y , the statement

```
E = X .* Y
```

results in

```
E =
     2
    -12
    -40
```

The operator ./ performs element-by-element division. The two arrays must have the same size, unless one of them is a scalar. Array powers or element-by-element powers are denoted by (.^). The trigonometric functions, and other elementary mathematical functions such as **abs**, **sqrt**, **real**, and **log**, also operate element by element.

Various norms (measure of size) of a vector can be obtained. For example, the *Euclidean norm* is the square root of the inner product of the vector and itself. The command

```
N = norm(X)
```

produces the output

```
N =
    9.1652
```

The angle between two vectors X and Y is defined by $\cos \theta = \frac{\langle X, Y \rangle}{\|X\| \|Y\|}$. The statement

```
Theta = acos( X'*Y/(norm(X)*norm(Y)) )
```

results in the output

```
Theta =
    2.7444
```

where Theta is in radians.

The *zero vector*, also referred to as origin, is a vector with all components equal to zero. For example, to build a zero row vector of size 4, the following command

```
Z = zeros(1, 4)
```

results in

```
Z =
    0    0    0    0
```

The *one vector* is a vector with each component equal to one. To generate a one vector of size 4, use

```
I = ones(1, 4)
```

The result is

```
I =
    1    1    1    1
```

In *MATLAB*, the colon (`:`) can be used to generate a row vector. For example

```
x = 1:8
```

generates a row vector of integers from 1 to 8.

```
x =
    1    2    3    4    5    6    7    8
```

For increments other than unity, the following command

```
z = 0 : pi/3 : pi
```

results in

```
z =
    0000    1.0472    2.0944    3.1416
```

For negative increments

```
x = 5 : -1:1
```

results in

```
x =
     5     4     3     2     1
```

Alternatively, special vectors can be created, the command **linspace(x, y, n)** creates a vector with n elements that are spaced linearly between x and y . Similarly, the command **logspace(x, y, n)** creates a vector with n elements that are spaced in even logarithmic increments between 10^x and 10^y .

1.7 ELEMENTARY MATRIX OPERATIONS

In *MATLAB*, a matrix is created with a rectangular array of numbers surrounded by brackets. The elements in each row are separated by blanks or commas. A semicolon must be used to indicate the end of a row. Matrix elements can be any *MATLAB* expression. The statement

```
A = [ 6  1  2; -1  8  3;  2  4  9]
```

results in the output

```
A =
     6     1     2
    -1     8     3
     2     4     9
```

If a semicolon is not used, each row must be entered in a separate line as shown below.

```
A = [ 6  1  2
    -1  8  3
     2  4  9]
```

The entire row or column of a matrix can be addressed by means of the symbol (:). For example

```
r3 = A(3, :)
```

results in

```
r3 =
     2     4     9
```

Similarly, the statement $A(:, 2)$ addresses all elements of the second column in A .

Matrices of the same dimension can be added or subtracted. Two matrices, A and B , can be multiplied together to form the product AB if they are conformable. Two symbols are used for nonsingular matrix division. $A \setminus B$ is equivalent to $A^{-1}B$, and A/B is equivalent to AB^{-1} .

Example 1.1

For the matrix equation below, $AX = B$, determine the vector X .

$$\begin{bmatrix} 4 & -2 & -10 \\ 2 & 10 & -12 \\ -4 & -6 & 16 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -10 \\ 32 \\ -16 \end{bmatrix}$$

The following statements

```
A = [4 -2 -10; 2 10 -12; -4 -6 16];
B = [-10; 32; -16];
X = A\B
```

result in the output

```
X =
    2.0000
    4.0000
    1.0000
```

In addition to the built-in functions, numerous mathematical functions are available in the form of M-files. For the current list and their applications, see the *MATLAB User's Guide*.

Example 1.2

Use the `inv` function to determine the inverse of matrix A in Example 1.1 and then determine X . The following statements

```
A = [4 -2 -10; 2 10 -12; -4 -6 16];
B = [-10; 32; -16];
C = inv(A)
X = C*B
```

result in the output

```
C =
    2.2000    2.3000    3.1000
    0.4000    0.6000    0.7000
    0.7000    0.8000    1.1000
X =
    2.0000
    4.0000
    1.0000
```

Example 1.3

Use the `lu` factorization function to express the matrix A of Example 1.2 as the product of upper and lower triangular matrices, $A = LU$. Then find X from $X = U^{-1}L^{-1}B$. Typing

```
A = [ 4 -2 -10; 2 10 -12; -4 -6 16 ]
B = [-10; 32 -16];
[L,U] = lu(A)
```

results in

```
L =
    1.0000         0         0
    0.5000    1.0000         0
   -1.0000   -0.7273    1.0000

U =
    4.0000   -2.0000  -10.0000
         0   11.0000   -7.0000
         0         0    0.9091
```

Now entering

```
X = inv(U)*inv(L)*B
```

results in

```
X =
    2.0000
    4.0000
    1.0000
```

Dimensioning is automatic in *MATLAB*. You can find the dimensions and rank of an existing matrix with the **size** and **rank** statements. For vectors, use the command **length**.

1.7.1 UTILITY MATRICES

There are many special utility matrices which are useful for matrix operations. A few examples are

eye(m, n)	Generates an m -by- n identity matrix.
zeros(m, n)	Generates an m -by- n matrix of zeros.
ones(m, n)	Generates an m -by- n matrix of ones.
diag(x)	Produces a diagonal matrix with the elements of x on the diagonal line.

For a complete list and help on elementary matrices and matrix manipulation, type **help elmat**. There are many other special built-in matrices. For a complete list and help on specialized matrices, type **help specmat**.

1.7.2 EIGENVALUES

If A is an n -by- n matrix, the n numbers λ that satisfy $Ax = \lambda x$ are the eigenvalues of A . They are found using **eig(A)**, which returns the eigenvalues in a column vector.

Eigenvalues and eigenvectors can be obtained with a double assignment statement $[X, D] = \text{eig}(A)$. The diagonal elements of D are the eigenvalues and the columns of X are the corresponding eigenvectors such that $AX = XD$.

Example 1.4

Find the eigenvalues and the eigenvectors of the matrix A given by

$$A = \begin{bmatrix} 0 & 1 & -1 \\ -6 & -11 & 6 \\ -6 & -11 & 5 \end{bmatrix}$$

$$A = [0 \ 1 \ -1; \ -6 \ -11 \ 6; \ -6 \ -11 \ 5];$$

$$[X,D] = \text{eig}(A)$$

The eigenvalues and the eigenvectors are obtained as follows

$$X = \begin{matrix} & -0.7071 & 0.2182 & -0.0921 \\ & 0.0000 & 0.4364 & -0.5523 \\ & -0.7071 & 0.8729 & -0.8285 \end{matrix}$$

$$D = \begin{matrix} & -1 & 0 & 0 \\ & 0 & -2 & 0 \\ & 0 & 0 & -3 \end{matrix}$$

1.8 COMPLEX NUMBERS

All the *MATLAB* arithmetic operators are available for complex operations. The imaginary unit $\sqrt{-1}$ is predefined by two variables i and j . In a program, if other values are assigned to i and j , they must be redefined as imaginary units, or other characters can be defined for the imaginary unit.

$$j = \text{sqrt}(-1) \quad \text{or} \quad i = \text{sqrt}(-1)$$

Once the complex unit has been defined, complex numbers can be generated.

Example 1.5

Evaluate the following function $V = Zc \cosh g + \sinh g/Zc$, where $Zc = 200 + j300$ and $g = 0.02 + j1.5$

$$i = \text{sqrt}(-1); Zc = 200 + 300*i; g = 0.02 + 1.5*i;$$

$$v = Zc * \cosh(g) + \sinh(g)/Zc$$

results in the output

$$v = 8.1672 + 25.2172i$$

It is important to note that, when complex numbers are entered as matrix elements within brackets, we avoid any blank spaces. If spaces are provided around the complex number sign, it represents two separate numbers.

Example 1.6

In the circuit shown in Figure 1.1, determine the node voltages V_1 and V_2 and the power delivered by each source.

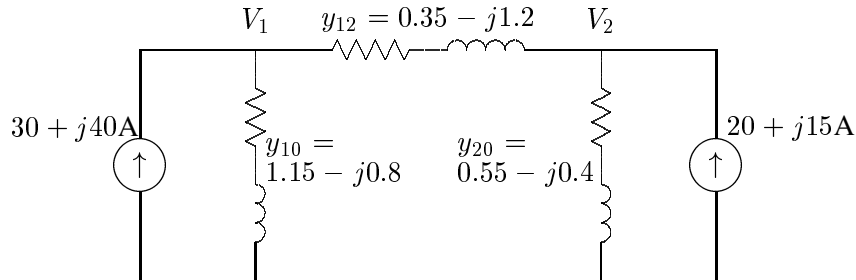


FIGURE 1.1
Circuit for Example 1.6.

Kirchhoff's current law results in the following matrix node equation.

$$\begin{bmatrix} 1.5 - j2.0 & -.35 + j1.2 \\ -.35 + j1.2 & 0.9 - j1.6 \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} 30 + j40 \\ 20 + j15 \end{bmatrix}$$

and the complex power of each source is given by $S = VI^*$. The following program is written to yield solutions to V_1 , V_2 and S using *MATLAB*.

```

j=sqrt(-1)                                % Defining j
I=[30+j*40; 20+j*15]                       % Column of node current phasors
Y=[1.5-j*2   -.35+j*1.2; -.35+j*1.2   .9-j*1.6]
                                             % Complex admittance matrix Y
disp('The solution is') V=inv(Y)*I          % Node voltage solution
S=V.*conj(I)                                % complex power at nodes

```

result in

The solution is

$$V = \begin{bmatrix} 3.5902 + 35.0928i \\ 6.0155 + 36.2212i \end{bmatrix}$$

$$S = \begin{bmatrix} 1511.4 + 909.2i \\ 663.6 + 634.2i \end{bmatrix}$$

In *MATLAB*, the conversion between polar and rectangular forms makes use of the following functions:

Operation	Description
$z = a + bi$ or $z = a + j * b$	Rectangular form
<code>real(z)</code>	Returns real part of z
<code>imag(z)</code>	Returns imaginary part of z
<code>abs(z)</code>	Absolute value of z
<code>angle(z)</code>	Phase angle of z
<code>conj(z)</code>	Conjugate of z
$z = M * exp(j * \theta)$	converts $M \angle \theta$ to rectangular form

The prime (') transposes a real matrix; but for complex matrices, the symbol (.'') must be used to find the transpose.

1.9 POLYNOMIAL ROOTS AND CHARACTERISTIC POLYNOMIAL

If p is a row vector containing the coefficients of a polynomial, **roots(p)** returns a column vector whose elements are the roots of the polynomial. If r is a column vector containing the roots of a polynomial, **poly(r)** returns a row vector whose elements are the coefficients of the polynomial.

Example 1.7

Find the roots of the following polynomial.

$$s^6 + 9s^5 + 31.25s^4 + 61.25s^3 + 67.75s^2 + 14.75s + 15$$

The polynomial coefficients are entered in a row vector in descending powers. The roots are found using **roots**.

$$p = [1 \quad 9 \quad 31.25 \quad 61.25 \quad 67.75 \quad 14.75 \quad 15]$$

$$r = \text{roots}(p)$$

The polynomial roots are obtained in column vector

$$r =$$

$$\begin{aligned} & -4.0000 \\ & -3.0000 \\ & -1.0000 + 2.0000i \\ & -1.0000 - 2.0000i \\ & 0.0000 + 0.5000i \\ & 0.0000 - 0.5000i \end{aligned}$$

Example 1.8

The roots of a polynomial are $-1, -2, -3 \pm j4$. Determine the polynomial equation.

