APPENDIX D

Introduction to MATLAB

APPENDIX OUTLINE

This appendix provides a "quick start" introduction to MATLAB. You should be familiar with Sections D.1 and D.2 before attempting to cover the MATLAB material in the main body of this text. Sections D.3 and D.4 are useful for more extensive applications such as projects and some of the homework exercises. The extensive MATLAB help system discussed in Section D.5 can be used to obtain more detailed information and more examples. Other sources are [Palm, 2011] and [Palm, 2008].

Appendix A is a guide to the MATLAB commands and functions that are relevant to the system dynamics methods used in this text. To obtain more information about a specific command or function, in the Command window type `help topic`, where `topic` is the name of the command or function. For more information on MATLAB help, type `help help`.

MATLAB has a number of "add-on" software modules, called *toolboxes,* that perform more specialized computations. These can be purchased separately, but all require the "core" MATLAB program to be used. This text uses some features of the core MATLAB program, the Control Systems toolbox, and Simulink, which requires MATLAB to run. ■

D.1 MATLAB INTERACTIVE SESSIONS

You can use MATLAB in interactive mode to obtain an immediate response to each command as it is typed, or you can run MATLAB programs stored in files. We discuss the interactive mode in this section and program files in Section D.3.

D.1.1 TYPEFACE CONVENTIONS

We use `typewriter font` to represent MATLAB commands, any text that you type in the computer, and any MATLAB responses that appear on the screen; for example, `y = 6*x`. Variables in normal mathematics text appear in italics; for example, $y = 6x$. We use boldface type for three purposes: to represent vectors and matrices in normal mathematics text (for example, $\mathbf{Ax} = \mathbf{b}$), to represent a key on the keyboard (for example, **Enter**), and to represent the name of a screen menu or an item that appears in such a menu (for example, **File**). It is assumed that you press the **Enter** key after you type a command. We do not show this action with a separate symbol.

D.1.2 USER INTERFACE

To start MATLAB on a Windows system, double-click on the MATLAB icon. You will then see the MATLAB Desktop. The Desktop manages the Command window and a Help Browser, as well as other tools. The default appearance of the Desktop consists of four windows. These are the Current Folder window, the command window, the Command History window, and the workspace window. The Command History window displays your previous commands for quick reference. The Current Folder window provides easy access to program files. The Workspace window enables you to examine your variables. Across the top of the Desktop are a row of menu names and a row of icons called the "toolbar."

You use the Command window to communicate with the MATLAB program, by typing instructions of various types called *commands, functions,* and *statements*. To simplify the discussion, we will just call the instructions by the generic name *commands*. In the Windows operating system the professional version of MATLAB displays a prompt (≫) to indicate that it is ready to receive instructions. Other MATLAB versions, such as the student version, might use a different prompt. You can quit MATLAB by typing `quit`. On Windows systems you can also click on the **File** menu, and then click on **Exit Matlab.**

You can use the workspace window to determine the type, size, and values of all the variables you create. The Command History window shows all the previous keystrokes you entered in the Command window. You can alter the appearance of the Desktop if you wish. To eliminate a window, click on icon in its upper right-hand corner.

D.1.3 ENTERING COMMANDS AND EXPRESSIONS

After you type a command, press **Enter** to execute it. For example, to divide 8 by 10, type `8/10` and press **Enter** (the symbol / is the MATLAB symbol for division). Your entry and the MATLAB response looks like the following on the screen (we call this interaction between you and MATLAB an interactive *session,* or simply a *session*).

```
>>8/10
ans =
   0.8000
```

MATLAB has assigned the answer to a variable called `ans`, which is an abbreviation for *answer*. You can use the variable `ans` for further calculations; for example, using the MATLAB symbol for multiplication (`*`):

```
>>5*ans
ans =
   4.0000
```

Note that the variable `ans` now has the value 4.

MATLAB uses high precision for its computations, but by default it usually displays its results using four decimal places. This is called the *short* format. This default can be changed by using the `format` command. MATLAB uses the notation `e` to represent exponentiation to a power of ten; for example, MATLAB displays the number 5.316×10^2 as `5.316e+02`.

Because MATLAB retains your previous keystrokes in a "command file," you can use the up arrow key ↑ to scroll back through the commands. Press the key once to see the previous entry, twice to see the entry before that, and so on. Use the down arrow key ↓ to scroll forward through the commands. When you find the line you want, you can edit it using the left and right arrow keys ← and →, and the **Back Space** and **Delete** keys. Press the **Enter** key to execute the command.

D.1.4 VARIABLES

A *variable* in MATLAB is a symbol used to contain a value. You can use variables to write mathematical expressions and assign the result to a variable of your own choosing, say `r`, as follows:

```
>>r=8/10
r =
   0.8000
```

You can put spaces in the line to improve its readability; for example, you can put a space before and after the = sign if you want. MATLAB ignores these spaces when making its calculations.

A semicolon at the end of a line suppresses printing the results to the screen. Even if you suppress the display with the semicolon, MATLAB still retains the variable's value. You can put several commands on the same line if you separate them with a comma—if you want to see the results of the previous command—or semicolon if you want to suppress the display. If you need to type a long line, you can use an *ellipsis,* by typing three periods, to delay execution.

The term *workspace* refers to the names and values of any variables in use in the current work session. Variable names must begin with a letter and must contain less than 32 characters; the rest of the name can contain letters, digits, and underscore characters. MATLAB is case-sensitive. Thus the following names represent five different variables: `speed`, `Speed`, `SPEED`, `Speed_1`, and `Speed_2`.

MATLAB retains the last value of a variable until you clear its value or quit MATLAB. You can use the `clear` command to remove the values of *all* variables from memory, or you can use the form `clear var1 var2` to clear the variables named `var1` and `var2`. The effect of the `clc` command is different; it clears the Command window of everything in the window display, but the variables retain their values.

You can type the name of a variable and press **Enter** to see its current value. If the variable does not have a value (i.e., if it does not exist), you will see an error message. You can also use the `exist` command. Type `exist('x')` to see if the variable `x` is in use. If a 1 is returned, the variable exists; a 0 indicates that it does not exist. The `who` command lists the names of all the variables in memory, but does not give their values. The form `who var1 var2` restricts the display to the variables specified. The wildcard character `*` can be used to display variables that match a pattern. For instance, `who A*` finds all variables in the current workspace that start with A. The `whos` command lists the variable names and their sizes, and indicates whether or not they have nonzero imaginary parts.

D.1.5 SCALAR ARITHMETIC

A *scalar* is a single number and a *scalar variable* is a variable that is used to contain a single number. MATLAB uses the symbols `+ - * / ^` for addition, subtraction, multiplication, division, and exponentiation (power) of scalars. For example, typing `x = 8 + 3*5` returns the answer `x = 23`. Typing `2^3-10` returns the answer `ans = -2`. The *forward slash* `/` represents *right division,* which is the normal division operator familiar to you. Typing `15/3` returns the result `ans = 5`. MATLAB has another division operator, called *left division,* which is denoted by the *backslash* `\`. The left division operator is useful for solving sets of linear algebraic equations.

The mathematical operations represented by the symbols `+ - * / \` and `^` follow a set of rules called *precedence*. Mathematical expressions are evaluated starting from the left, with the exponentiation operation having the highest order of precedence, followed by multiplication and division with equal precedence, followed by addition and subtraction with equal precedence. Parentheses can be used to alter this order. Evaluation begins with the innermost pair of parentheses, and proceeds outward. Note the effect of precedence in the following session.

```
>>3*4^2 - 12 - 8/4*2 + 8 + 3*5
ans =
   55
>>3*(4^2) - 12 - 8/(4*2) + (8 + 3)*5
ans =
   90
>>3*(4^2) + 5 + 27^1/3 + 32^0.2
ans =
   64
>>(3*4)^2 + 5 + 27^(1/3) + 32^(0.2)
ans =
   154
```

To avoid mistakes, you should feel free to insert parentheses wherever you are unsure of the effect precedence will have on the calculation.

