UNIT I **MATLAB ENVIRONMENT** **10 hrs**

Defining Variables – functions – Matrices and Vectors –Strings – Input and Output statements -Script files – Arrays in Mat lab – Addressing Arrays – Dynamic Array – Cell Array – Structure Array – File input and output – Opening & Closing – Writing & Reading data from files.

UNIT I

MATLAB® is a very powerful software package that has many built-in tools for solving problems and for graphical illustrations. The simplest method for using the MATLAB product is interactively; an expression is entered by the user and MATLAB immediately responds with a result. It is also possible to write programs in MATLAB, which are essentially groups of commands that are executed sequentially. This chapter will focus on the basics, including many operators and built-in functions that can be used in interactive expressions. Means of storing values, including vectors and matrices, will also be introduced.

Getting into MAT LAB

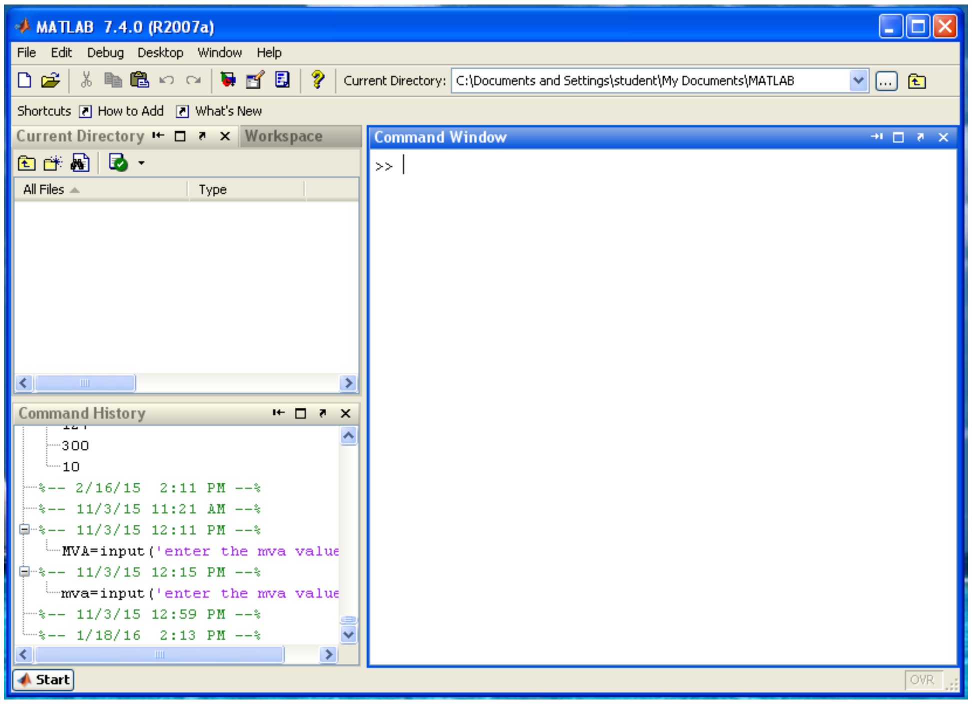
MATLAB is a mathematical and graphical software package; it has numerical, graphical, and programming capabilities. It has built-in functions to do many operations, and there are toolboxes that can be added to augment these functions (e.g., for signal processing). There are versions available for different hardware platforms, and there are both professional and student editions. When the MATLAB software is started, a window is opened: the main part is the Command Window (see Figure 1.1). In the Command Window, there is a statement that says:

In the Command Window, you should see:

```
>>
```

The >> is called the prompt. In the Student Edition, the prompt appears as:

```
EDU>>
```