Complex numbers may be entered using function i or j . The roots are then entered in a column vector. The polynomial equation is obtained using **poly** as follows

```
i = sqrt(-1)
r = [-1 -2 -3+4*i -3-4*i ]
p = poly(r)
```

The coefficients of the polynomial equation are obtained in a row vector.

```
p =
    1    9   45   87   50
```

Therefore, the polynomial equation is

$$s^4 + 9s^3 + 45s^2 + 87s + 50 = 0$$

Example 1.9

Determine the roots of the characteristic equation of the following matrix.

$$A = \begin{bmatrix} 0 & 1 & -1 \\ -6 & -11 & 6 \\ -6 & -11 & 5 \end{bmatrix}$$

The characteristic equation of the matrix is found by **poly**, and the roots of this equation are found by **roots**.

```
A = [ 0 1 -1; -6 -11 6; -6 -11 5];
p = poly(A)
r = roots(p)
```

The result is as follows

```
p =
    1.0000    6.0000   11.0000    6.0000
r =
   -3.0000
   -2.0000
   -1.0000
```

The roots of the characteristic equation are the same as the eigenvalues of matrix *A*. Thus, in place of the **poly** and **roots** function, we may use

```
r = eig(A)
```

1.9.1 PRODUCT AND DIVISION OF POLYNOMIALS

The product of polynomials is the convolution of the coefficients. The division of polynomials is obtained by using the deconvolution command.

Example 1.10

(a) Given $A = s^2 + 7s + 12$, and $B = s^2 + 9$, find $C = AB$.

(b) Given $Z = s^4 + 9s^3 + 37s^2 + 81s + 52$, and $Y = s^2 + 4s + 13$, find $X = \frac{Z}{Y}$.

The commands

```
A = [1 7 12]; B = [1 0 9];
C = conv(A, B)
Z = [1 9 37 81 52]; Y = [1 4 13];
[X, r] = deconv(Z, Y)
```

result in

```
C =
    1    7   21   63  108
X =
    1    5    4
r =
    0    0    0
```

1.9.2 POLYNOMIAL CURVE FITTING

In general, a polynomial fit to data in vector x and y is a function p of the form

$$p(x) = c_1x^d + c_2x^{d-1} + \dots + c_n$$

The degree is d , and the number of coefficients is $n = d+1$. Given a set of points in vectors x and y , **polyfit(x, y, d)** returns the coefficients of d th order polynomial in descending powers of x .

Example 1.11

Find a polynomial of degree 3 to fit the following data

x	0	1	2	4	6	10
y	1	7	23	109	307	1231

```
x = [ 0  1  2  4  6 10];
y = [ 1  7 23 109 307 1231];
c = polyfit(x,y,3)
```

The coefficients of a third degree polynomial are found as follows

```
c =
    1.0000    2.0000    3.0000    1.0000
```

i.e., $y = x^3 + 2x^2 + 3x + 1$.

1.9.3 POLYNOMIAL EVALUATION

If c is a vector whose elements are the coefficients of a polynomial in descending powers, the `polyval(c, x)` is the value of the polynomial evaluated at x . For example, to evaluate the above polynomial at points 0, 1, 2, 3, and 4, use the commands

```
c = [1 2 3 1];
x = 0:1:4;
y = polyval(c, x)
```

which result in

```
y =
    7    23    55   109
```

1.9.4 PARTIAL-FRACTION EXPANSION

`[r, p, k] = residue[b, a]` finds the residues, poles, and direct terms of a partial fraction expansion of the ratio of two polynomials

$$\frac{P(s)}{Q(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \cdots + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \cdots + a_1 s + a_0}$$

Vectors \mathbf{b} and \mathbf{a} specify the coefficients of the polynomials in descending powers of s . The residues are returned in column vector \mathbf{r} , the pole locations in column vector \mathbf{p} , and the direct terms in row vector \mathbf{k} .

Example 1.12

Determine the partial fraction expansion for

$$F(s) = \frac{2s^3 + 9s + 1}{s^3 + s^2 + 4s + 4}$$

```
b = [ 2 0 9 1];
a = [ 1 1 4 4];
[r,p,k] = residue(b,a)
```

The result is as follows

```
r =
    0.0000   -0.2500i
    0.0000    +0.2500i
   -2.0000
p =
    0.0000   +2.0000i
    0.0000   -2.0000i
   -1.0000
K =
    2.0000
```

Therefore the partial fraction expansion is

$$2 + \frac{-2}{s+1} + \frac{j0.25}{s+j2} + \frac{-j0.25}{s-j2} = 2 + \frac{-2}{s+1} + \frac{1}{s^2+4}$$

[b, a] = residue(r, p, K) converts the partial fraction expansion back to the polynomial $P(s)/Q(s)$.

For a complete list and help on matrix analysis, linear equations, eigenvalues, and matrix functions, type **help matfun**.

1.10 GRAPHICS

MATLAB can create high-resolution, publication-quality 2-D, 3-D, linear, semilog, log, polar, bar chart and contour plots on plotters, dot-matrix printers, and laser printers. Some of the 2-D graph types are **plot**, **loglog**, **semilogx**, **semi-logy**, **polar**, and **bar**. The syntax for the above plots includes the following optional symbols and colors.

<i>COLOR SPECIFICATION</i>		<i>LINE STYLE-OPTION</i>	
<i>Long name</i>	<i>Short name</i>	<i>Style</i>	<i>Symbol</i>
black	k	solid	—
blue	b	dashed	- -
cyan	c	dotted	:
green	g	dash-dot	-. .
magenta	m	point	.
red	r	circle	o
white	w	x-mark	x
yellow	y	plus	+
		star	*

Some of the Specialized 2-D plots are listed below:

area	Filled area plot
bar	Bar graph
barh	Horizontal bar graph
comet	Comet-like trajectory
ezplot	Easy to use function plotter
ezpolar	Easy to use polar coordinate plotter
feather	Feather plot
fill	Filled 2-D polygons
fplot	Plot function
hist	Histogram
pareto	Pareto chart
pie	Pie chart
plotmatrix	Scatter plot matrix
stem	Discrete sequence or "stem" plot
stairs	Stairstep plot

You have three options for plotting multiple curves on the same graph. For example,

```
plot(x1, y1, 'r', x2, y2, '+b', x3, y3, '--')
```

plots (x_1, y_1) with a solid red line, (x_2, y_2) with a blue + mark, and (x_3, y_3) with a dashed line. If **X** and **Y** are matrices of the same size, **plot(X, Y)** will plot the columns of **Y** versus the column of **X**.

Alternatively, the **hold** command can be used to place new plots on the previous graph. **hold on** holds the current plot and all axes properties; subsequent plot commands are added to the existing graph. **hold off** returns to the default mode whereby a new plot command replaces the previous plot. **hold**, by itself, toggles the hold state.

Another way for plotting multiple curves on the same graph is the use of the **line** command. For example, if a graph is generated by the command `plot(x1, y1)`, then the commands

```
line(x2, y2, '+b')
line(x3, y3, '--')
```

Add curve (x_2, y_2) with a blue + mark, and (x_2, y_2) with a dashed line to the existing graph generated by the previous plot command. Multiple figure windows can be created by the **figure** command. **figure**, by itself, opens a new figure window, and returns the next available figure number, known as the figure handle. **figure(h)** makes the figure with handle *h* the current figure for subsequent plotting commands. Plots may be annotated with title, $x - y$ labels and grid. The command **grid** adds a grid to the graph. The commands **title('Graph title')** titles the plot, and **xlabel('x-axis label')**, **ylabel('y-axis label')** label the plot with the specified string argument. The command **text(x-coordinate, y-coordinate, 'text')** can be used for placing text on the graph, where the coordinate values are taken from the current plot. For example, the statement

```
text(3.5, 1.5, 'Voltage')
```

will write Voltage at point (3.5, 1.5) in the current plot. Alternatively, you can use the **gtext('text')** command for interactive labeling. Using this command after a plot provides a crosshair in the Figure window and lets the user specify the location of the text by clicking the mouse at the desired location. Finally, the command **legend(string1, string2, string3, ...)** may be used to place a legend on the current plot using the specified strings as labels. This command has many optional arguments. For example, **legend(linetype1, string1, linetype2, string2, linetype3, string3, ...)** specifies the line types/color for each label at a suitable location. However, you can move the legend to a desired location with the mouse. **legend off** removes the legend from the current axes.

MATLAB provides automatic scaling. The command **axis([x min. x max. y min. y max.])** enforces the manual scaling. For example

```
axis([-10 40 -60 60])
```


produces an x -axis scale from -10 to 40 and a y -axis scale from -60 to 60 . Typing **axis** again or **axis('auto')** resumes auto scaling. Also, the aspect ratio of the plot can be made equal to one with the command **axis('square')**. With a square aspect ratio, a line with slope 1 is at a true 45 degree angle. **axis('equal')** will make the x - and y -axis scaling factor and tic mark increments the same. For a complete list and help on general purpose graphic functions, and two- and three-dimensional graphics, see **help graphics**, **help plotxy**, and **help plotxyz**.

There are many other specialized commands for two-dimensional plotting. Among the most useful are the **semilogx** and **semilogy**, which produce a plot with an x -axis log scale and a y -axis log scale. An interesting graphic command is the **comet** plot. The command **comet(x, y)** plots the data in vectors **x** and **y** with a comet moving through the data points, and you can see the curve as it is being plotted. For a complete list and help on general purpose graphic functions and two-dimensional graphics, see **help graphics** and **help plotxy**.

1.11 GRAPHICS HARD COPY

The easiest way to obtain hard-copy printout is to make use of the Windows built-in facilities. In the Figure window, you can pull down the file menu and click on the **Print** command to send the current graph directly to the printer. You can also import a graph to your favorite word processor. To do this, select **Copy options** from the **Edit** pull-down menu, and check mark the **Invert background** option in the dialog box to invert the background. Then, use **Copy** command to copy the graph into the clipboard. Launch your word processor and use the **Paste** command to import the graph.

Some word processors may not provide the extensive support of the Windows graphics and the captured graph may be corrupted in color. To eliminate this problem use the command

```
system_dependent(14, 'on')
```

which sets the metafile rendering to the lowest common denominator. To set the metafile rendering to normal, use

```
system_dependent(14, 'off')
```

In addition *MATLAB* provides a function called **print** that can be used to produce high resolution graphic files. For example,

```
print -dhpgl [filename]
```

saves the graph under the specified *filename* with extension *hgl*. This file may be processed with an HPGL-compatible plotter. Similarly, the command

```
print -dilll [filename]
```

produces a graphic file compatible with the Adobe Illustrator[®]88. Another **print** option allows you to save and reload a figure. The command

```
print -dmfile [ filename ]
```

produces a MAT file and M-file to reproduce the figure again.

In the Figure window, from the File pull-down menu you can use Save As... to save the figure with extension fig. This file can be opened in the Figure window again. Also, from the File pull-down menu you can use Expert... to save the graph in several different format, such as: emf, bmp, eps, ai, jpg, tiff, png, pcx, pbm, pgm, and ppm extensions.