D.1.6 THE ASSIGNMENT OPERATOR

The = sign in MATLAB is the called the *assignment* operator or *replacement* operator. It works differently than the equals sign you are familiar with from mathematics. When you type `x = 3`, you tell MATLAB to assign the value 3 to the variable `x`. This usage is no different than in mathematics. However, in MATLAB we can also type something like this: `x = x + 2`. This tells MATLAB to add 2 to the current value of `x`, and to

replace the current value of `x` with this new value. If `x` originally had the value 3, its new value would be 5. This usage of the `=` operator is different than its use in mathematics. For example, the mathematics equation $x = x + 2$ is invalid because it implies that $0 = 2$ (subtract x from both sides of the equation to see this).

The variable on the *left-hand* side of the `=` operator is replaced by the value generated by the *right-hand* side. Thus, one variable, and only one variable, must be on the left-hand side of the `=` operator. Thus, in MATLAB you cannot type `6 = x`. Another consequence of this restriction is that you cannot write in MATLAB expressions like `x + 2 = 20`. The corresponding equation $x + 2 = 20$ is acceptable in algebra, and has the solution $x = 18$, but MATLAB cannot solve such an equation without additional commands that are available in the Symbolic Math toolbox.

Another restriction is that the right-hand side of the `=` operator must have a computable value. For example, if the variable `y` has not been assigned a value, then typing `x = 5 + y` will generate an error message in MATLAB.

D.2 COMPUTING WITH MATLAB

MATLAB has several predefined special constants. The symbol `Inf` stands for ∞, which in practice means a number so large that MATLAB cannot represent it. For example, typing `5/0` will generate the answer `Inf`. The symbol `NaN` stands for "Not a Number." It indicates an undefined numerical result such as that obtained by typing `0/0`. The symbol `eps` is the smallest number which, when added to 1 by the computer, creates a number greater than 1. It is used as an indicator of the accuracy of computations. The symbol `pi` represents the number $\pi = 3.14159\ldots$. The symbols `i` and `j` denote the imaginary unit, where $i = j = \sqrt{-1}$. They are used to create and represent complex numbers, such as `x = 5 + 8i`.

D.2.1 COMPLEX NUMBER ALGEBRA

MATLAB handles complex number algebra automatically. It accepts both `i` and `j` to designate the imaginary part, but it displays imaginary parts using `i` only. For example, the number $c_1 = 1 - 2i$ is entered as follows: `c1 = 1-2i`. Note that an asterisk is not needed between `i` or `j` and a number, although it is required with a variable, such as `c2 = 5 - i*c1`. This can cause errors if you are not careful. For example, the expressions `y = 7/2*i` and `x = 7/2i` give two different results: $y = (7/2)i = 3.5i$ and $x = 7/(2i) = -3.5i$.

Addition, subtraction, multiplication, and division of complex numbers are easily done. For example,

```
>>s = 3+7i;w = 5-9i;
>>w+s
ans =
   8.0000 - 2.0000i
>>w*s
ans =
   78.0000 + 8.0000i
>>w/s
ans =
   -0.8276 - 1.0690i
```

D.2.2 BUILT-IN FUNCTIONS

MATLAB has hundreds of built-in functions. One of these is the *square root* function, `sqrt`. A pair of parentheses is used after the function's name to enclose the value—called the function's *argument*—that is operated on by the function. For example, to compute the square root of 9, you type `sqrt(9)`. To compute sin x, where x has a value in radians, you type `sin(x)`. To compute cos x, type `cos(x)`. The exponential function e^x is computed from `exp(x)`. The natural logarithm, ln x, is computed by typing `log(x)`. You compute the base 10 logarithm by typing `log10(x)`. The inverse sine, or arcsine, is obtained by typing `asin(x)`. It returns an answer in radians, not degrees. Appendix A lists the available elementary functions. Type `help elfun` to obtain more information.

The difference between a function and a command or a statement is that functions have their arguments enclosed in parentheses. For example, to compute sin 2 using the `sin` function, you type `sin(2)`. Commands, such as `clear`, need not have arguments, but if they do, they are not enclosed in parentheses; for example, `clear x`. Statements cannot have arguments; for example, `clc` and `quit` are statements.

D.2.3 ARRAYS

One of the strengths of MATLAB is its ability to handle collections of numbers, called *arrays,* as if they were a single variable. A numerical array is an ordered collection of numbers (a set of numbers arranged in a specific order). An example of an array is one that contains the numbers 0, 1, 3, and 6 in that order. We can use square brackets to define the variable `x` to contain this collection by typing `x = [0, 1, 3, 6]`. The elements of the array must be separated by commas or spaces. Note that the variable `y` defined as `y = [6, 3, 1, 0]` is not the same as `x` because the order is different.

We can add the two arrays `x` and `y` to produce another array `z` by typing the single line `z = x + y`. To compute `z`, MATLAB adds all the corresponding numbers in `x` and `y` to produce `z`. The resulting array `z` is `[6, 4, 4, 6]`.

Array addition and subtraction require that both arrays have the same size. The only exception to this in MATLAB occurs when we add or subtract a *scalar* to or from an array. In this case, the scalar is added or subtracted from each element in the array. For example, if `x = [0, 1, 3, 6]`, typing `v = x + 2` gives `v = [2, 3, 5, 8]`.

Arrays that display on the screen as a single row of numbers with more than one column are called *row arrays*. You can create *column* arrays, which have more than one row, by using a semicolon to separate the rows. So, for example, `[3, 7, 2]` is not the same array as `[3; 7; 2]`.

D.2.4 CREATING AND ACCESSING ARRAYS

To create an array having many entries, you need not type all the numbers if they are regularly spaced. Instead, you type the first number, the spacing, and the last number, separated by colons. For example, the numbers 0, 0.1, 0.2, . . . , 10 can be assigned to the variable `u` by typing `u = [0:0.1:10];` to create a row array having 101 elements. Use the transpose operator `'` to convert a row array into a column array, and vice versa.

You can see all the values in `u` by typing `u` after the prompt or, for example, you can see the seventh value by typing `u(7)`. The number 7 is called an *array index,* because it points to a particular element in the array. For example, typing `u(7)` gives

`ans = 0.6000`. You can use the `length` function to determine how many values are in an array. For example, typing `m = length(u)` gives `m = 101`.

One advantage of MATLAB is that it can operate on arrays with a single command. For example, to compute $w = 5 \sin u$ for $u = 0, 0.1, 0.2, \ldots, 10$, the session is

```
>>u = [0:0.1:10]; w = 5*sin(u);
```

or more compactly,

```
>>w = sin([0:0.1:10]);
```

This single line computes the formula $w = 5 \sin u$ 101 times, once for each value in the array `u`, to produce an array `w` that has 101 values. To achieve this result with conventional programming languages requires that a loop be written.

D.2.5 MATRICES

A *matrix* is a two-dimensional array; that is, one that has multiple rows and multiple columns. One way to create a matrix is to type each row as a row array (with its elements separated by a comma or a space) followed by a semicolon except for the last row. The semicolon serves to separate the rows. For example, to create the matrix

$$\mathbf{A} = \begin{bmatrix} 4 & 7 \\ 9 & 2 \end{bmatrix}$$

the session is

```
>>A = [4, 7; 9, 2];
```

To access the element of a matrix in row r and column c, type `A(r,c)`. For example, typing `A(2,1)` returns the value 9. Note that the row number comes first.

Matrix addition, subtraction, and multiplication by a scalar follow the same rules as for row and column arrays.

D.2.6 ARRAY MULTIPLICATION

Multiplication of arrays by a scalar is easily understood. For example, to double each element of the array `r = [3, 5, 2]`, you type `v = 2*r` to obtain `v = [6, 10, 4]`. Similarly, multiplying a matrix **A** by a scalar w produces a matrix whose elements are the elements of **A** multiplied by w. However, multiplication of two arrays is not so straightforward. In fact, there are two definitions of multiplication used by MATLAB: (1) array multiplication and (2) matrix multiplication.

To compute the matrix product **AB** you type `A*B`. The resulting matrix is computed using the standard rules of matrix multiplication. However, typing `A.*B` produces a different matrix, in which each element is the product of the two corresponding elements in `A` and `B`. This form is called *array or element-by-element multiplication*. Note that `.*` is *one* symbol.

To illustrate the difference between the two types of multiplication, consider this table, which gives the speed of an aircraft on each leg of a certain trip, and the time spent on each leg.