MATLAB 7.4.0 (R2007a)
File Edit Debug Desktop Window Help
Current Directory: C:\Documents and Settings\student\My Documents\MATLAB
Shortcuts How to Add What's New
Current Directory Workspace
All Files Type
Command History
Command Window
Start
OVR

```
>> |
```

```
300
10
%-- 2/16/15  2:11 PM --%
%-- 11/3/15 11:21 AM --%
%-- 11/3/15 12:11 PM --%
MVA=input('enter the mva value
%-- 11/3/15 12:15 PM --%
mva=input('enter the mva value
%-- 11/3/15 12:59 PM --%
%-- 1/18/16  2:13 PM --%
```

Figure 1.1 MATLAB Window

In the Command Window, MATLAB can be used interactively. At the prompt, any MATLAB command or expression can be entered, and MATLAB will immediately respond with the result. It is also possible to write ***programs*** in MATLAB, which are contained in ***script files*** or M-files. There are several commands that can serve as an introduction to MATLAB and allow you to get help:

- **info** will display contact information for the product
- **demo** has demos of several options in MATLAB
- **help** will explain any command; **help help** will explain how help works
- **help browser** opens a Help Window
- **lookfor** searches through the help for a specific string (be aware that this can take a long time)

To get out of MATLAB, either type **quit** at the prompt, or chooses File, then Exit MATLAB from the menu. In addition to the Command Window, there are several other windows that can be opened and may be opened by default. What is described here is the default layout for these windows, although there are other possible configurations. Directly above the Command Window, there is a pull-down menu for the Current Directory. The folder that is set as the Current Directory is where files will be saved. By default, this is the Work Directory, but that can be changed.

To the left of the Command Window, there are two tabs for Current Directory Window and Workspace Window. If the Current Directory tab is chosen, the files stored in that directory are displayed. The Command History Window shows commands that have been entered, not just in the current session (in the current Command Window), but previously as well. This default configuration can be altered by clicking Desktop, or using the icons at the top-right corner of each window: either an “x,” which will close that particular window; or a curled arrow, which in its initial state pointing to the upper right lets you undock that window. Once undocked, clicking the curled arrow pointing to the lower right will dock the window again.

Variables and Assignment Statements

In order to store a value in a MATLAB session, or in a program, a ***variable*** is used. The Workspace Window shows variables that have been created. One easy way to create a variable is to use an ***assignment statement***. The format of an assignment statement is

variablename = expression

The variable is always on the left, followed by the ***assignment operator***, = (unlike in mathematics, the single equal sign does *not* mean equality), followed by an expression. The expression is evaluated and then that value is stored in the variable. For example, this is the way it would appear in the Command Window:

```
>> mynum = 6
mynum =
6
>>
```

Here, the ***user*** (the person working in MATLAB) typed mynum = 6 at the prompt, and MATLAB stored the integer 6 in the variable called *mynum*, and then displayed the result followed by the prompt again. Since the equal sign is the assignment operator, and does not mean equality, the statement should be read as “mynum gets the value of 6” (*not* “mynum equals 6”). Note that the variable name must always be on the left, and the expression on the right. An error will occur if these are reversed.

```
>> 6 = mynum
??? 6 = mynum
    |
Error: The expression to the left of the equals sign is not a valid target for an assignment.
```
Putting a semicolon at the end of a statement suppresses the output. For example,
```
>> res = 9 – 2;
>>
```
This would assign the result of the expression on the right side, the value 7, to the variable *res*; it just doesn't show that result. Instead, another prompt appears immediately. However, at this point in the Workspace Window the variables *mynum* and *res* can be seen.