Example 1.13

Create a linear X - Y plot for the following variables.

x	0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0
y	10	10	16	24	30	38	52	68	82	96	123

For a small amount of data, you can type in data explicitly using brackets.

```
x = [ 0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0];
y = [10 10 16 24 30 38 52 68 82 96 123];
plot(x, y), grid
xlabel('x'), ylabel('y'), title('A simple plot example')
```

plot(x, y) produces a linear plot of y versus x on the screen, as shown in Figure 1.2.

For large amounts of data, use the text editor to create a file with extension **m**. Typing the *filename* creates your data in the workspace.

Example 1.14

Fit a polynomial of order 2 to the data in Example 1.13. Plot the given data point with symbol x , and the fitted curve with a solid line. Place a boxed legend on the graph.

The command **p = polyfit(x, y, 2)** is used to find the coefficients of a polynomial of degree 2 that fits the data, and the command **yc = polyval(p, x)** is used to evaluate the polynomial at all values in **x**. We use the following command.

```
x = [ 0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0];
y = [10 10 16 24 30 38 52 68 82 96 123];
p = polyfit(x, y, 2) % finds the coefficients of a polynomial
                    % of degree 2 that fits the data
yc = polyval(p, x);%polynomial is evaluated at all points in x
plot(x, y, 'x', x, yc)%plots data with x and fitted polynomial
xlabel('x'), ylabel('y'), grid
title('Polynomial curve fitting')
legend('Actual data', 'Fitted polynomial', 4)
```

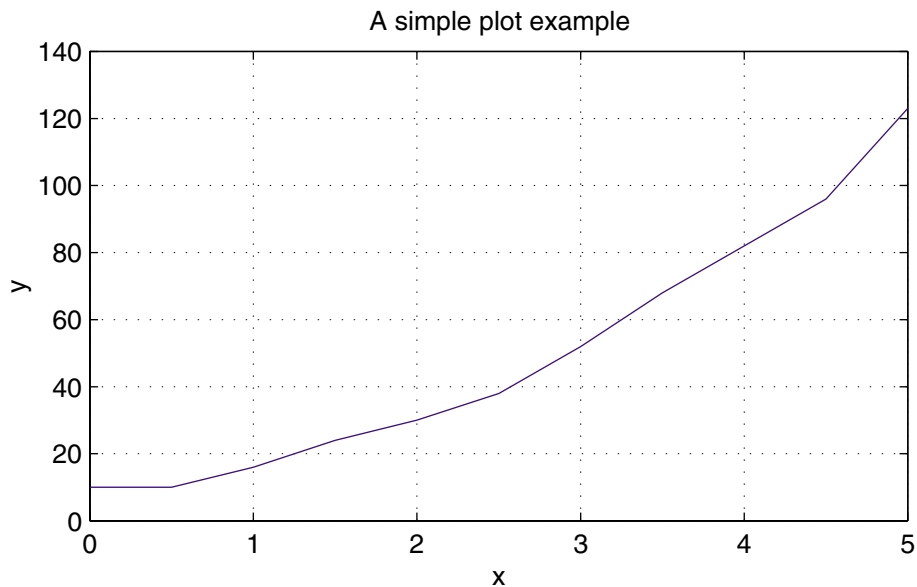


FIGURE 1.2
Example of X - Y plot.

The result is the array of coefficients of the polynomial of degree 2, and is

$$p = \begin{array}{ccc} & 4.0232 & 2.0107 & 9.6783 \end{array}$$

Thus, the parabola $4.0x^2 + 2.0x + 9.68$ is found that fits the given data in the least-square sense. The plots are shown in Figure 1.3.

Example 1.15

Plot function $y = 1 + e^{-2t} \sin(8t - \pi/2)$ from 0 to 3 seconds. Find the time corresponding to the peak value of the function and the peak value. The graph is to be labeled, titled, and have grid lines displayed.

Remember to use `.*` for the element-by-element multiplication of the two terms in the given equation. The command `[cp, k] = max(c)` returns the peak value and the index `k` corresponding to the peak time. We use the following commands.

```
t=0:.005:3; c = 1+ exp(-2*t).*sin(8*t - pi/2);
[cp, k] = max(c) % cp is the maximum value of c at interval k
tp = t(k) % tp is the peak time
plot(t, c), xlabel(' t - sec'), ylabel('c'), grid
title('Damped sine curve')
text(0.6, 1.4, ['cp =', num2str(cp)])%Text in quote & the value
% of cp are printed at the specified location
text(0.6, 1.3, ['tp = ', num2str(tp)])
```

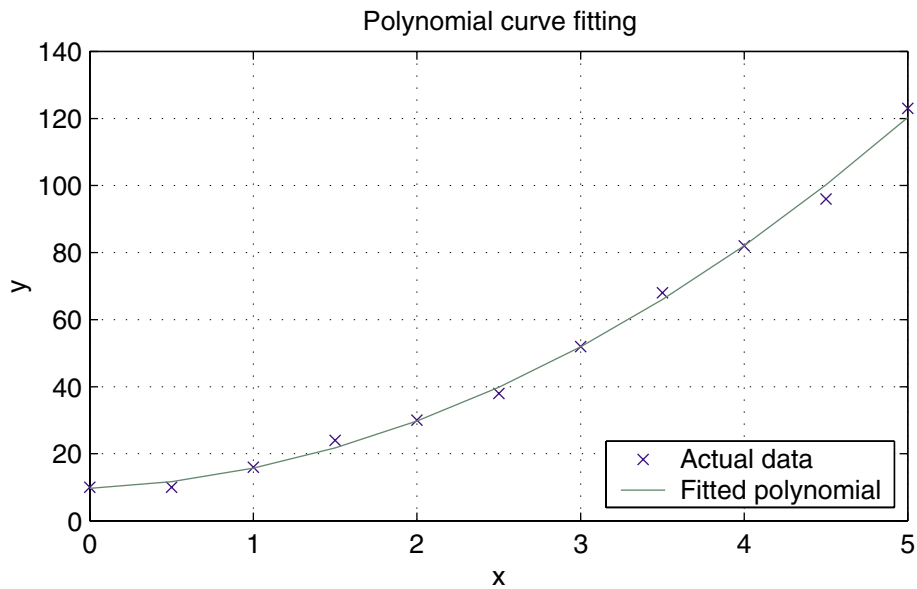


FIGURE 1.3
Fitting a parabola to the data in Example 1.13.

The result is

```
cp =
    1.4702
k =
    73
tp =
    0.3600
```

and the plot is shown in Figure 1.4.

An interactive way to find the data points on the curve is by using the **ginput** command. Entering `[x, y] = ginput` will put a crosshair on the graph. Position the crosshair at the desired location on the curve, and click the mouse. You can repeat this procedure for extracting coordinates for as many points as required. When the return key is pressed, the input is terminated and the extracted data is printed on the command menu. For example, to find the peak value and the peak time for the function in Example 1.15, try

```
[tp, cp] = ginput
```

A crosshair will appear. Move the crosshair to the peak position, and click the mouse. Press the return key to get

```
cp =
    1.47
tp =
    0.36
```

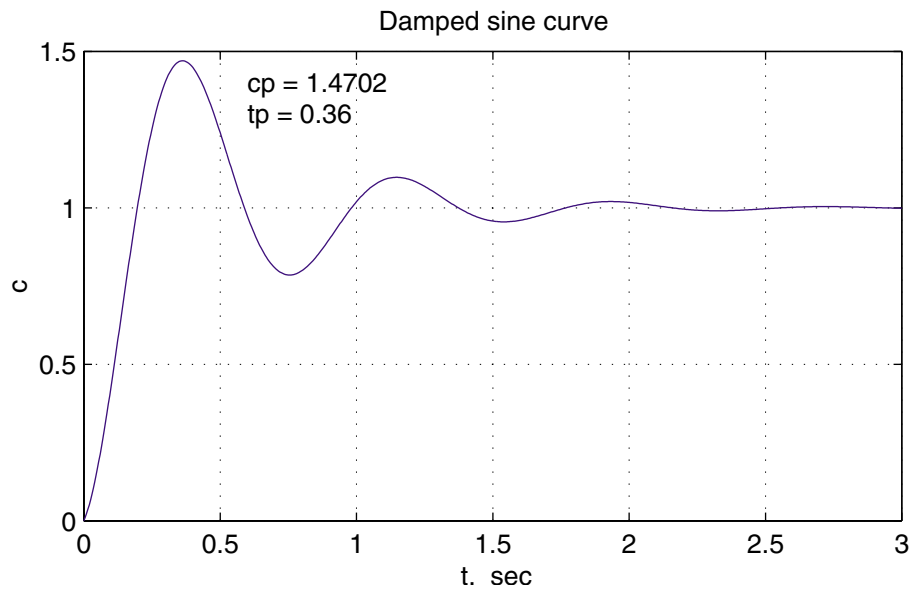


FIGURE 1.4
Graph of Example 1.15.

subplot splits the Figure window into multiple portions, in order to show several plots at the same time. The statement **subplot(m, n, p)** breaks the Figure window into an m -by- n box and uses the p th box for the subsequent plot. Thus, the command **subplot(2, 2, 3)**, **plot(x,y)** divides the Figure window into four subwindows and plots y versus x in the third subwindow, which is the first subwindow in the second row. The command **subplot(111)** returns to the default Figure window. This is demonstrated in the next example.

Example 1.16

Divide the Figure window into four partitions, and plot the following functions for ωt from 0 to 3π in steps of 0.05.

1. Plot $v = 120 \sin \omega t$ and $i = 100 \sin(\omega t - \pi/4)$ versus ωt on the upper left portion.
2. Plot $p = vi$ on the upper right portion.
3. Given $F_m = 3.0$, plot $f_a = F_m \sin \omega t$, $F_b = F_m \sin(\omega t - 2\pi/3)$, and $F_c = F_m \sin(\omega t - 4\pi/3)$ versus ωt on the lower left portion.
4. For $f_R = 3F_m$, construct a circle of radius f_R on the lower right portion.

```
wt = 0: 0.05: 3*pi; v=120*sin(wt);           %Sinusoidal voltage
i = 100*sin(wt - pi/4);                       %Sinusoidal current
```