	Leg			
	1	**2**	**3**	**4**
Speed (mi/hr)	200	250	400	300
Time (hr)	2	5	3	4

We can define a row array `s` containing the speeds and a row array `t` containing the times for each leg. Thus `s = [200, 250, 400, 300]` and `t = [2, 5, 3, 4]`. To find the miles traveled on each leg, we multiply the speed by the time. To do this we use the multiplication `s.*t` to produce the row array whose elements are the products of the corresponding elements in `s` and `t`. This process is illustrated as follows.

$$\texttt{s.*t} = [200(2), 250(5), 400(3), 300(4)] = [400, 1250, 1200, 1200]$$

The resulting array contains the miles traveled by the aircraft on each leg of the trip.

To find only the total miles traveled, we can use matrix multiplication `s*t'`, where the `'` denotes the transpose operation (note that the operator `.*` is not used here). The transpose converts the row array `t` into a column array. This product gives the *sum* of the individual element products; that is

$$\texttt{s*t'} = [200(2) + 250(5) + 400(3) + 300(4)] = 4050$$

The complete session to perform these calculations is

```
>>s = [200, 250, 400, 300];
>>t = [2, 5, 3, 4];
>>miles_per_leg = s.*t
miles_per_leg =
   400    1250    1200    1200
>>total_miles = s*t'
total_miles =
   4050
```

D.2.7 ARRAY DIVISION AND EXPONENTIATION

Array division, also called element-by-element division, is defined similarly to array multiplication, except of course that the elements of one array are divided by the elements of the other array. Both arrays must have the same size. The symbol for array right division is `./`. For example, if `x = [8, 12, 15]` and `y = [-2, 6, 5]` then `z = x./y` gives `z = [-4, 2, 3]`.

In MATLAB not only can we can raise arrays to powers, but we can also raise scalars and arrays to *array* powers. To perform exponentiation on an element-by-element basis, we must use the `.^` symbol. For example, if `x = [3, 5, 8]`, then typing `x.^3` produces the array $[3^3, 5^3, 8^3] = [27, 125, 512]$.

We can raise a scalar to an array power. For example, if `p = [2, 4, 5]`, then typing `3.^p` produces the array $[3^2, 3^4, 3^5] = [9, 81, 243]$. This illustrates a common situation where it helps to remember that `.^` is a *single* symbol; the dot in `3.^p` is not a decimal point associated with the number 3. Thus `3.^p` gives the same result as `3..^p` or as `(3).^p`.

D.2.8 POLYNOMIAL ROOTS

We can describe a polynomial in MATLAB with an array whose elements are the polynomial's coefficients, *starting with the coefficient of the highest power*. For example, the polynomial $4x^3 - 8x^2 + 7x - 5$ would be represented by the array `[4, -8, 7, -5]`. Polynomial roots can be found with the `roots(a)` function, where `a` is the polynomial's coefficient array. The result is a column array that contains the polynomial's

roots. For example, to find the roots of $x^3 - 7x^2 + 40x - 34 = 0$, the session is

```
>>roots([1,-7,40,-34])
ans =
   3.0000 + 5.000i
   3.0000 - 5.000i
   1.0000
```

The roots are $x = 3 \pm 5i$ and $x = 1$.

The `poly(r)` function computes the coefficients of the polynomial whose roots are specified by the array `r`. The result is a row array that contains the polynomial's coefficients. To find the polynomial whose roots are 1 and $3 \pm 5i$, the session is

```
>>poly([1,3+5i,3-5i])
ans =
   1.0000     -7.0000     40.0000      -34.0000
```

Thus, the polynomial is $x^3 - 7x^2 + 40x - 34$.

D.2.9 POLYNOMIAL ALGEBRA

You can do polynomial algebra with MATLAB. To add two polynomials, add the arrays that describe their coefficients. If the polynomials are of different degrees, add zeroes to the coefficient array of the lower-degree polynomial. For example, consider $f(x) = 9x^3 - 5x^2 + 3x + 7$, whose coefficient array is `f = [9, -5, 3, 7]`, and $g(x) = 6x^2 - x + 2$, whose coefficient array is `g = [6, -1, 2]`. The degree of $g(x)$ is one less that of $f(x)$. Therefore, to add $f(x)$ and $g(x)$, we append one 0 to `g` to "fool" MATLAB into thinking $g(x)$ is a third-degree polynomial. That is, we type `g = [0 g]` to obtain `[0, 6, -1, 2]` for `g`. This represents $g(x) = 0x^3 + 6x^2 - x + 2$. To add the polynomials, type `h = f + g`. The result is `h = [9, 1, 2, 9]`, which corresponds to $h(x) = 9x^3 + x^2 + 2x + 9$. Subtraction is done in a similar way.

To multiply a polynomial by a scalar, simply multiply the coefficient array by that scalar. For example, $5h(x)$ is obtained from `5*h`, where `h = [45, 5, 10, 45]`.

Multiplication of polynomials by hand can be tedious, and polynomial division is even more so, but these operations are easily done with MATLAB. Use the `conv` command to multiply polynomials (it stands for "convolve"), and use the `deconv` command to perform synthetic division (`deconv` stands for "deconvolve").

Consider the product

$$f(x)g(x) = (9x^3 - 5x^2 + 3x + 7)(6x^2 - x + 2) = 54x^5 - 39x^4 + 41x^3 + 29x^2 - x + 14$$

Dividing $f(x)$ by $g(x)$ using synthetic division gives a quotient of

$$\frac{f(x)}{g(x)} = \frac{9x^3 - 5x^2 + 3x + 7}{6x^2 - x + 2} = 1.5x - 0.5833$$

with a remainder of $-0.5833x + 8.1667$. The MATLAB session to perform these operations is

```
>>f = [9, -5, 3, 7]; g = [6, -1, 2];
>>product = conv(f,g)
product =
   54   -39   41    29    -1    14
```

```
>>[quotient, remainder] = deconv(f,g)
quotient =
   1.5    -0.5833
remainder =
   0    0    -0.5833    8.1667
```

The `conv` and `deconv` commands do not require that the polynomials have the same degree.

The `polyval` function is useful for plotting polynomials, and is discussed later. Roots of functions other than polynomials can be obtained with the `fzero` function. Type `help fzero` for more information.

D.2.10 PLOTTING WITH MATLAB

MATLAB contains many powerful functions and commands for easily creating plots of several different types, such as rectilinear, logarithmic, surface, and contour plots, for example. To plot the function $y = \sin 2x$ for $0 \leq x \leq 10$, we choose an increment of 0.01 to generate a large number of `x` values in order to produce a smooth curve. The function `plot(x,y)` generates a plot with the `x` values on the horizontal axis (the abscissa) and the `y` values on the vertical axis (the ordinate). The session is

```
>>x = [0:0.01:10]; y = sin(2*x);
>>plot(x,y),xlabel('x'),ylabel('sin(2x)')
```

The `xlabel` function places the text in single quotes as a label on the horizontal axis. The `ylabel` function performs a similar function for the vertical axis. The plot appears on the screen in a graphics window. If a hard copy of the plot is desired, the plot can be printed by selecting **Print** from the **File** menu on the graphics window. The window can be closed by selecting **Close** on the **File** menu in the graphics window. You will then be returned to the prompt in the Command window.

Other useful plotting functions are `title` and `gtext`. These functions place text on the plot. Both accept text within parentheses and single quotes, as with the `xlabel` function. The `title` function places the text at the top of the plot; the `gtext` function places the text at the point on the plot where the cursor is located when you click the left mouse button.

Sometimes it is useful or necessary to obtain the coordinates of a point on a plotted curve. The function `ginput` can be used for this purpose. Place it at the end of all the plot and plot formatting statements, so that the plot will be in its final form. The command `[x,y] = ginput(n)` gets n points and returns the x and y coordinates in the vectors `x` and `y`, which have a length n. Position the cursor using a mouse, and press the mouse button. The returned coordinates have the same scale as the coordinates on the plot.

In cases where you are plotting *data,* as opposed to functions, you should use *data markers* to plot each data point (unless there are very many data points). To mark each point with a plus sign +, the required syntax for the `plot` function is `plot(x,y,'+')`. You can connect the data points with lines if you wish. In that case, you must plot the data twice, once with a data marker, and once without a marker. For example, suppose the data for the independent variable is `x = [15:2:23]` and the dependent variable values are `y = [20, 50, 60, 90, 70]`. To plot the y data with plus signs connected by straight lines, type `plot(x,y,'+',x,y)`. Other data markers are available. Type `help plot` for information about other markers.