Note: In the remainder of the text, the prompt that appears after the result will not be shown. The spaces in a statement or expression do not affect the result, but make it easier to read. The following statement that has no spaces would accomplish exactly the same thing as the previous statement:
```
>> res = 9–2;
```
MATLAB uses a default variable named *ans* if an expression is typed at the prompt and it is not assigned to a variable. For example, the result of the expression 6 3 is stored in the variable *ans*:
```
>> 6 + 3
ans =
9
```
This default variable is reused any time just an expression is typed at the prompt. A short-cut for retyping commands is to press the up-arrow, which will go back to the previously typed command(s). For example, if you decided to assign the result of the expression 6 3 to the variable *res* instead of using the default *ans*, you could press the up-arrow and then the left-arrow to modify the command rather than retyping the whole statement:
```
>> res = 6 + 3
res =
9
```
This is very useful, especially if a long expression is entered with an error, and you want to go back to correct it. To change a variable, another assignment statement can be used that assigns the value of a different expression to it. Consider, for example, the following sequence of statements:
```
>> mynum = 3
mynum =
3
>> mynum = 4 + 2
mynum =
6
>> mynum = mynum + 1
mynum =
7
```
In the first assignment statement, the value 3 is assigned to the variable *mynum*. In the next assignment statement, *mynum* is changed to have the value of the expression 4 2, or 6. In the third assignment statement, *mynum* is changed again, to the result of the expression mynum 1. Since at that time *mynum* had the value 6, the value of the expression was 6 1, or 7. At that point, if the expression mynum 3 is entered, the default variable *ans* is used since the result of this expression is not assigned to a variable.

Thus, the value of *ans* becomes 10 but *mynum* is unchanged (it is still 7). Note that just typing the name of a variable will display its value.

```
>> mynum + 3

ans =
10
>> mynum
mynum =
7
```

Initializing, Incrementing, and Decrementing

Frequently, values of variables change. Putting the first or initial value in a variable is called ***initializing*** the variable. Adding to a variable is called ***incrementing***. For example, the statement

```
mynum = mynum + 1
```

increments the variable *mynum* by 1.

Variable Names

Variable names are an example of ***identifier names***. We will see other examples of identifier names, such as filenames, in future chapters. The rules for identifier names are:

- The name must begin with a letter of the alphabet. After that, the name can contain letters, digits, and the underscore character (e.g., value_1), but it cannot have a space.
- There is a limit to the length of the name; the built-in function **namelengthmax** tells how many characters this is.
- MATLAB is case-sensitive. That means that there is a difference between upper- and lowercase letters. So, variables called *mynum*, *MYNUM*, and *Mynum* are all different.
- There are certain words called ***reserved words*** that cannot be used as variable names.
- Names of built-in functions can, but should not, be used as variable names.

Additionally, variable names should always be ***mnemonic***, which means they should make some sense. For example, if the variable is storing the radius of a circle, a name such as "radius" would make sense; "x" probably wouldn't. The Workspace Window shows the variables that have been created in the current Command Window and their values.
The following commands relate to variables:

- **who** shows variables that have been defined in this Command Window (this just shows the names of the variables)
- **whos** shows variables that have been defined in this Command Window (this shows more information on the variables, similar to what is in the Workspace Window)
- **clear** clears out all variables so they no longer exist
- **clear** *variablename* clears out a particular variable

If nothing appears when **who** or **whos** is entered, that means there aren't any variables! For example, in the beginning of a MATLAB session, variables could be created and then selectively cleared (remember that the semicolon suppresses output):

```
>> who
>> mynum = 3;
>> mynum + 5;
>> who
Your variables are:
Ans      mynum
>> clear mynum
>> who
Your variables are:
ans
```

Expressions

Expressions can be created using values, variables that have already been created, operators, built-in functions, and parentheses. For numbers, these can include operators such as multiplication, and functions such as trigonometric functions. An example of such an expression would be:

```
>> 2 * sin(1.4)
ans =
1.9709
```

The Format Function and Ellipsis

The ***default*** in MATLAB is to display numbers that have decimal places with four decimal places, as already shown. The **format** command can be used to specify the output format of expressions. There are many options, including making the format **short** (the default) or **long**. For example, changing the format to **long** will result in 15 decimal places. This will remain in effect until the format is changed back to **short**, as demonstrated with an expression and with the built-in value for **pi**.

```
>> format long
>> 2 * sin(1.4)
ans =
1.970899459976920
>> pi
ans =
3.141592653589793
>> format short
>> 2 * sin(1.4)
ans =
1.9709
>> pi
ans =
3.1416
```