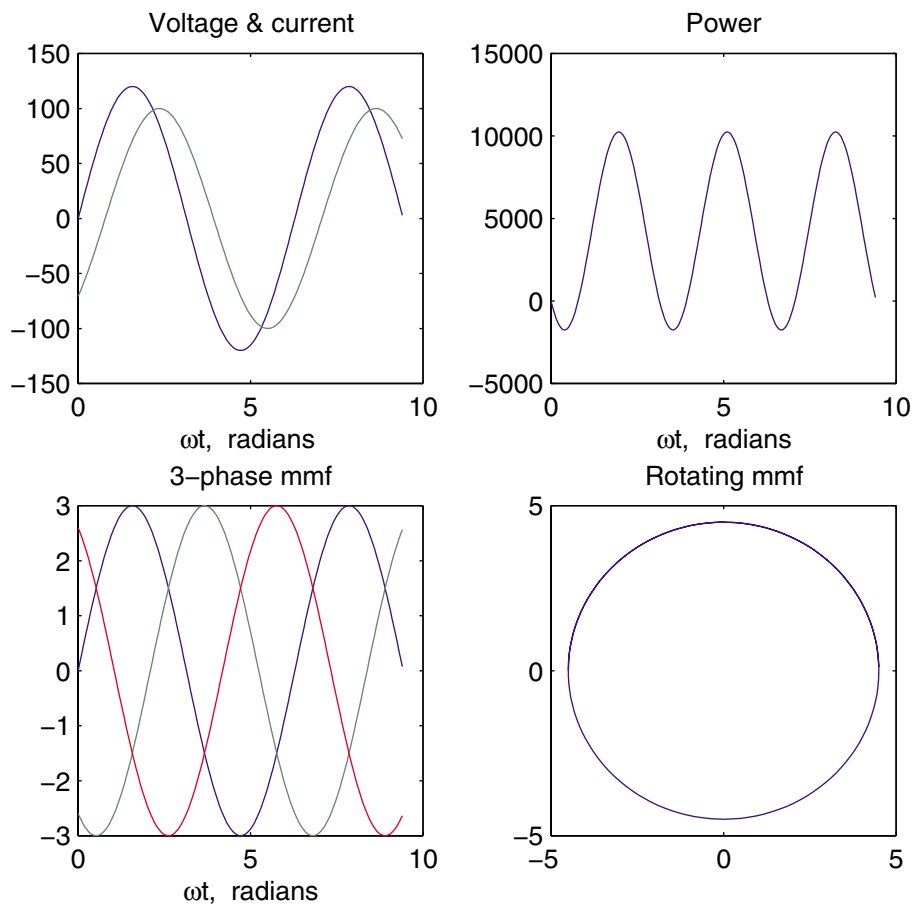


FIGURE 1.5
Subplot demonstration.

```

p = v.*i; %Instantaneous power
subplot(2, 2, 1), plot(wt, v, wt, i); %Plot of v & i versus wt
title('Voltage & current'), xlabel('\omegat, radians');
subplot(2, 2, 2), plot(wt, p); % Instantaneous power vs. wt
title('Power'), xlabel(' \omegat, radians ')
Fm=3.0;
fa = Fm*sin(wt); % Three-phase mmf's fa, fb, fc
fb = Fm*sin(wt - 2*pi/3); fc = Fm*sin(wt - 4*pi/3);
subplot(2, 2, 3), plot(wt, fa, wt, fb, wt, fc)
title('3-phase mmf'), xlabel(' \omegat, radians ')
fR = 3/2*Fm;
subplot(2, 2, 4), plot(-fR*cos(wt), fR*sin(wt))
title('Rotating mmf'), subplot(111)

```

Example 1.16 results are shown in Figure 1.5.

1.12 THREE-DIMENSIONAL PLOTS

MATLAB provides extensive facilities for visualization of three-dimensional data. The most common are plots of curves in a three-dimensional space, mesh plots, surface plots, and contour plots. The command **plot3(x, y, z, 'style option')** produces a curve in the three-dimensional space. The viewing angle may be specified by the command **view(azimuth, elevation)**. The arguments *azimuth*, and *elevation* specifies the horizontal and vertical rotation in degrees, respectively. The **title**, **xlabel**, **ylabel**, etc., may be used for three-dimensional plots. The **mesh** and **surf** commands have several optional arguments and are used for plotting meshes and surfaces. The **contour(z)** command creates a contour plot of matrix **z**, treating the values in **z** as heights above the plane. The statement **mesh(z)** creates a three-dimensional plot of the elements in matrix **z**. A mesh surface is defined by the **z** coordinates of points above a rectangular grid in the x-y plane. The plot is formed by joining adjacent points with straight lines. **meshgrid** transforms the domain specified by vector **x** and **y** into arrays **X** and **Y**. For a complete list and help on general purpose Graphic functions and three-dimensional graphics, see **help graphics** and **help plotxyz**. Also type **demo** to open the *MATLAB Expo Menu Map* and visit *MATLAB*. Select and observe the demos in the Visualization section.

Following is a list of elementary 3-D plots and some specialized 3-D graphs.

plot3	Plot lines and points in 3-D space
mesh	3-D mesh surface
surf	3-D colored surface
fill3	Filled 3-D polygons
comet3	3-D comet-like trajectories
ezgraph3	General purpose surface plotter
ezmesh	Easy to use 3-D mesh plotter
ezmeshc	Easy to use combination mesh/contour plotter
ezplot3	Easy to use 3-D parametric curve plotter
ezsurf	Easy to use 3-D colored surface plotter
ezsurfz	Easy to use combination surf/contour plotter
meshc	Combination mesh/contour plot
meshz	3-D mesh with curtain
scatter3	3-D scatter plot
stem3	3-D stem plot
surfz	Combination surf/contour plot
trisurf	Triangular surface plot
trimesh	Triangular mesh plot
cylinder	Generate cylinder
sphere	Generate sphere

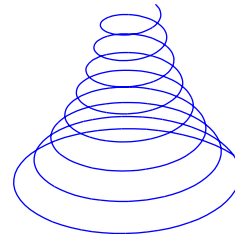
Example 1.17 ch1ex17.m

Few examples of 3-D plots and mesh plots are given in Figure 1.6.

Plot of a parametric space curve

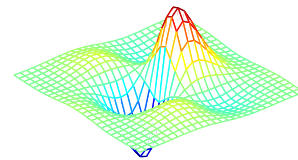
$$x(t) = e^{-0.03t} \cos t, \quad y(t) = e^{-0.03t} \sin t, \quad z(t) = t$$

```
t= 0:0.1:16*pi;
x=exp(-0.03*t).*cos(t);
y=exp(-0.03*t).*sin(t);
z=t;
plot3(x, y, z), axis off
```



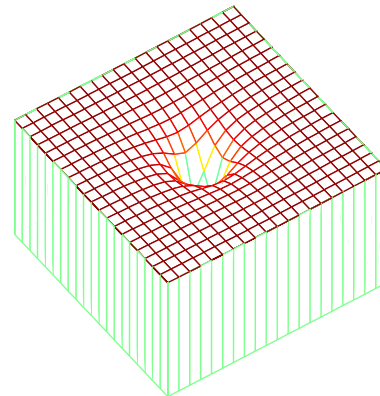
Plot of function $z = \sin x \cos y e^{-(x^2 + y^2)^{0.5}}$
using mesh

```
t = -4:0.3:4;
[x,y] = meshgrid(t,t);
z = sin(x).*cos(y).*exp(-(x.^2+y.^2).^0.5);
mesh(x,y,z), axis off
```



Plot of function $z = -0.1/(x^2 + y^2 + 1)$

```
using meshz
x= -3:0.3:3; y=x;
[x, y]=meshgrid(x,y);
z=-0.1./(x.^2+y.^2+.1);
meshz(z) , axis off
view(-35, 60)
```

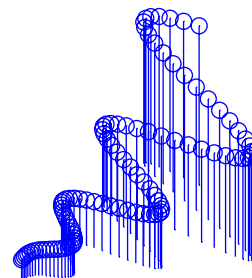


Discrete plot of

$$x=t, \quad y=t \cos t, \quad z=e^{0.1t}$$

using stem3

```
t=0:.2:20;
x=t; y=t.*cos(t);
z=exp(0.1*t);
stem3(x,y,z), axis off
```



Cylindrical surface created by

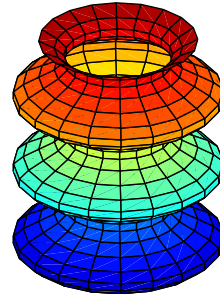
$$\rho = 3 + \sin t$$

using cylinder function

```
t=0:pi/5:6*pi;
```

```
p=3+sin(t);
```

```
cylinder(p), axis off
```



Plot of a unit sphere and a scaled sphere
using sphere function

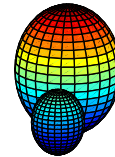
```
[x,y,z]=sphere(24);
```

```
subplot(2,2,2), surf(x-2, y-2, z-1);
```

```
hold on
```

```
surf(2*x, 2*y, 2*z);
```

```
axis off
```



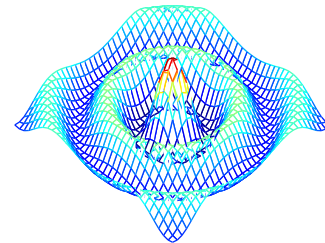
Cartezian plot of Bessel function

$$J_0[x^2+y^2]^{1/2} \quad -12 < x < 12, \quad -12 < y < 12$$

```
[x,y]=meshgrid(-12:.7:12, -12:.7:12);
```

```
r=sqrt(x.^2+y.^2);z= bessel(0,r);
```

```
m=[-45 60]; mesh(z,m), axis off
```



Contour lines and directional vectors
using contour and quiver functions

```
[x,y,z]=peaks(20);
```

```
[nx, ny]=gradient(z,1,1);
```

```
contour(x,y,z,10)
```

```
hold on
```

```
quiver(x,y,nx,ny)
```

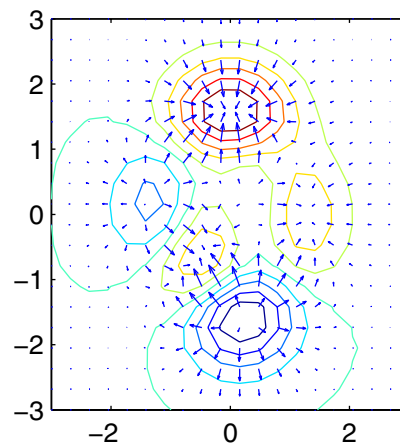


FIGURE 1.6

Graphs of Example 1.17.

For more specialized 3-D graphs and color related functions see **help specgraph**.

1.13 HANDLE GRAPHICS

It is often desirable to be able to customize the graphical output. *MATLAB* allows object-oriented programming, enabling the user to have complete control over the details of a graph. *MATLAB* provides many low-level commands known as *Handle Graphics*. These commands makes it possible to access individual objects and their properties and change any property of an object without affecting other properties or objects. Handle Graphics provides a graphical user interface (GUI) in which the user interface includes push buttons and menus. These topics are not discussed here; like *MATLAB* syntax, they are easy to follow, and we leave these topics for the interested reader to explore.

1.14 LOOPS AND LOGICAL STATEMENTS

MATLAB provides loops and logical statements for programming, like **for**, **while**, and **if** statements. The **for** statement instructs the computer to perform all subsequent expressions up to the **end** statement for a specified number of counted times. The expression may be a matrix. The following is an example of a nested loop.

```
for i = 1:n, for j = 1:n
    expression
end, end
```

The **while** statement allows statements to be repeated an indefinite number of times under the control of a logic statement. The **if**, **else**, and **elseif** statements allow conditional execution of statements. *MATLAB* has six relational operators and four logical operators, which are defined in the following table.

<i>Relational Operator</i>		<i>Logical Operator</i>	
==	equal	&	logical AND
~=	not equal		logical OR
<	less than	~	logical complement
<=	less than or equal to	xor	exclusive OR
>	greater than		
>=	greater than or equal to		

MATLAB is an interpreted language and macro operations such as matrices multiplies faster than micro operations such as incrementing an index, and use of loops are somewhat inefficient. Since variables in *MATLAB* are arrays and matrices, try to use vector operations as much as possible instead of loops. The loops should be used mainly for control operations. The use of loops are demonstrated in the next four examples.