D.2.11 MULTIPLE PLOTS

You can create multiple plots—called *overlay* plots—by including another set or sets of values in the `plot` function. For example, to plot the functions $y = 2\sqrt{x}$ and $z = 4 \sin 3x$ for $0 \leq x \leq 5$ on the same plot, the session is

```
>>x = [0:0.01:5]; y = 2*sqrt(x); z = 4*sin(3*x);
>>plot(x,y,x,z),xlabel('x'),gtext('y'),gtext('z')
```

After the plot appears on the screen, the program waits for you to position the cursor and click the mouse button, once for each `gtext` function used. Although MATLAB displays different colors for each curve, if you are going to print the plot on a black-and-white printer, you should label each curve so that you know which curve represents y and which curve represents z. One way of doing this is to use the `gtext` function to place the labels y and z next to the appropriate curves, as shown in the preceding session. Another way is to use the `legend` function. The plotting functions `xlabel`, `ylabel`, `title`, and `gtext` must be placed after the `plot` function and separated by commas.

You can also distinguish curves from one another by using different line types for each curve. For example, to plot the z curve using a dashed line, replace the `plot(x,y,x,z)` function in the preceding session with `plot(x,y,x,z,'--')`. Other line types can be used. Type `help plot` for information about these line types.

You can use the `subplot` function to create multiple but separate plots in the same figure window (and on the same printed page). The `subplot` function creates a rectangular tiling having r rows and c columns of "panes," each containing a plot. The syntax is `subplot(r,c,n)`, where n denotes the pane number currently being plotted. For example, to plot `y` and `z` created in the previous session as two separate plots, with the plot of `y` above the plot of `z`, the session is

```
>>subplot(2,1,1),plot(x,y),xlabel('x'),ylabel('y')
>>subplot(2,1,2),plot(x,z),xlabel('x'),ylabel('z')
```

D.2.12 PLOTTING POLYNOMIALS

The `polyval(a,x)` function evaluates a polynomial at specified values of its independent variable `x`. The polynomial's coefficient array is `a`. The result is the same size as `x`. The `polyval` function is very useful for plotting polynomials. To do this you should define an array that contains many values of the independent variable x in order to obtain a smooth plot. For example, to plot the polynomial $f(x) = 9x^3 - 5x^2 + 3x + 7$ for $-2 \leq x \leq 5$, the session is

```
>>a = [9,-5,3,7]; x = [-2:0.01:5];
>>f = polyval(a,x);
>>plot(x,f),xlabel('x'),ylabel('f(x)'),grid
```

The `grid` command puts grid lines on the plot.

D.2.13 THE PLOT EDITOR

The aforementioned plot formatting commands are useful when you want to generate plots automatically with a program file. This is most convenient when you need to make many plots of similar type. If you need a small number of plots, you can use the Plot Editor to place text, lines, and arrows on the plot. Once the plot is displayed, click on the northwest-facing arrow to start the Plot Editor mode. This mode enables you to

edit the plot elements by double-clicking on an element, such as an axis or a plotted curve. To insert an axis label or to change the limits or scale, double-click on any axis, then select the tab of the desired axis, enter its label, and change its scale and limits, if desired.

Click on the Data Cursor button to use the mouse to extract coordinates from a plotted curve. Click on the Show Plot Tools button to display the Figure Palette. This palette enables you to add elements such as arrows, text, rectangles, and ellipses to the plot.

D.3 WORKING WITH FILES

MATLAB uses several types of files that enable you to save programs, data, and user-defined functions. MATLAB program files and function files are saved with the extension '`.m`', and thus are called M-files. Files are stored in *directories,* which are called *folders* on some computer systems. Directories can have subdirectories below them. The *path* tells us and MATLAB how to find a particular file.

D.3.1 THE SEARCH PATH

Suppose you have saved the program file `problem6.m` in the main directory of a memory stick that you insert in drive J:. The path for this file is J:\. As MATLAB is normally installed, when you type `problem6` in the Command window, MATLAB looks in the current directory for a program file named `problem6.m` and executes `problem6` if it finds it.

If `problem6.m` is in directory J:\ and if directory `J:\` is not in the search path, MATLAB will not find the file and will generate an error message, unless you tell it where to look. You can do this by typing `cd J:`, which stands for "change directory to J:." This will force MATLAB to look in that directory to find your file. The general syntax of this command is `cd dirname`, where `dirname` is the full path to the directory.

You can determine the current directory (the one where MATLAB looks for your file) by typing `pwd`. To see a list of all the files in the current directory, type `dir`. To see the files in the directory `dirname`, type `dir dirname`. The `what` command displays a list of the MATLAB-specific files in the current directory. The `what dirname` command does the same for the directory `dirname`.

D.3.2 SCRIPT FILES AND THE EDITOR/DEBUGGER

In addition to using the interactive mode, in which all commands are entered directly in the Command window, you can also perform operations by running a MATLAB program stored in *script* file. This type of file has MATLAB commands in it, so that when you run such a file it is equivalent to typing all the commands—one at a time—at the Command window prompt. You can run the file by typing its name at the Command window prompt. When the problem requires many commands, or has arrays with many elements, or you need to repeat a set of commands several times, it is more convenient to use a script file. If you need to access the same set of data frequently, you can store the data in an array within a script file.

You write and save MATLAB programs in M-files, which have the extension '`.m`'; for example, `program1.m`. You execute a script file at the Command window prompt by typing its name without the extension '`.m`'. Another type of M-file is a *function file,* which is useful when you need to repeat the operation of a set of commands. You can create your own function files; at the end of this section we discuss how to do this.

The symbol % designates a comment, which is not executed by MATLAB. Comments are put in script files for the purpose of documenting the file. The comment symbol may be put anywhere in the line. MATLAB ignores everything to the right of the % symbol. A very simple example of a script file is shown next. It computes the sine of the square root of several numbers and displays the results on the screen.

```
% Program example1.m
% This program computes the sine of the square root,
% and displays the result.
x = sqrt([3:2:11]);
y = sin(x)
```

To create this new M-file in the MS Windows environment, in the Command window select **New Script**. You will then see a new Editor window. Type in the file as just shown. You can use the keyboard. When finished, select **Save** from the toolbar. The Editor will automatically provide the extension '.m' and save the file in the MATLAB current directory, which for now we will assume to be on the hard drive.

Once the file has been saved, in the MATLAB Command window type the script file's name `example1` to execute the program. You should see displayed in the Command window the computed values of the array `y = 0.9870 0.7867 0.4758 0.1411 -0.1741`.

Keep in mind the following when using script files:

1. The name of a script file must follow the MATLAB convention for naming variables; that is, the name must begin with a letter, and may include digits and the underscore character, up to 31 characters.
2. Recall that typing a variable's name at the Command window prompt causes MATLAB to display the value of that variable. Thus, do not give a script file the same name as a variable it computes, because MATLAB will not be able to execute that script file more than once, unless you clear the variable.
3. Do not give a script file the same name as a MATLAB command or function. You can check to see if a command, function, or file name already exists by using the `exist` command.
4. You can use the `type` command to view an M-file without opening it with a text editor. For example, to view the file `example1`, the command is `type example1`.

D.3.3 INPUT AND OUTPUT FOR SCRIPT FILES

You can use the `disp` function to display the value of a variable from within a script file. For example, type `disp(x)` to display the value of `x`. You can also display messages with the `disp` function. Type `disp('text')` to display the message enclosed within single quotes. The `input` function displays a prompt and assigns an entered value to a variable. The syntax is `x = input('Enter a value for x:')`. The function displays the message within the single quotes and assigns the value typed at the keyboard to the variable `x`. Of course, any variable name and any text message can be used.

D.3.4 DEBUGGING PROGRAMS

Debugging a program is the process of finding and removing the "bugs," or errors, in a program. Such errors usually fall into one of two categories: (1) Syntax errors such as omitting a parenthesis or comma, or spelling a command name incorrectly. MATLAB

usually detects the more obvious errors and displays a message describing the error and its location. (2) Errors due to an incorrect mathematical procedure. These are called *runtime errors*. They do not necessarily occur every time the program is executed; their occurrence often depends on the particular input data. A common example is division by zero.