The **format** command can also be used to control the spacing between the MATLAB command or expression and the result; it can be either **loose** (the default) or **compact**

```
>> format loose
>> 2^7
ans =
128
>> format compact
>> 2^7
```

ans =
128
Especially long expressions can be continued on the next line by typing three (or more) periods, which is the continuation operator, or the **ellipsis**. For example,
>> *3 + 55 – 62 + 4 – 5* .â•›.â•›.
+ 22 – 1
ans =
16

Operators

There are in general two kinds of operators: ***unary*** operators, which operate on a single value or ***operand***; and ***binary*** operators, which operate on two values or operands. The symbol "–", for example, is both the unary operator for negation and the binary operator for subtraction. Here are some of the common operators that can be used with numeric expressions:
+ addition
– negation, subtraction
* multiplication
/ division (divided by e.g. 10/5 is 2)
\ division (divided into e.g. 5\10 is 2)
^ exponentiation (e.g., 5^2 is 25)

Operator Precedence Rules

Some operators have ***precedence*** over others. For example, in the expression 4 5 * 3, the multiplication takes precedence over the addition, so first 5 is multiplied by 3, then 4 is added to the result. Using parentheses can change the precedence in an expression:
>> *4 + 5 * 3*
ans =
19
>> *(4 + 5) * 3*
ans =
27

Within a given precedence level, the expressions are evaluated from left to right (this is called the ***associativity***). ***Nested parentheses*** are parentheses inside of others; the expression in the ***inner parentheses*** is evaluated first. For example, in the expression 5 –(6 *(4 + 2)), first the addition is performed, then the multiplication, and finally the subtraction to result in -31. Parentheses can also be used simply to make an expression clearer. For example, in the expression ((4 +(3 * 5))–1) the parentheses are not necessary, but are used to show the order in which the expression will be evaluated. For the operators that have been covered so far, the following is the precedence

(from the highest to the lowest):
() parentheses
^ exponentiation
– negation
*, /, \ all multiplication and division
+, – addition and subtraction

Built-In Functions and Help

There are many, many built-in functions in MATLAB. The **help** command can be used to find out what functions MATLAB has, and also how to use them. For example, typing **help** at the prompt in the Command Window will show a list of help topics, which are groups of related functions. This is a very long list; the most elementary help topics are in the beginning.

For example, one of these is listed as **matlab\elfun**; it includes the elementary math functions. Another of the first help topics is **matlab\ops**, which shows the operators that can be used in expressions. To see a list of the functions contained within a particular help topic, type **help** followed by the name of the topic. For example,

```
>> help elfun
```

will show a list of the elementary math functions. It is a very long list, and is broken into trigonometric (for which the default is radians, but there are equivalent functions that instead use degrees), exponential, complex, and rounding and remainder functions. To find out what a particular function does and how to call it, type **help** and then the name of the function. For example,

```
>> help sin
```

will give a description of the **sin** function.
To ***call*** a function, the name of the function is given followed by the ***argument(s)*** that are passed to the function in parentheses. Most functions then ***return*** value(s). For example, to find the absolute value of –4, the following expression would be entered:

```
>> abs(–4)
```

which is a ***call*** to the function **abs**. The number in the parentheses, the –4, is the ***argument***. The value 4 would then be ***returned*** as a result. In addition to the trigonometric functions, the elfun help topic also has some rounding and remainder functions that are very useful. Some of these include **fix**, **floor**, **ceil**, **round**, **rem**, and **sign**. The **rem** function returns the remainder from a division; for example 5 goes into 13 twice with a remainder of 3, so the result of this expression is 3:

```
>> rem(13,5)

ans =
3
```

Another function in the elfun help topic is the **sign** function, which returns 1 if the argument is positive, 0 if it is 0, and –1 if it is negative. For example,

```
>> sign(–5)
ans =
–1
>> sign(3)
ans =
1
```