Example 1.18

A rectangular signal can be represented as a series sum of harmonically related sine or cosine signals. Consider the partial sum of the following periodic signals (Fourier series).

$$\begin{aligned} x(t) &= \frac{4}{\pi} \left[\sin \omega_0 t + \frac{1}{3} \sin 3\omega_0 t + \frac{1}{5} \sin 5\omega_0 t + \dots \right] \\ &= \frac{4}{\pi} \sum_{n=1}^{\infty} \frac{1}{n} \sin n\omega_0 t \quad n \text{ is an odd integer} \end{aligned}$$

The following simple MATLAB statements uses a loop to generate this sum for any given odd integer.

```
n = input('Enter an odd integer');
w_0t = 0:.01:2*pi;
x = 0;
for k = 1:2:n;
    x = x + 1/k*sin(k*w_0t);
end
x=4/pi*x;
plot(w_0t, x), xlabel('\omega_0t')
text(3.5,.7,['Sum of ', num2str((n+1)/2),' sine waves'])
```

The result for $n = 101$ is plotted as shown in Figure 1.7.

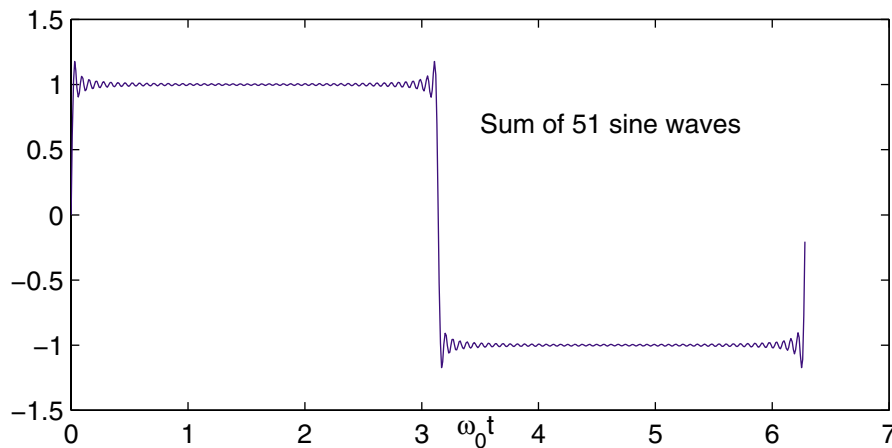


FIGURE 1.7
Graph Of Example 1.18.

Example 1.19

A network function known as transfer function is expressed by

$$F(s) = \frac{a_n s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0}{b_n s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0}$$

(a) write a *MATLAB* function named 'mybode' to evaluate the magnitude and phase angle of the above function for $s = j\omega$ where $0 < \omega < \infty$.

(b) Write a script function that uses 'mybode' to evaluate the magnitude and phase angle of

$$F(s) = \frac{1000(s+1)}{s^3 + 110s^2 + 1000s}$$

over a logarithmic range of $0.1 \leq \omega \leq 1000$.

The following function make use of two simple loops to sum up the numerator and denominator terms and returns an array containing magnitudes and phase angles in degree over the specified range.

```
% The function mybode returns the magnitude and phase
% angle of the frequency response transfer function.
% num and den are the numerator and denominator
% coefficients in descending order. w is the frequency
% array in rad/sec.
function[mag, phase] = mybode(num, den, w);
m=length(num); n=length(den);
N=num(m); D=den(n);
s=j*w;
for k=1:m-1
    N=N+num(m-k)*s.^k;
end
for k=1:n-1
    D=D+den(n-k)*s.^k;
end
H=N./D;
mag=abs(H);
phase=angle(H)*180/pi;
```

(b) The following script file uses 'mybode' to evaluate and plot the magnitude and phase angle of the function given (b) over the specified range of frequency.

```
num=[1000 1000], den=[1 110 1000 1];
w=logspace(-1, 3); % logarithmic range from 0.1 to 1000
[mag, phase]=mybode(num, den, w);
subplot(2,1,1), semilogx(w, mag), grid
xlabel('\omega'), ylabel('Magnitude')
subplot(2,1,2), semilogx(w, phase), grid
xlabel('\omega'), ylabel('\theta, degree')
```

The result is shown in Figure 1.8.

The *MATLAB* control system toolbox has a function called **bode** which obtains the frequency response plots of a given transfer function. This function is used in Chapters 4 and 7.

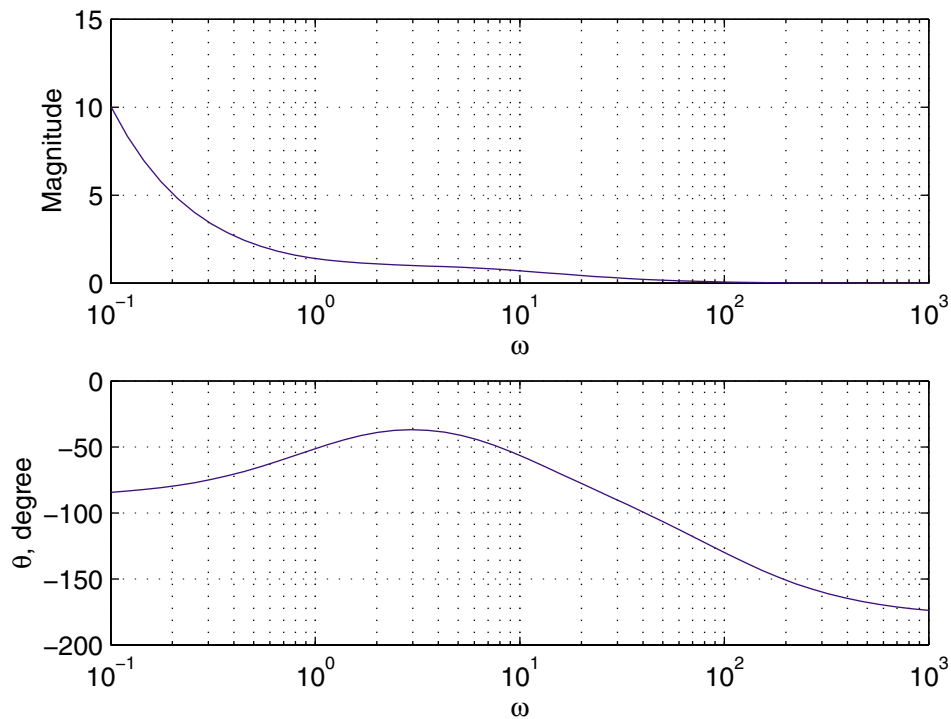


FIGURE 1.8
Magnitude and phase angle plots for Example 1.19.

Example 1.20

Another example of loops is in the numerical solution of differential equations. Euler's method is the simplest and the least accurate of all numerical methods. Consider the following simple one-dimensional first-order system.

$$a_1 \frac{dx}{dt} + a_0 x = c$$

rewriting in the form

$$\frac{dx}{dt} = \frac{c}{a_1} - \frac{a_0}{a_1} x$$

If at t_0 the value of $x(t_0)$ denoted by x_0 is given, the subsequent values of x can be determined by

$$x_{k+1} = x_k + \left. \frac{dx}{dt} \right|_{x_k} \Delta t$$

By applying the above algorithm successively, we can find approximate values of $x(t)$ at enough points from an initial state (t_0, x_0) to a final state (t_f, x_f) . Euler's method assumes that the derivative is constant over the entire interval Δt . An improvement

can be obtained by calculating the derivative at both the beginning and end of intervals, and then using their average value. This algorithm known as the *modified Euler's method* is given by

$$x_{k+1}^c = x_k + \left(\frac{\frac{dx}{dt} \Big|_{x_k} + \frac{dx}{dt} \Big|_{x_{k+1}^p}}{2} \right) \Delta t$$

Use the above algorithm to find the numerical solution of the following differential equation, and plot the result up to a final time of $t = 15$ seconds in steps of 0.01 second.

$$4 \frac{dx(t)}{dt} + 2x(t) = 10 \sin 8\pi t \quad \text{given, } x_0 = 1$$

we use the following statements

```
a1 = 4; a0 = 2;
x0 = 1; t0 = 0;                                % Initial state
Dt = 0.01; tf=15;                               % Step size and final time
t=[]; x=[];                                     % Initializing all the arrays
np = (tf -t0)/Dt;
t(1) = t0; x(1) = x0;
for k=1:np
    c=10*sin(pi*t(k));
    t(k+1)=t(k)+Dt;
    Dx1= c/a1 - a0/a1*x(k);                    % Derivative at the beginning
    x(k+1)=x(k)+Dx1*Dt;                        % Predicted value
    Dx2=c/a1 - a0/a1*x(k+1); % Derivative at the end of interval
    Dxavg=(Dx1+Dx2)/2; % Average value of the two derivatives
    x(k+1)=x(k) + Dxavg*Dt;                    % Corrected value
end
plot(t, x), grid
xlabel('t, sec'), ylabel('x(t)')
```

The result is shown in Figure 1.9.

MATLAB provides several powerful functions for the numerical solution of differential equations. Two of the functions employing the Runge-Kutta-Fehlberg methods are **ode23** and **ode45**, based on the Fehlberg second- and third-order pair of formulas for medium accuracy and forth- and fifth-order pair for higher accuracy. These functions are described and utilized in Chapter 2.

Example 1.21

Consider the system of n equations in n variables

$$\begin{aligned} f_1(x_1, x_2, \dots, x_n) &= c_1 \\ f_2(x_1, x_2, \dots, x_n) &= c_2 \\ &\dots\dots\dots \\ f_n(x_1, x_2, \dots, x_n) &= c_n \end{aligned}$$

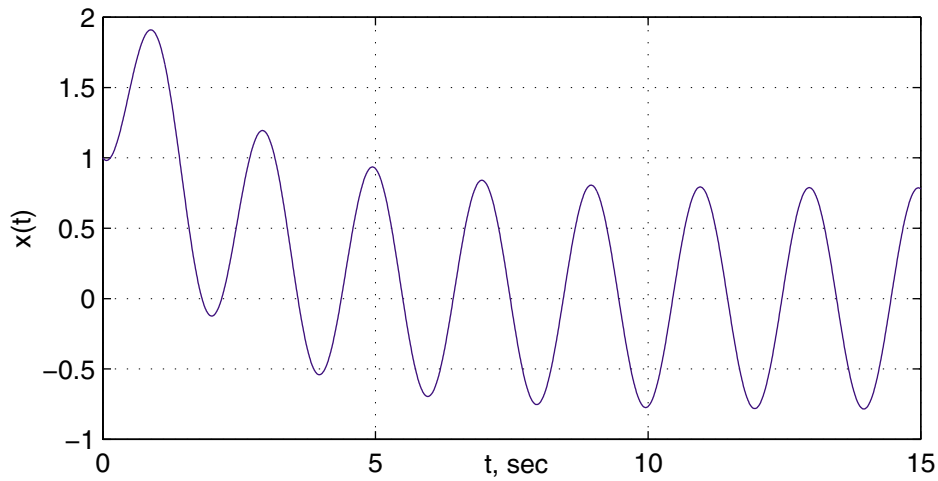


FIGURE 1.9
Numerical solution of Example 1.20.