The MATLAB error messages usually enable you to find syntax errors. However, runtime errors are more difficult to locate. To locate such an error, try the following: (1) Test your program with a simple version of the problem, whose answers can be checked by hand calculations. (2) Display any intermediate calculations by removing semicolons at the end of statements. (3) Use the debugging features of the Debugger. However, one of advantages of MATLAB is that it requires relatively simple programs to accomplish many types of tasks. Thus, you probably will not need to use the Debugger for most of the problems encountered in this text.

D.3.5 USER-DEFINED FUNCTIONS

Another type of M-file is a *function file*. Unlike a script file, all the variables in a function file are *local,* which means their values are available only within the function. Function files are useful when you need to repeat a set of commands several times. The first line in a function file must begin with a *function definition line* that has a list of inputs and outputs. This line distinguishes a function M-file from a script M-file. Its syntax is as follows:

```
function [output variables] = function_name(input variables)
```

Note that the output variables are enclosed in *square brackets* while the input variables must be enclosed with *parentheses*. The `function_name` must be the same as the filename in which it is saved (with the .m extension). That is, if we name a function `drop`, it must be saved in the file `drop.m`. The function is "called" by typing its name (e.g., `drop`) at the command line. The word `function` in the function definition line must be *lower case*.

The following examples show permissible variations in the format of the function line. The differences depend on whether there is no output, a single output, or multiple outputs.

Function definition line	File name
1. `function [area_square] = square(side);`	square.m
2. `function area_square = square(side);`	square.m
3. `function [volume_box] = box(height,width,length);`	box.m
4. `function [area_circle,circumf] = circle(radius);`	circle.m
5. `function sqplot(side);`	sqplot.m

Example 1 is a function with one input and one output. The square brackets are optional when there is only one output (see Example 2). Example 3 has one output and three inputs. Example 4 has two outputs and one input. Example 5 has no output variable (an example of this would be a function that generates a plot). In such cases, the equals sign should be omitted.

Comment lines starting with the % sign may be put anywhere. However, it is important to note that all comment lines immediately following the function definition line are displayed by MATLAB if `help` is used to obtain information about the function. The first comment line can be accessed by the `lookfor` command, which is discussed in Section D.5.

Both built-in and user-defined functions can be called either with the output variables explicitly specified, as with the previous Examples 1 through 4, or without any output variables specified. For example, we can call the function `square` as follows: `square(side)` if we are not interested in accessing its output variable `area_square` (the function might perform some other operation that we want to occur, such as displaying a result). Note that if the semicolon is omitted at the end of the function call statement, the first variable in the output variable list will be displayed using the default variable name `ans`.

For example, the following function, called `drop`, computes a falling object's velocity and distance dropped. The input variables are the acceleration g, the initial velocity v_0, and the elapsed time t. Note that we must use the element-by-element operations for any operations involving function inputs that are arrays. Here we anticipate that `t` will be an array, so we use the element-by-element operator (`.^`).

```
function [dist,vel] = drop(g,vO,t);
% Computes velocity and distance of a dropped object.
% The distance and velocity are computed as functions
% of g, the initial velocity vO, and the time t.
vel = g*t + vO;
dist = 0.5*g*t.^2 + vO*t;
```

The following examples show various ways the function `drop` can be called.

1. The variable names used in the function definition may, but need not, be used when the function is called. For example, consider the session:

```
>>a = 32.2;
>>initial_speed = 10;
>>time = 5;
>>[feet_dropped,speed] = drop(a,initial_speed,time)
```

2. The input variables need not be assigned values outside the function prior to the function call. For example,

```
>>[feet_dropped,speed] = drop(32.2,10,5)
```

3. The inputs and outputs may be arrays. For example,

```
>>[feet_dropped,speed] = drop(32.2,10,[0:1:5])
```

produces the arrays `feet_dropped` and `speed` each with six values, corresponding to the six values of time in the array `time`.

Some of the MATLAB commands act on functions, so if the function of interest is not a simple function, you must define the function in an M-file to use one of these commands. For example, we can use the `fzero` function to find the zero of a function of a single variable, which we denote by `x`. One form of its syntax is `fzero('function', x0)`, where `function` is a string containing the name of the function. The `fzero` function returns a value of `x` that is near `x0`. It identifies only points where the function crosses the x axis, not points where the function just touches the axis. If $y = x + 2e^{-x} - 3$, define the following function file:

```
function y = f1(x)
y = x + 2*exp(-x) - 3;
```

To find the zero near $x = 3$, type `x = fzero('f1',3)`. The returned answer is $x = 2.8887$.

The `fminbnd` function can be used to find the minimum of a function of a single variable, which we denote by `x`. One form of its syntax is `fminbnd('function', x1, x2)`, where `function` is a string containing the name of the function. The `fminbnd` function returns a value of `x` that minimizes the function in the interval $x1 \leq x \leq x2$. For example, if $y = 1 - xe^{-x}$, define the following function file:

```
function y = f2(x)
y = 1-x.*exp(-x);
```

To find the value of x that gives a minimum for $0 \leq x \leq 5$, type `x = fmin('f2',0,5)`. The returned answer is $x = 1$. To find the minimum value of y, type `y = f2(x)`. The result is `y = 0.6321`.

D.4 LOGICAL OPERATORS AND LOOPS

MATLAB has relational operators, logical operators, and conditional statements that enable you to write decision-making programs whose operations depend on the results of calculations made by the program. MATLAB also has two types of loop structures for performing calculations repeatedly a specified number of times or until some condition is satisfied.

D.4.1 RELATIONAL OPERATORS

MATLAB has six *relational* operators to make comparisons between arrays. These operators are shown in Table D.4.1. Note that the "equal to" operator consists of two = signs, not a single = sign as you might expect. The single = sign is the *assignment* operator in MATLAB. The result of a comparison using the relational operators is either a 0 (if the comparison is *false*), or a 1 (if the comparison is *true*), and the result can be used as a variable.

For example, if `x = 2` and `y = 5`, typing `z = x < y` returns the value `z = 1`, because `x` is less than `y`. Typing `u = x==y` returns the value `u = 0` because `x` does not equal `y`. To make the statements more readable, we can group the operations using parentheses. For example, `z = (x < y)` and `u = (x==y)`. Note, however, that the variables `z` and `u` are *logical* variables, as opposed to *numeric* variables. A logical variable may take on only the logical values, true or false. These values are represented by 1 and 0, but they should not be used for numerical calculations. For example, typing `b = sin(z)` generates an error message. This distinction is discussed in more detail in the subsection "Logical Arrays."

When used to compare arrays, the relational operators compare the arrays on an element-by-element basis. The arrays being compared must have the same size. The

Table D.4.1 Relational operators.

Relational operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

only exception occurs when we compare an array to a scalar. In that case, all the elements of the array are compared to the scalar. For example, suppose that `x = [6, 3, 9]` and `y = [14, 2, 9]`. Then typing `z = (x < y)` gives the result `z = [1, 0, 0]`. Typing `z = (x == y)` gives the result `z = [0, 0, 1]`. Typing `z = (x > 8)` gives `z = [0, 0, 1]`.

The relational operators can be used for array addressing. For example, with `x = [6, 3, 9, 11]` and `y = [14, 2, 9, 13]`, typing `z = x(x < y)` finds all the elements in `x` that are less than the corresponding elements in `y`. The result is the array `z = [6, 11]`.

D.4.2 LOGICAL OPERATORS

MATLAB has three *logical* operators, which are sometimes called *Boolean* operators (see Table D.4.2). These operators perform element-by-element operations. With the exception of the NOT operator (~), they have a lower precedence than the arithmetic and relational operators (see Table D.4.3). The NOT symbol is called the *tilde*.

The NOT operation `~A` returns an array of the same dimension as `A`, having 1's where `A` is 0, and 0's where `A` is nonzero. For example, if `x = [6, 3, 9]` and `y = [14, 2, 9]`, then the statement `z = ~x` returns the array `z = [0, 0, 0]`, and the statement `z = ~x > y` returns the result `z = [0, 0, 0]`. The later statement is equivalent to `z = (~x) > y`, whereas `z = ~(x > y)` gives the result `z = [1, 0, 1]`. This is equivalent to `z = (x <= y)`.