Constants

Variables are used to store values that can change, or that are not known ahead of time. Most languages also have the capacity to store ***constants***, which are values that are known ahead of time, and cannot possibly change. An example of a constant value would be **pi**, or , which is 3.14159…. In MATLAB, there are functions that return some of these constant values. Some of these include:

pi	3.14159….
i	square root of 1
j	square root of 1
inf	infinity
NaN	stands for “not a number”; e.g., the result of 0/0

Types

Every expression, or variable, has a ***type*** associated with it. MATLAB supports many types of values, which are called ***classes***. A class is essentially a combination of a type and the operations that can be performed on values of that type. For example, there are types to store different kinds of numbers. For float or real numbers, or in other words numbers with a decimal place (e.g., 5.3), there are two basic types: **single** and **double**. The name of the type **double** is short for double precision; it stores larger numbers than **single**. MATLAB uses a ***floating point*** representation for these numbers. For integers, there are many integer types (e.g., **int8**, **int16**, **int32**, and **int64**). The numbers in the names represent the number of bits used to store values of that type. For example, the type **int8** uses eight bits altogether to store the integer and its sign. Since one bit is used for the sign, this means that seven bits are used to store the actual number. Each bit stores the number in binary (0’s or 1’s), and 0 is also a possible value, which means that 2 ^ 7 – 1 or 127 is the largest number that can be stored. The range of values that can be stored in **int8** is actually from –128 to 127. This range can be found for any type by passing the name of the type as a string (which means in single quotes) to the functions **intmin** and **intmax**. For example,

```
>> intmin('int8')

ans =
–128
>> intmax('int8')
ans =
127
```

The larger the number in the type name, the larger the number that can be stored in it. We will for the most part use the type **int32** when an integer type is required. The type **char** is used to store either single ***characters*** (e.g., ‘x’) or ***strings***, which are sequences of characters (e.g., ‘cat’). Both characters and strings are enclosed in single quotes. The type **logical** is used to store true/false values. If any variables have been created in the Command Window, they can be seen in the Workspace Window. In that window, for every variable, the variable name, value, and class (which is essentially its type) can be seen. Other attributes of variables can also be seen in the Workspace Window. Which attributes are visible by default depends on the version of MATLAB. However, when the Workspace Window is chosen, clicking View allows the user to choose which attributes will be displayed. By default, numbers are stored as the type **double** in MATLAB. There are, however, many functions that convert values from one type to another. The names of these functions are the same as the names of the types just shown. They can be used as functions to convert a value to that type. This is called ***casting*** the value to a different type, or type casting. For example, to convert a value from the type **double**, which is the default, to the type **int32**, the function **int32** would be used. Typing the following assignment statement:

```
>> val = 6+3
```

would result in the number 9 being stored in the variable *val*, with the default type of **double**, which can be seen in the Workspace Window. Subsequently, the assignment statement

```
>> val = int32(val);
```

would change the type of the variable to **int32**, but would not change its value. If we instead stored the result in another variable, we could see the difference in the types by using **whos**.

```
>> val = 6 + 3;
>> vali = int32(val);
>> whos
Name Size Bytes Class Attributes
val        1x1       8          double
vali       1x1        4         int32
```

One reason for using an integer type for a variable is to save space.

Random Numbers

When a program is being written to work with data, and the data is not yet available, it is often useful to test the program first by initializing the data variables to ***random numbers***. There are several built-in functions in MATLAB that ***generate*** random numbers, some of which will be illustrated in this section. Random number generators or functions are not truly random. Basically, the way it works is that the process. starts with one number, called a ***seed***. Frequently, the initial seed is either a predetermined value or it is obtained from the built-in clock in the computer. Then, based on this seed, a process determines the next random number. Using that number as the seed the next time, another random number is generated, and so forth. These are actually called ***pseudo-random***; they are not truly random because there is a process that determines the next value each time. The function **rand** can be used to generate random real numbers; calling it generates one random real number in the range from 0 to 1. There are no arguments passed to the **rand** function. Here are two examples of calling the **rand** function:

```
>> rand
ans =
0.9501
>> rand
ans =
0.2311
```