The most widely used method for solving simultaneous nonlinear algebraic equations is the Newton-Raphson method. Newton's method is a successive approximation procedure based on an initial estimate of the unknown and the use of Taylor's series expansion and is given by

$$X^{(k+1)} = X^{(k)} + [J^{(k)}]^{-1} \Delta C^{(k)}$$

where $J^{(k)}$ is a matrix whose elements are the partial derivatives of $F^{(k)}$ and $\Delta C^{(k)}$ is

$$\Delta C^{(k)} = C - F^{(k)}$$

The iteration procedure is initiated by assuming an approximate solution for each of the independent variables ($x_1^{(0)}, x_2^{(0)}, \dots, x_n^{(0)}$). At the end of each iteration, the calculated values of all variables are tested against the previous values. If all changes in the variables are within the specified accuracy, a solution has converged, otherwise another iteration must be performed.

Starting with the initial values, $x_1 = 1$, $x_2 = 1$, and $x_3 = 1$, solve the following system of equations by the Newton-Raphson method.

$$\begin{aligned} x_1^2 - x_2^2 + x_3^2 &= 11 \\ x_1 x_2 + x_2^2 - 3x_3 &= 3 \\ x_1 - x_1 x_3 + x_2 x_3 &= 6 \end{aligned}$$

Taking partial derivatives of the above functions results in the Jacobian matrix

$$J = \begin{bmatrix} 2x_1 & -2x_2 & 2x_3 \\ x_2 & x_1 + 2x_2 & -3 \\ 1 - x_3 & x_3 & -x_1 + x_2 \end{bmatrix}$$

The following statements solve the given system of equations by the Newton-Raphson algorithm

```
Dx=[10;10;10]; %Change in variable is set to a high value
x=[1; 1; 1]; % Initial estimate
C=[11; 3; 6];
iter = 0; % Iteration counter
while max(abs(Dx))>=.0001 & iter<10; % Test for convergence
iter = iter + 1 % No. of iterations
F = [x(1)^2-x(2)^2+x(3)^2 % Functions
x(1)*x(2)+x(2)^2-3*x(3)
x(1)-x(1)*x(3)+x(2)*x(3)];
DC =C - F % Residuals
J = [2*x(1) -2*x(2) 2*x(3) % Jacobian matrix
x(2) x(1)+2*x(2) -3
1-x(3) x(3) -x(1)+x(2)]
Dx=J\DC %Change in variable
x=x+Dx % Successive solution
end
```

The program results for the first iteration are

```
DC =          J =
    10          2    -2    2
     4          1     3   -3
     5          0     1    0
Dx =          x =
   4.750        5.750
   5.000        6.000
   5.250        6.250
```

After six iterations, the solution converges to $x_1 = 2.0000$, $x_2 = 3.0000$, and $x_3 = 4.0000$.

1.15 SIMULATION DIAGRAM

The differential equations of a lumped linear network can be written in the form

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}u(t) \end{aligned} \quad (1.1)$$

This system of first-order differential equations is known as the state equation of the system, and \mathbf{x} is the state vector. One advantage of the state-space method is

that the form lends itself easily to the digital and/or analog computer methods of solution. Further, the state-space method can be easily extended to analysis of nonlinear systems. State equations may be obtained from an n th-order differential equation or directly from the system model by identifying appropriate state variables.

To illustrate how we select a set of state variables, consider an n th-order linear plant model described by the differential equation

$$\frac{d^n y}{dt^n} + a_{n-1} \frac{d^{n-1} y}{dt^{n-1}} + \dots + a_1 \frac{dy}{dt} + a_0 y = u(t) \quad (1.2)$$

where $y(t)$ is the plant output and $u(t)$ is its input. A state model for this system is not unique, but depends on the choice of a set of state variables. A useful set of state variables, referred to as *phase variables*, is defined as

$$x_1 = y, \quad x_2 = \dot{y}, \quad x_3 = \ddot{y}, \quad \dots, \quad x_n = y^{n-1}$$

We express $\dot{x}_k = x_{k+1}$ for $k = 1, 2, \dots, n-1$, and then solve for $d^n y/dt^n$, and replace y and its derivatives by the corresponding state variables to give

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ &\vdots \\ \dot{x}_{n-1} &= x_n \\ \dot{x}_n &= -a_0 x_1 - a_1 x_2 - \dots - a_{n-1} x_n + u(t) \end{aligned} \quad (1.3)$$

or in matrix form

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_{n-1} \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u(t) \quad (1.4)$$

and the output equation is

$$y = [1 \quad 0 \quad 0 \quad \dots \quad 0] \mathbf{x} \quad (1.5)$$

The M-file **ode2phv.m** is developed which converts an n th-order ordinary differential equation to the state-space phase variable form. $[\mathbf{A}, \mathbf{B}, \mathbf{C}] = \text{ode2phv}(\mathbf{a}, \mathbf{k})$ returns the matrices \mathbf{A} , \mathbf{B} , \mathbf{C} , where \mathbf{a} is a row vector containing coefficients of the equation in descending order, and \mathbf{k} is the coefficient of the right-hand side.

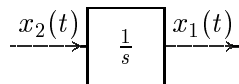
Equation (1.3) indicates that state variables are determined by integrating the corresponding state equation. A diagram known as the simulation diagram can be constructed to model the given differential equations. The basic element of the simulation diagram is the integrator. The first equation in (1.3) is

$$\dot{x}_1 = x_2$$

Integrating, we have

$$x_1 = \int x_2 dx$$

The above integral is shown by the following time-domain symbol. The integrating block is identified by symbol $\frac{1}{s}$. Adding an integrator for the remaining state variables and completing the last equation in (1.3) via a summing point and feedback paths, a simulation diagram is obtained.



1.16 INTRODUCTION TO SIMULINK

SIMULINK is an interactive environment for modeling, analyzing, and simulating a wide variety of dynamic systems. *SIMULINK* provides a graphical user interface for constructing block diagram models using “drag-and-drop” operations. A system is configured in terms of block diagram representation from a library of standard components. *SIMULINK* is very easy to learn. A system in block diagram representation is built easily and the simulation results are displayed quickly.

Simulation algorithms and parameters can be changed in the middle of a simulation with intuitive results, thus providing the user with a ready access learning tool for simulating many of the operational problems found in the real world. *SIMULINK* is particularly useful for studying the effects of nonlinearities on the behavior of the system, and as such, it is also an ideal research tool. The key features of *SIMULINK* are

- Interactive simulations with live display.
- A comprehensive block library for creating linear, nonlinear, discrete or hybrid multi-input/output systems.
- Seven integration methods for fixed-step, variable-step, and stiff systems.
- Unlimited hierarchical model structure.
- Scalar and vector connections.
- Mask facility for creating custom blocks and block libraries.

SIMULINK provides an open architecture that allows you to extend the simulation environment:

- You can easily perform “what if” analyses by changing model parameters – either interactively or in batch mode – while your simulations are running.
- Creating custom blocks and block libraries with your own icons and user interfaces from *MATLAB*, Fortran, or C code.
- You can generate C code from *SIMULINK* models for embedded applications and for rapid prototyping of control systems.
- You can create hierarchical models by grouping blocks into subsystems. There are no limits on the number of blocks or connections.
- *SIMULINK* provides immediate access to the mathematical, graphical, and programming capabilities of *MATLAB*, you can analyze data, automate procedures, and optimize parameters directly from *SIMULINK*.
- The advanced design and analysis capabilities of the toolboxes can be executed from within a simulation using the mask facility in *SIMULINK*.
- The *SIMULINK* block library can be extended with special-purpose blocksets. The DSP Blockset can be used for DSP algorithm development, while the Fixed-Point Blockset extends *SIMULINK* for modeling and simulating digital control systems and digital filters.

1.16.1 SIMULATION PARAMETERS AND SOLVER

You set the simulation parameters and select the solver by choosing **Parameters** from the Simulation menu. *SIMULINK* displays the **Simulation Parameters** dialog box, which uses three “pages” to manage simulation parameters. **Solver**, **Workspace I/O**, and **Diagnostics**.

SOLVER PAGE

The Solver page appears when you first choose **Parameters** from the **Simulation menu** or when you select the Solver tab. The Solver page allows you to:

- Set the start and stop times – You can change the start time and stop time for the simulation by entering new values in the Start time and Stop time fields. The default start time is 0.0 seconds and the default stop time is 10.0 seconds.
- Choose the solver and specify solver parameters – The default solver provide accurate and efficient results for most problems. Some solvers may be more efficient than others at solving a particular problem; you can choose between variable-step and fixed-step solvers. Variable-step solvers can modify their step sizes during the simulation. These are **ode45**, **ode23**, **ode113**, **ode15s**, **ode23s**, and **discrete**. The default is **ode45**. For variable-step solvers, you can set the maximum and suggested initial step size parameters. By default, these parameters are automatically determined, indicated by the value auto. For fixed-step solvers, you can choose **ode5**, **ode4**, **ode3**, **ode2**, **ode1**, and **discrete**.

- Output Options – The Output options area of the dialog box enables you to control how much output the simulation generates. You can choose from three popup options. These are: Refine output, Produce additional output, and Produce specified output only.

WORKSPACE I/O PAGE

The Workspace I/O page manages the input from and the output to the *MATLAB* workspace, and allows:

- Loading input from the workspace – Input can be specified either as *MATLAB* command or as a matrix for the Import blocks.
- Saving the output to the workspace – You can specify return variables by selecting the Time, State, and/or Output check boxes in the Save to workspace area.

DIAGNOSTICS PAGE

The Diagnostics page allows you to select the level of warning messages displayed during a simulation.

1.16.2 THE SIMULATION PARAMETERS DIALOG BOX

Table below summarizes the actions performed by the dialog box buttons, which appear on the bottom of each dialog box page.

Button	Action
Apply	Applies the current parameter values and keeps the dialog box open. During a simulation, the parameter values are applied immediately.
Revert	Changes the parameter values back to the values they had when the Dialog box was most recently opened and applies the parameters.
Help	Displays help text for the dialog box page.
Close	Applies the parameter values and closes the dialog box. During a simulation, the parameter values are applied immediately.

To stop a simulation, choose Stop from the Simulation menu. The keyboard shortcut for stopping a simulation is Ctrl-T. You can suspend a running simulation by choosing Pause from the Simulation menu. When you select Pause, the menu item changes to Continue. You proceed with a suspended simulation by choosing Continue.

1.16.3 BLOCK DIAGRAM CONSTRUCTION

At the *MATLAB* prompt, type *SIMULINK*. The *SIMULINK* BLOCK LIBRARY, containing seven icons, and five pull-down menu heads, appears. Each icon contains various components in the titled category. To see the content of each category, double click on its icon. The easy-to-use pull-down menus allow you to create a *SIMULINK* block diagram, or open an existing file, perform the simulation, and make any modifications. Basically, one has to specify the model of the system (state space, discrete, transfer functions, nonlinear ode's, etc), the input (source) to the system, and where the output (sink) of the simulation of the system will go. Generally when building a model, design it first on the paper, then build it using the computer. When you start putting the blocks together into a model, add the blocks to the model window before adding the lines that connect them. This way, you can reduce how often you need to open block libraries. An introduction to *SIMULINK* is presented by constructing the *SIMULINK* diagram for the following examples.

MODELING EQUATIONS

Here are some examples that may improve your understanding of how to model equations.