The `&` and `|` operators compare two arrays of the same dimension. The only exception, as with the relational operators, is that an array can be compared to a scalar. The AND operator `&` returns 1's where both `A` and `B` have nonzero elements, and 0's where any element of `A` or `B` is zero. The expression `z = 0&3` returns `z = 0`, `z = 2&3` returns `z = 1`, `z = 0&0` returns `z = 0`, and `z = [5, -3, 0, 0]&[2, 4, 0, 5]` returns `z = [1, 1, 0, 0]`. Because of operator precedence, `z = 1&2 + 3` is equivalent to `z = 1&(2 + 3)`, which returns `z = 1`. Similarly, `z = 5 < 6&1` is equivalent to `z = (5 < 6)&1`, which returns `z = 1`.

Table D.4.2 Logical operators.

Operator	Name	Definition
~	NOT	~A returns an array the same dimension as A, which has 1's where A is 0, and 0's where A is nonzero.
&	AND	A & B returns an array the same dimension as A and B, which has 1's where both A and B have nonzero elements, and 0's where either A or B is zero.
|	OR	A|B returns an array the same dimension as A and B, which has 1's where at least one element in A or B is nonzero, and 0's where A and B are both zero.

Table D.4.3 Order of precedence for operator types.

Precedence	Operator type
First	Parentheses, evaluated starting with the innermost pair.
Second	Arithmetic operators and NOT (~), evaluated from left to right.
Third	Relational operators, evaluated from left to right.
Fourth	Logical AND.
Fifth	Logical OR.

Table D.4.4 Truth table for logical operators.

x	y	~x	x|y	x&y
true	true	false	true	true
true	false	false	true	false
false	true	true	true	false
false	false	true	false	false

Let `x = [6, 3, 9]` and `y = [14, 2, 9]`, and let `a = [4, 3, 12]`. The expression

```
(x > y) & a
```

gives `z = [0, 1, 0]`, and the expression

```
z = (x > y) & (x > a)
```

returns the result `z = [0, 0, 0]`.

Be careful when using the logical operators with inequalities. For example, note that `~(x > y)` is equivalent to `y >= x`. It is *not* equivalent to `y > x`. As another example, the relation $5 < x < 10$ must be written as

```
(5 < x) & (x < 10)
```

in MATLAB.

The OR operator `|` returns 1's where at least one of `A` and `B` has nonzero elements, and 0's where both `A` and `B` are zero. The expression `z = 0|3` returns `z = 1`, the expression `z = 0|0` returns `z = 0`, and the expression

```
z = [5, -3, 0, 0]|[2, 4, 0, 5]
```

returns `z = [1, 1, 0, 1]`. Because of operator precedence, the expression `z = 3 < 5|4 == 7` is equivalent to `z = (3 < 5)|(4 == 7)`, which returns `z = 1`. Similarly, the expression `z = 1|0 & 1` is equivalent to `z = (1|0)&1`, which returns `z = 1`, while the expression `z = 1|0 & 0` returns `z = 0`, and the expression `z = 0 & 0|1` returns `z = 1`.

Because of the precedence of the NOT operator the statement `z = ~3 == 7|4 == 6` returns the result `z = 0`, which is equivalent to `z = ((~3) == 7)|(4 == 6)`.

Table D.4.4 is a so-called *truth table* that defines the operations of the logical operators, where "true" is equivalent to 1, and "false" is equivalent to 0. Let `x` and `y` represent the first two columns of the truth table in terms of 1's and 0's. The following MATLAB session generates the truth table in terms of 1's and 0's.

```
>> x = [1, 1, 0, 0]'; y = [1, 0, 1, 0]';
>> Truth_Table = [x,y,~x,x|y,x&y]
Truth_Table =
   1  1  0  1  1
   1  0  0  1  0
   0  1  1  1  0
   0  0  1  0  0
```

D.4.3 THE `find` FUNCTION

The `find` function is very useful for creating decision-making programs, especially when combined with the relational and logical operators. The function `find(x)`

computes an array containing the indices of the nonzero elements of the array x. For example, consider the session

```
>>x = [-2, 0, 4]; y = find(x)
y =
   1     3
```

The resulting array `y = [1, 3]` indicates that the first and third elements of x are nonzero. Note that the `find` function returns the *indices,* and not the *values*. In the following session, note the difference between the result obtained by `x(x<y)` and the result obtained by `find(x < y)`.

```
>>x = [6, 3, 9, 11];y = [14, 2, 9, 13];
>>values = x(x < y)
values =
   6   11
>>indices = find(x < y)
indices =
   1     4
```

Thus, there are two values in the array x that are less than the *corresponding* values in the array y. They are the first and fourth values, which are 6 and 11.

The logical operators can be used with the `find` function. For example, consider the session

```
>>x = [5, -3, 0, 0, 8]; y = [2, 4, 0, 5, 7];
>>z = find(x & y)
z =
   1   2   5
>>values = y(x & y)
values =
   2   4   7
```

The array obtained with the `find` function, `z = [1, 2, 5]`, indicates that the first, second, and fifth elements of x and y are both nonzero. The array `values = [2, 4, 7]`, which is obtained from `y(x & y)`, shows that there are three nonzero values in y that correspond to nonzero values in x. These are 2, 4, and 7, the first, second, and fifth values.

D.4.4 LOGICAL ARRAYS

Logical variables can have only the values 1 (true) and 0 (false), and a logical array contains logical values. However, just because an array contains only 0's and 1's, it is not necessarily a logical array. For example, in the following session k and w appear the same, but k is a logical array and w is a numeric array, and thus an error message is issued.

```
>> x = [-2:2]
x =
   -2   -1   0   1   2
>> k = (abs(x)>1)
k =
     1   0    0    0    1
>> z = x(k)
```

```
z =
   -2    2
>> w = [1,0,0,0,1];
>> v = x(w)
??? Subscript indices must either be real positive...
   integers or logicals.
```

Logical arrays can be created with the relational and logical operators and with the `logical` function. The `logical` function returns an array that can be used for logical indexing and logical tests. Typing `B = logical(A)`, where `A` is a numeric array, returns the logical array `B`. So to correct the error in the previous session, you can type instead `w = logical([1,0,0,0,1])`.

You can use the `double` function to convert a logical array to an array of numerical 0's and 1's of class `double`. For example, `x = (5>3); y = double(x);`. In addition, most arithmetic operations convert a logical array to a double array.

D.4.5 ACCESSING ARRAYS

Typing `A(B)`, where `B` is a logical array of the same size as `A`, returns the values of `A` at the indices where `B` is nonzero. Given `A = [5,6,7;8,9,10;11,12,13]` and `B = logical(eye(3))`, we can extract the diagonal elements of `A` by typing `C = A(B)` to obtain `C = [5;9;13]`. Specifying array subscripts with logical arrays extracts the elements that correspond to the true (1) elements in the logical array.

Note, however, that using the *numeric* array `eye(3)`, as `C = A(eye(3))` results in an error message because the elements of `eye(3)` do not correspond to locations in `A`. If the numeric array values correspond to valid locations, you may use a numeric array to extract the elements. For example, to extract the diagonal elements of `A` with a numeric array, type `C = A([1,5,9])`.

D.4.6 CONDITIONAL STATEMENTS

The MATLAB *conditional* statements enable us to write programs that make decisions. Conditional statements contain one or more of the `if`, `else`, and `elseif` statements. The `end` statement is used to denote the end of a conditional statement. These conditional statements read somewhat like their English language equivalents. For example, suppose that x is a scalar, and that we want to compute $y = \sqrt{x}$ only if $x \geq 0$. In English, we could specify this as: "If x is greater than or equal to zero, compute y from $y = \sqrt{x}$, otherwise, do nothing." The `if` statement in the next script file accomplishes this in MATLAB, assuming that the variable `x` already has a scalar value.

```
if x >= 0
   y = sqrt(x)
end
```

If `x` is negative, the program takes no action.

When more than one action must occur as a result of a decision, we can use the `else` and `elseif` statements. The statements after the `else` are executed if all the preceding `if` and `elseif` expressions are false. The general form of the `if` statement is

`if` *logical expression*
statement group 1

```
elseif logical expression
   statement group 2
else
   statement group 3
end
```

There can be more than one `elseif` statement. The `else` and `elseif` statements can be omitted if not required. However, if both are used, the `else` statement must come after the `elseif` statements to take care of all conditions that might be unaccounted for.