Example 1.22

Model the equation that converts Celsius temperature to Fahrenheit. Obtain a display of Fahrenheit-Celsius temperature graph over a range of 0 to 100°C.

$$T_F = \frac{9}{5}T_C + 32 \quad (1.6)$$

First, consider the blocks needed to build the model. These are:

- A ramp block to input the temperature signal, from the source library.
- A constant block, to define the constant of 32, also from the source library.
- A gain block, to multiply the input signal by 9/5, from the Linear library.
- A sum block, to add the two quantities, also from the Linear library.
- A scope block to display the output, from the sink library.

To create a *SIMULINK* block diagram presentation select **new...** from the **File** menu. This provides an untitled blank window for designing and simulating a dynamic system. Copy the above blocks from the block libraries into the new window by depressing the mouse button and dragging. Assign the parameter values to the Gain and Constant blocks by opening (double clicking on) each block and entering the appropriate value. Then click on the **close** button to apply the value and close the dialog box. The next step is to connect these icons together by drawing lines connecting the

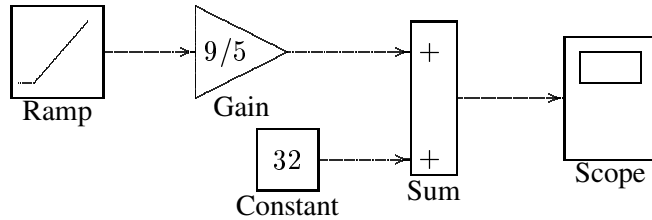


FIGURE 1.10
Simulink diagram for the system of Example 1.22.

icons using the left mouse button (hold the button down and drag the mouse to draw a line). You should now have the *SIMULINK* block diagram as shown in Figure 1.10.

The Ramp block inputs Celsius temperature. Open this block, set the Slope to 1, Start time to 0, and the Initial output to 0. The Gain block multiplies that temperature by the constant 9/5. The sum block adds the value 32 to the result and outputs the Fahrenheit temperature. Pull down the Simulation dialog box and select Parameters. Set the Start time to zero and the Stop Time to 100. Pull down the **File** menu and use **Save** to save the model under **simexa22** Start the simulation. Double click on the Scope, click on the **Auto Scale**, the result is displayed as shown in Figure 1.11.

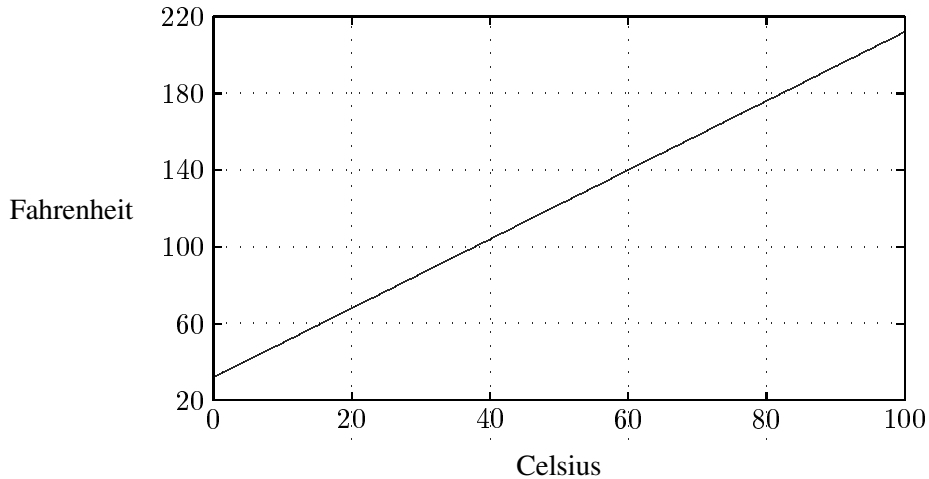


FIGURE 1.11
Fahrenheit-Celsius temperature graph for Example 1.22.

Example 1.23

Construct a simulation diagram for the state equation described by

$$\begin{aligned}\frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= \frac{1}{M}[f(t) - Bx_2 - Kx_1]\end{aligned}$$

where $M = 1$ kg, $B = 5$ N/m/sec, $K = 25$ N/m, and $f(t) = 25u(t)$.

The simulation diagram is drawn from the above equations by inspection and is shown in Figure 1.12.

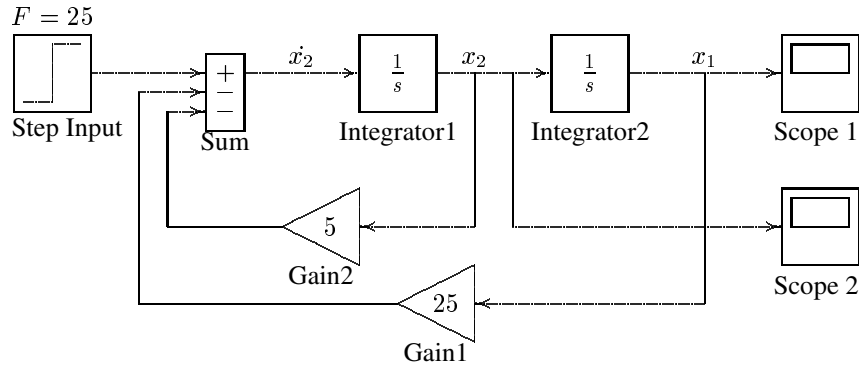


FIGURE 1.12
Simulink diagram for the system of Example 1.23.

To create a *SIMULINK* block diagram presentation select **new...** from the **File** menu. This provides an untitled blank window for designing and simulating a dynamic system. You can copy blocks from within any of the seven block libraries or other previously opened windows into the new window by depressing the mouse button and dragging. Open the **Source Library** and drag the Step Input block to your window. Double click on Step Input to open its dialog box. Set the step time to 0, and set the Initial Value to 0 and the Final Value to 25 to represent the step input. Open the **Linear Library** and drag the **Sum** block to the right of the Step Input block. Open the Sum dialog box and enter + - - under List of Signs. Using the left mouse button, click and drag from the Step output port to the Summing block input port to connect them. Drag a copy of the Integrator block from the **Linear Library** and connect it to the output port of the Sum block. Click on the Integrator block once to highlight it. Use the Edit command from the menu bar to copy and paste a second Integrator. Next drag a copy of the Gain block from the **Linear Library**. Highlight the Gain block, and from the pull-down **Options** menu, click on the Flip Horizontal to rotate the Gain block by 180° . Double click on Gain block to open its dialog box and set the gain to 5. Make a copy of this block and set its gain to 25. Connect the output ports of the Gain blocks to the Sum block and their input ports to the locations shown in Figure 1.12. Finally, get two Auto-Scale Graphs from the **Sink Library**, and connect them to the output of each Integrator. Before starting simulation, you must set the simulation parameters. Pull down the **Simulation** dialog box and select **Parameters**. Set the Start Time to zero, the Stop Time to 3, and for a more accurate integration, set the Maximum Step Size to 0.1. Leave the other parameters at their default values. Press OK to close the dialog box.

If you don't like some aspect of the diagram, you can change it in a variety of ways. You can move any of the icons by clicking on its center and dragging. You can move any of the lines by clicking on one of its corners and dragging. You can change the size and the shape of any of the icons by clicking and dragging on its corners. You can remove any line or icon by clicking on it to select it and using the

cut command from the **edit** menu. You should now have exactly the same system as shown in Figure 1.12. Pull down the **File** menu and use **Save as** to save the model under a file name **simexa23**. Start the simulation. *SIMULINK* will create the Figure windows and display the system responses. To see the second Figure window, click and drag the first one to a new location. The simulation results are shown in Figures 1.13 and 1.14.

SIMULINK enables you to construct and simulate many complex systems, such as control systems modeled by block diagram with transfer functions including the effect of nonlinearities. In addition, *SIMULINK* provides a number of built-in state variable models and subsystems that can be utilized easily.

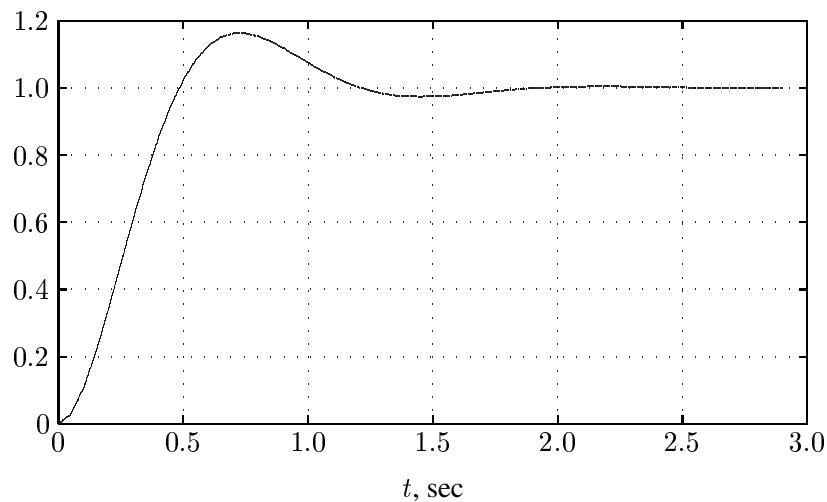


FIGURE 1.13
Displacement response of the system described in Example 1.23.

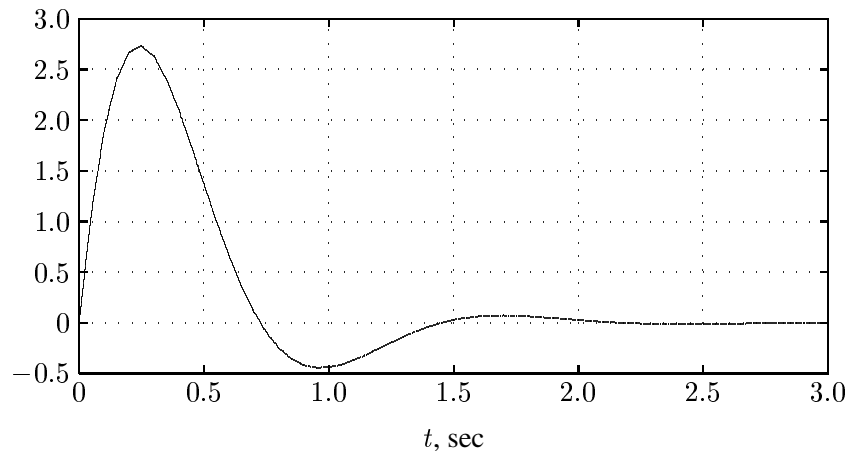


FIGURE 1.14
Velocity response of the system described in Example 1.23.

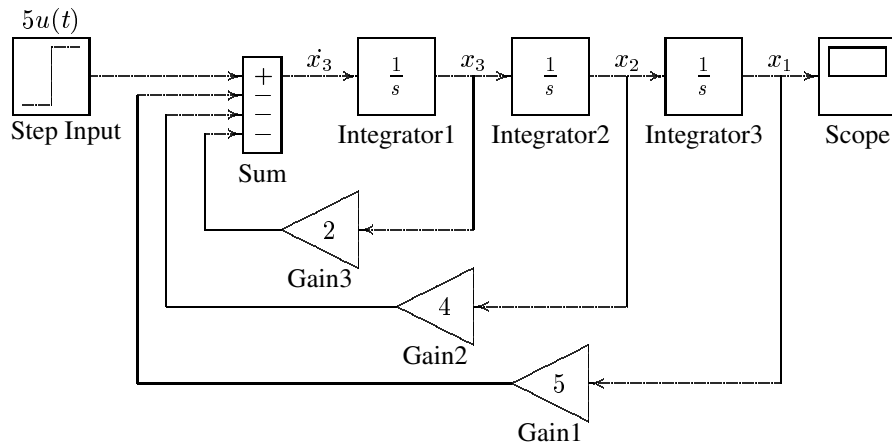


FIGURE 1.15
Simulation diagram for the system of Example 1.24.