Suppose that we want to compute y from $y = \sqrt{x}$ for $x \geq 0$, and $y = -\sqrt{-x}$ for $x < 0$. The next script file will calculate y, assuming that the variable `x` already has a scalar value.

```
if x >= 0
   y = sqrt(x)
else
   y = -sqrt(-x)
end
```

As another example, suppose that we want to compute y such that $y = 0$ if $x < 0$, $y = 10x$ if $0 \leq x < 25$, and $y = 50\sqrt{x}$ if $x \geq 25$. The next script file will compute y, assuming that the variable `x` already has a scalar value.

```
if x >= 25
   y = 50*sqrt(x)
elseif x >= 0
   y = 10*x
else
   y = 0
end
```

Note that the `elseif` statement does not require a separate `end` statement.

When the test, `if` *logical expression,* is performed, where the logical expression may be an array, the test returns a value of true only if *all* the elements of the logical expression are true! For example, if we fail to grasp this, the following statements do not perform the way we might expect.

```
x = [4, -9, 25];
if x < 0
   disp('Some of the elements of x are negative.')
else
   y = sqrt(x)
end
```

When this program is run it gives the result `y = 2, 0 + 3.000i, 5`. The program does not compute the square root of each element in `x` in sequence. Instead it tests the truth of the relation `x < 0`. The test `if x < 0` returns a false value because it generates the vector `[0, 1, 0]`. Compare the last program with the following statements:

```
x=[4, -9, 25];
if x >= 0
   y = sqrt(x)
```

```
else
   disp('Some of the elements of x are negative.')
end
```

When executed, it produces the result: `Some of the elements of x are negative.` The difference is due to the different conditional statements used. Here the test, `if x < 0`, is true, whereas the test, `if x >= 0`, returns a false value because `x >= 0` returns the vector `[1, 0, 1]`.

Conditional structures can be nested; that is, one structure can contain another structure.

D.4.7 THE `for` LOOP

A *loop* is a structure for repeating a calculation a number of times. Each repetition of the loop is a *pass*. There are two types of explicit loops in MATLAB: the `for` loop, which is used when the number of passes is known or can be computed ahead of time, and the `while` loop, which is used when the looping process must terminate when a specified condition is satisfied, and thus the number of passes is not known or computable in advance.

A simple example of a `for` loop is

```
m = 0;
x(1) = 10;
for k = 2:3:11
   m = m+1;
   x(m+1) = x(m) + k^2;
end
```

The *loop variable* `k` is initially assigned the value 2. During each successive pass through the loop `k` is incremented by 3, and `x` is calculated until `k` exceeds 11. Thus, `k` takes on the values 2, 5, 8, and 11. The variable `m` indicates the index of the array `x`. The program then continues to execute any statements following the `end` statement. When the loop is finished the array `x` will have the values `x(1)=14`, `x(2)=39`, `x(3)=103`, `x(4)=224`. The name of the loop variable need not be `k`.

Note the following rules when using `for` loops with the loop variable expression `k = m:s:n`:

1. The step value `s` can be negative. For example, `k = 10:-2:4` produces `k` = 10, 8, 6, 4.
2. If `s` is omitted, the step value defaults to 1.
3. If `s` is positive, the loop will not be executed if `m` is greater than `n`.
4. If `s` is negative, the loop will not be executed if `m` is less than `n`.
5. If `m` equals `n`, the loop will be executed only once.
6. If the step value `s` is not an integer, roundoff errors can cause the loop to execute a different number of passes than intended.

You should not alter the value of the loop variable `k` within the *statements*. Doing so can cause unpredictable results.

D.4.8 THE `while` LOOP

The `while` loop is used in cases where the looping process must terminate when a specified condition is satisfied, and thus the number of passes is not known or computable in advance. The basic structure of a `while` loop is the following.

```
while logical expression
   statement group
end
```

MATLAB first tests the truth of the *logical expression,* which must contain a loop variable. If the *logical expression* is true, the statements in the *statement group* will be executed. For the `while` loop to function properly, the following two conditions are required:

1. The loop variable must have a value before the `while` statement is executed.
2. The loop variable must be changed somehow by the statements in the *statement group*.

The statements in the *statement group* are executed once during each pass, using the current value of the loop variable. The looping continues until the *logical expression* is false.

A simple example of a `while` loop is

```
x = 5; k = 0;
while x < 25
   k = k + 1;
   y(k) = 3*x;
   x = 2*x - 1;
end
```

The loop variable `x` is initially assigned the value 5, and it has this value until the statement `x = 2*x - 1` is encountered the first time. Its value then changes to 9. Before each pass through the loop, `x` is checked to see if its value is less than 25. If so, the pass is made. If not, the loop is skipped and the program continues to execute any statements following the `end` statement. The variable `x` takes on the values 9, 17, and 33 within the loop. The resulting array `y` contains the values `y(1) = 15`, `y(2) = 27`, and `y(3) = 51`.

Every `for` and `while` statement must be matched by an accompanying `end` statement. As with conditional statements, loops can be nested.

D.5 THE MATLAB HELP SYSTEM

Two principal ways to obtain more information about MATLAB commands and features are to use the help commands in the Command window and to access help by clicking on the Help Button. The help commands can be used to display syntax information for a specified topic in the Command window. For additional help, you can run demos, contact technical support, search documentation for other MathWorks products, view a list of other books, and participate in a newsgroup.

D.5.1 HELP COMMANDS

Four MATLAB commands are useful for obtaining online information about MATLAB topics.

- Typing `help topic` displays in the Command window a description of the specified `topic`.
- Typing `helpwin topic` displays the help text for the specified `topic` inside the Help Browser window.
- Typing `lookfor topic` displays in the Command window a brief description for all items whose description includes the specified keyword `topic`.
- Typing `doc topic` opens the Help Browser to the reference page for the specified topic, providing a description, additional remarks, and examples.

The `help` command is the basic way to determine the syntax of a particular command, function, or statement. For example, typing `help log10` produces the following display:

```
LOG10 Common (base 10) logarithm.
   LOG10(X) is the base 10 logarithm of the elements of X.
   Complex results are produced if X is not positive.

   See also LOG, LOG2, EXP, LOGM.
```

Note that the display describes what the function does, warns about any unexpected results if nonstandard argument values are used, and directs the user to other related functions.

All the MATLAB functions are organized into groups, and the MATLAB directory structure is based on this grouping. For instance, all elementary mathematical functions such as `log10` reside in the `elfun` directory, and the polynomial functions reside in the `polyfun` directory. To list the names of all the functions in that directory, with a brief description of each, type `help polyfun`. If you are unsure of what directory to search, type `help` to obtain a list all the directories, with a description of the function category each represents.

Typing `helpwin topic` displays the help text for the specified `topic` inside the Help Browser window. Links are created to functions referenced in the "See also" line of the help text. You can also access the Help Window by selecting the **Help** option under the **Help** menu, or by clicking the question mark button on the toolbar.

The `lookfor` command enables you to search for topics based on a keyword. It searches through the first line of help text, which is known as the H1 line, for each MATLAB function, and returns all the H1 lines containing a specified keyword. For example, MATLAB does not have a function named `sine`. So the response from `help sine` is `sine.m not found`. However, typing `lookfor sine` produces over a dozen matches, depending on which toolboxes you have installed. For example, some of the matches you will see are

```
ACOS      Inverse cosine.
ACOSH     Inverse hyperbolic cosine.
ASIN      Inverse sine.
ASINH     Inverse hyperbolic sine.
COS       Cosine.
COSH      Hyperbolic cosine.
```

```
SIN       Sine.
SINH      Hyperbolic sine.
```

From this list you can find the correct name for the sine function. Note that all words containing sine are returned, such as cosine. Adding `-all` to the lookfor command searches the entire help entry, not just the H1 line.

Typing `doc topic` displays the documentation for the topic in the Help Browser. For example, typing `doc step` displays information about the function `step`. Typing `doc toolbox` displays the documentation road map page for the specified toolbox. For example, typing `doc control` displays information about the Control Systems toolbox.

To open the Help menu, click on the **Help** button on the toolbar, or click the question mark button in the toolbar. You will see a menu that lets your search documentation and see examples.

D.5.2 MATHWORKS WEBSITE

The MathWorks maintains a very useful website that you can use to ask questions, make suggestions, and report possible bugs. You can also search for solutions by querying an up-to-date database of technical support information. You can access the website by clicking on Support **Web site** under the **Help** menu in the Desktop.

REFERENCES

[Palm, 2011] W. J. Palm III *Introduction to MATLAB for Engineers*, 3rd ed. McGraw-Hill, New York, 2011.

[Palm, 2008] W. J. Palm III *A Concise Introduction to MATLAB*, McGraw-Hill, New York, 2008.

PROBLEMS

D.1 Assuming that the variables a, b, c, d, and f are scalars, write MATLAB statements to compute and display the following expressions. Test your statements for the values $a = 1.12$, $b = 2.34$, $c = 0.72$, $d = 0.81$, and $f = 19.83$.

$$x = 1 + \frac{a}{b} + \frac{c}{f^2} \qquad s = \frac{b-a}{d-c}$$

$$r = \frac{1}{1/a + 1/b + 1/c + 1/d} \qquad y = ab\frac{1}{c}\frac{f^2}{2}$$

D.2 Suppose that $x = -7 - 5j$ and $y = 4 + 3j$. Use MATLAB to compute

a. $x + y$ b. xy c. x/y d. $\frac{3}{2}j$ e. $\frac{3}{2j}$

D.3 Suppose x takes on the values $x = 1, 1.2, 1.4, \ldots, 5$. Use MATLAB to compute the array `y` that results from the function $y = 7\sin(4x)$. Use MATLAB to determine how many elements are in the array `y`, and the value of the third element in the array `y`.

D.4 The following table shows the hourly wages, hours worked, and output (number of widgets produced) in 1 week for five widget makers.

	Worker				
	1	2	3	4	5
Hourly wage ($)	5	5.50	6.50	6	6.25
Hours worked	40	43	37	50	45
Output (widgets)	1000	1100	1000	1200	1100

Use MATLAB to answer these questions:

a. How much did each worker earn in the week?
b. What is the total salary amount paid out?
c. How many widgets were made?
d. What is the average cost to produce one widget?
e. How many hours does it take to produce one widget, on average?
f. Assuming that the output of each worker has the same quality, which worker is the most efficient? Which is the least efficient?

D.5 Write a MATLAB assignment statement for each of the following functions, assuming that w, x, y, and z are vector quantities of equal length, and that c and d are scalars.

$$f = \frac{1}{\sqrt{(2\pi c)/x}} \qquad E = \frac{x + w/(y+z)}{x + w/(y-z)}$$

$$A = \frac{e^{-c/(2x)}}{(\ln\, y)\sqrt{dz}} \qquad S = \frac{x(2.15 + 0.35y)^{1.8}}{z(1-x)^{y}}$$

D.6 The following tables show the costs associated with a certain product, and the production volume for the four quarters of the business year. Use MATLAB to find (a) the quarterly costs for materials, labor, and transportation; (b) the total material, labor, and transportation costs for the year; and (c) the total quarterly costs.

	Unit product costs ($\$ \times 10^3$)		
Product	Materials	Labor	Transportation
1	7	3	2
2	3	1	3
3	9	4	5
4	2	5	4
5	6	2	1

	Quarterly production volume			
Product	First quarter	Second quarter	Third quarter	Fourth quarter
1	16	14	10	12
2	12	15	11	13
3	8	9	7	11
4	14	13	15	17
5	13	16	12	18

D.7 Use MATLAB to find the roots of $13s^3 + 182s^2 - 184s + 2503 = 0$, and use `poly` to confirm your answer.

D.8 Use MATLAB to find the roots of the polynomial $36s^3 + 12s^2 - 5s + 10$. Use the `polyval` function to verify the solution.

D.9 Use MATLAB to find the polynomial whose roots are $3 \pm 6j$, 8, 8, and 20. Use MATLAB to confirm your answer.

D.10 Use MATLAB to find the following product:

$$\left(10s^3 - 9s^2 - 6s + 12\right)\left(5s^3 - 4s^2 - 12s + 8\right)$$

D.11 Use MATLAB to find the quotient and remainder of

$$\frac{14s^3 - 6s^2 + 3s + 9}{5s^2 + 7s - 4}$$

D.12 Use MATLAB to plot the functions $u = 2\log_{10}(60x + 1)$ and $v = 3\cos(6x)$ over the interval $0 \leq x \leq 2$. Properly label the plot and each curve. The variables u and v represent speed in miles per hour; the variable x represents distance in miles.

D.13 Use MATLAB to plot the polynomials $y = 3x^4 - 6x^3 + 8x^2 + 4x + 90$ and $z = 3x^3 + 5x^2 - 8x + 70$ over the interval $-3 \leq x \leq 3$. Properly label the plot and each curve. The variables y and z represent current in milliamps; the variable x represents voltage in volts.

D.14 Write a script file using conditional statements to evaluate the following function, assuming that the scalar variable x has a value. The function is $y = e^{x+1}$ for $x < -1$, $y = 2 + \cos(\pi x)$ for $-1 \leq x < 5$, and $y = 10(x - 5) + 1$ for $x \geq 5$. Use your file to evaluate y for $x = -5$, $x = 3$, and $x = 15$, and check the results by hand.

D.15 Suppose that `x = [-15,-8,9,8,5]` and `y = [-20,12,-4,8,9]`. What is the result of the following operations? Determine the answers by hand, and then use MATLAB to check your answers.

a. `z = (x < y)`

b. `z = (x > y)`

c. `z = (x ~= y)`

d. `z = (x == y)`

e. `z = (y > -4)`

D.16 The given arrays `price_A`, `price_B` and `price_C` contain the price in dollars of three stocks over ten days.

a. Use MATLAB to determine how many days the price of stock A was above both the price of stock B and the price of stock C.

b. Use MATLAB to determine how many days the price of stock A was above either the price of stock B or the price of stock C.

c. Use MATLAB to determine how many days the price of stock A was above either the price of stock B or the price of stock C, but not both.

```
price_A = [19, 18, 22, 21, 25, 19, 17, 21, 27, 29]
price_B = [22, 17, 20, 19, 24, 18, 16, 25, 28, 27]
price_C = [17, 13, 22, 23, 19, 17, 20, 21, 24, 28]
```

D.17 The price, in dollars, of a certain stock over a ten day period is given in the following array.

```
price = [19, 18, 22, 21, 25, 19, 17, 21, 27, 29]
```

Suppose you owned 1000 shares at the start of the ten day period, and you bought 100 shares every day the price was below \$20, and sold 100 shares every day the price was above \$25. Use MATLAB to compute (a) the amount you spent in buying shares, (b) the amount you received from the sale of shares, (c) the total number of shares you own after the tenth day, and (d) the net increase in the worth of your portfolio of stock.

D.18 Use a `for` loop to determine the sum of the first 10 terms in the series $5k^3$, $k = 1, 2, 3, \ldots, 10$.

D.19 Use a `while` loop to determine how many terms in the series 2^k, $k = 1, 2, 3, \ldots,$ are required for the sum of the terms to exceed 2000. What is the sum for this number of terms?

D.20 One bank pays 5.5% annual interest, while a second bank pays 4.5% annual interest. Determine how much longer it will take to accumulate at least \$50,000 in the second bank account if you deposit \$1000 initially, and \$1000 at the end of each year.

D.21 Given a number x and the quadrant q ($q = 1, 2, 3, 4$), write a program to compute $\sin^{-1}(x)$ in degrees, taking into account the quadrant. The program should display an error message if $|x| > 1$.

D.22 The function $y = 1 + e^{-0.2x} \sin(x + 2)$ has two minimum points in the interval $0 < x < 10$. Create a user-defined function and use the `fminbnd` function to find the values of x and y at each minimum.

D.23 An object thrown vertically with a speed v_0 will reach a height h at time t, where

$$h = v_0 t - \frac{1}{2} g t^2$$

Write and test a function that computes the time t required to reach a specified height h, for a given value of v_0. The function's inputs should be h, v_0, and g. Test your function for the case where $h = 100$ m, $v_0 = 50$ m/s, and $g = 9.81$ m/s^2. Interpret both answers.