Example 1.24

Consider the system defined by

$$2 \frac{d^3 y}{dt^3} + 4 \frac{d^2 y}{dt^2} + 8 \frac{dy}{dt} + 10y = 10u(t)$$

We have a third-order system; thus there are three state variables. Let us choose the state variables as

$$\begin{aligned} x_1 &= y \\ x_2 &= \dot{y} \\ x_3 &= \ddot{y} \end{aligned}$$

Then we obtain

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= -5x_1 - 4x_2 - 2x_3 + 5u(t) \end{aligned}$$

The last of these three equations was obtained by solving the original differential equation for the highest derivative term \ddot{y} and then substituting $y = x_1$, $\dot{y} = x_2$, and $\ddot{y} = x_3$ into the resulting equation. Using matrix notation, the state equation is

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -5 & -4 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix} u(t)$$

and the output equation is given by

$$y = [1 \quad 0 \quad 0] \mathbf{x}$$

The simulation diagram is obtained from the system differential equations and is given in Figure 1.15.

A *SIMULINK* Block diagram is constructed and saved as **simexa24**. The simulation response is shown in Figure 1.16.

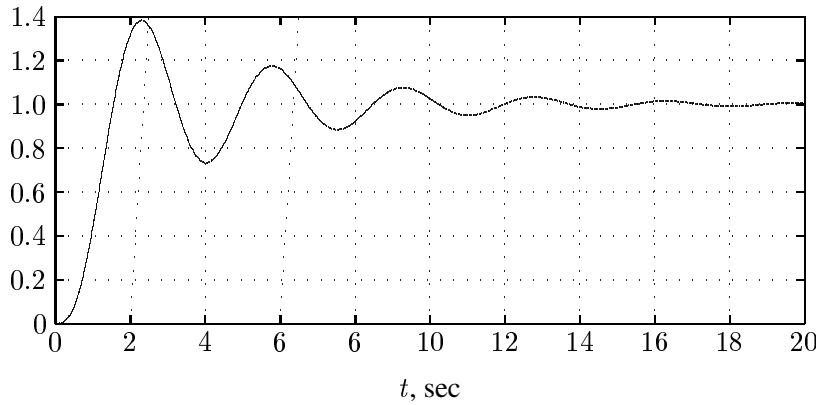


FIGURE 1.16
Simulation result for the system in Example 1.24.

Example 1.25

Use the **state-space** model to simulate the state and output equations described in Example 1.24.

The **State-Space** model provides a dialog box where the *A*, *B*, *C*, and *D* matrices can be entered in *MATLAB* matrix notation, or by variables defined in Workspace. A *SIMULINK* diagram using the **State-Space** model is constructed as shown in Figure 1.17, and saved as **simexa25**.

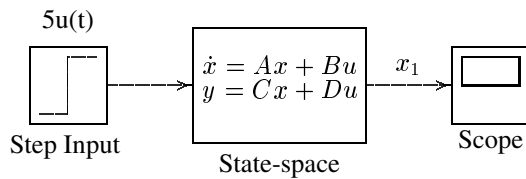


FIGURE 1.17
State-space model for system in Example 1.25.

Note that in this example, the output is given by $y = x_1$, and we define *C* as $C = [1 \ 0 \ 0]$. If it is desired to access all the states, then we can define *C* as an identity matrix, in this case a third order, i.e., $C = \text{eye}(3)$, and *D* as $D = \text{zeros}(3, 1)$. The output is a vector of state variables. A **DeMux** block may be added to produce individual states for graphing separately.

1.16.4 USING THE TO WORKSPACE BLOCK

The To Workspace block can be used to return output trajectories to the *MATLAB* Workspace. Example 1.26 illustrates this use.

Example 1.26

Obtain the step response of the following transfer function, and send the result to the *MATLAB* Workspace.

$$\frac{C(s)}{R(s)} = \frac{25}{s^2 + 2s + 25}$$

where $r(t)$ is a unit step function. The *SIMULINK* block diagram is constructed and saved in a file named **simexa26** as shown in Figure 1.18.

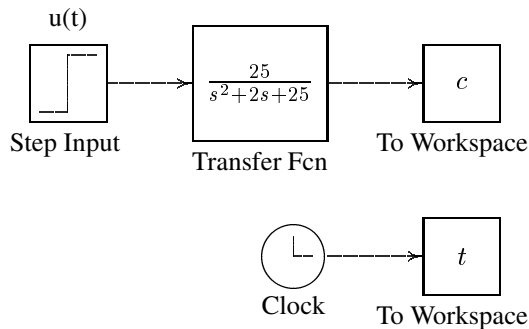


FIGURE 1.18
Simulink model for system in Example 1.26.

The To Workspace block can accept a vector input, with each input element's trajectories stored as a column vector in the resulting workspace variable. To specify the variables open the To Workspace block and for the variable name enter c . The time vector is stored by feeding a Clock block into To Workspace block. For this block variable name specify t . The vectors c and t are returned to *MATLAB* Workspace upon simulation.

1.16.5 LINEAR STATE-SPACE MODEL FROM SIMULINK DIAGRAM

SIMULINK provides the **linmod**, and **dlinmod** functions to extract linear models from the block diagram model in the form of the state-space matrices A , B , C , and D . State-space matrices describe the linear input-output relationship as

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (1.7)$$

$$y(t) = Cx(t) + Du(t) \quad (1.8)$$

The following Example illustrates the use of **linmod** function. The input and outputs of the *SIMULINK* diagram must be defined using **Inport** and **Outport** blocks in place of the **Source** and **Sink** blocks.

Thus, the transfer function model is

$$T(s) = \frac{20s + 40}{s^4 + 11s^3 + 66s^2 + 76s + 40}$$

Once the data is in the state-space form, or converted to a transfer function model, you can apply functions in Control System Toolbox for further analysis:

- Bode phase and magnitude frequency plot:

```
bode(A, B, C, D) or bode(num, den)
```

- Linearized time response:

```
step(A, B, C, D)    or  step(num, den)
lsim(A, B, C, D)   or  lsim(num, den)
impulse(A, B, C, D) or  impulse(num, den)
```

1.16.6 SUBSYSTEMS AND MASKING

SIMULINK subsystems, provide a capability within *SIMULINK* similar to subprograms in traditional programming languages.

Masking is a powerful *SIMULINK* feature that enables you to customize the dialog box and icon for a block or subsystem. With masking, you can simplify the use of your model by replacing many dialog boxes in a subsystem with a single one.

Example 1.28

To encapsulate a portion of an existing *SIMULINK* model into a subsystem, consider the *SIMULINK* model of Example 1.24 shown in Figure 1.20, and proceed as follows:

1. Select all the blocks and signal lines to be included in the subsystem with the bounding box as shown.
2. Choose Edit and select Create Subsystem from the model window menu bar. *SIMULINK* will replace the select blocks with a subsystem block that has an input port for each signal entering the new subsystem and an output port for each signal leaving the new subsystem. *SIMULINK* will assign default names to the input and output ports.

To mask a block, select the block, then choose Create Mask from the Edit menu. The Mask Editor appears. The Mask Editor consists of three pages, each handling a different aspect of the mask.

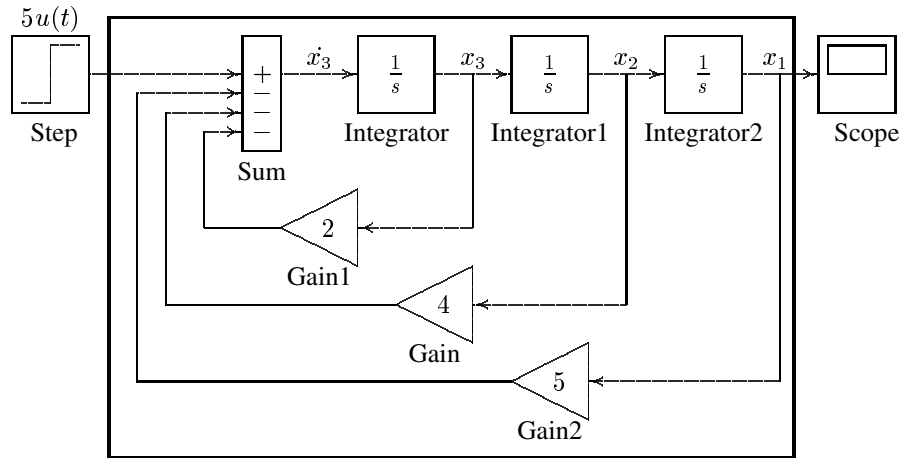


FIGURE 1.20
Simulation diagram for the system of Example 1.24.

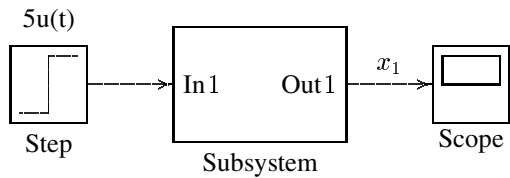


FIGURE 1.21
Simulation diagram for the system of Example 1.24.

- The Initialization page enables you to define and describe mask dialog box parameter prompts, name the variables associated with the parameters, and specify initialization commands.
- The Icon page enables you to define the block icon.
- The Documentation page enables you to define the mask type, and specify the block description and the block help.

In this example for icon the system transfer function is entered with command

```
dpoly([10], [2 4 8 10])
```

A short description of the system and relevant help topics can be entered in the Documentation page. The subsystem block is saved in a file named **simexa29**.

INDEX

- Array powers, 8
- Axis, 20
- Case-sensitive, 4
- Character String, 7
- Characteristic Polynomial, 15
- Colon, 9
- Column vector, 7
- Complex numbers, 13
- Division of polynomials, 16
- Dot product, 8
- Eigenvalues, 12
- Element-by-element division, 8
- Element-by-element multiplication, 8
- Elementary matrix operation, 10
- Function file, 3
- Graphic hard-copy, 21
- Graphics, 19
- Handle graphics, 30
- Help, 2
- Help Desk, 2
- Inner product, 8
- Installing the Text TOOLBOX, 2
- Logical statements, 30
- Loops, 30
- Matrix division, 10
- Matrix multiplication, 10
- One vector, 9
- Output format, 4
- Partial Fraction Expansion, 18
- Path Browser, 2
- Phase variables, 37
- Plant output, 37
- Polynomial curve fitting, 17
- Polynomial evaluation, 18
- Polynomial Roots, 15
- Product of polynomials, 16
- Row vector, 7
- Running MATLAB, 2
- Semicolon, 4
- Simulation diagram, 37
- Simulink, 38
- State equation, 36
- State variables, 37
- Subplot, 25
- Three-dimensional plots, 27
- Transpose, 7
- Utility Matrices, 12
- Variables, 4
- Vector operation, 7
- Zero vector, 9