The seed for the **rand** function will always be the same each time MATLAB is started, unless the state is changed, for example, by the following:

```
rand('state',sum(100*clock))
```

This uses the current date and time that are returned from the built-in **clock** function to set the seed. Note: this is done only once in any given MATLAB session to set the seed; the **rand** function can then be used as shown earlier any number of times to generate random numbers. Since **rand** returns a real number in the range from 0 to 1, multiplying the result by an integer N would return a random real number in the range from 0 to N. For example, multiplying by 10 returns a real in the range from 0 to 10, so this expression

```
rand*10
```

would return a result in the range from 0 to 10. To generate a random real number in the range from *low* to *high*, first create the variables *low* and *high*. Then, use the expression rand*(high–low) low. For example, the sequence

```
>> low = 3;
>> high = 5;
>> rand*(high–low)+low
```

would generate a random real number in the range from 3 to 5.
However, in MATLAB, there is another built-in function that specifically generates random integers, **randint**. Calling the function with **randint(1,1,N)** generates one random integer in the range from 0 to N – 1. The first two arguments essentially specify that one random integer will be returned; the third argument gives the range of that random integer. For example,

```
>> randint(1,1,4)
```

generates a random integer in the range from 0 to 3. Note: Even though this creates random integers, the type is actually the default type **double**. A range can also be passed to the **randint** function. For example, the following specifies a random integer in the range from 1 to 20:

```
>> randint(1,1,[1,20])
```

Vectors and Matrices

Vectors and ***matrices*** are used to store sets of values, all of which are the same type. A vector can be either a ***row vector*** or a ***column vector***. A matrix can be visualized as a table of values. The dimensions of a matrix are r × c, where r is the number of rows and c is the number of columns. This is pronounced "r by c." If a vector has *n* elements, a row vector would have the dimensions 1 × *n*, and a column vector would have the dimensions *n* × 1. A ***scalar*** (one value) has the dimensions 1 × 1. Therefore, vectors and scalars are actually just subsets of matrices. Here are some diagrams showing, from left to right, a scalar, a column vector, a row vector, and a matrix:

5

3
7
4

5	88	3	11

9	6	3
5	7	2
4	33	8

The scalar is 1 × 1, the column vector is 3 × 1 (3 rows by 1 column), the row vector is 1 × 4 (1 row by 4 columns), and the matrix is 3 × 3. All the values stored in these matrices are stored in what are called ***elements***. MATLAB is written to work with matrices; the name MATLAB is short for "matrix laboratory." For this reason, it is very easy to create vector and matrix variables, and there are many operations and functions that can be used on vectors and matrices. A vector in MATLAB is equivalent to what is called a one-dimensional ***array*** in other languages. A matrix is equivalent to a two-dimensional array. Usually, even in MATLAB, some operations that can be performed on either vectors or matrices are referred to as ***array operations***. The term ***array*** also frequently is used to mean generically either a vector or a matrix.

1.5.1 Creating Row Vectors

There are several ways to create row vector variables. The most direct way is to put the values that you want in the vector in square brackets, separated by either spaces or commas. For example, both of these assignment statements create the same vector v:

```
>> v = [1       2       3       4]
v =
1       2       3       4
>> v = [1,2,3,4]
```

```
v =
1        2        3        4
```

Both of these create a row vector variable that has four elements; each value is stored in a separate element in the vector.

The Colon Operator and** Linspace **Function

If, as in the earlier examples, the values in the vector are regularly spaced, the ***colon operator*** can be used to ***iterate*** through these values. For example, 1:5
results in all the integers from 1 to 5:

```
>> vec = 1:5
vec =
1        2        3        4        5
```

Note that in this case, the brackets [ ] are not necessary to define the vector.
With the colon operator, a ***step value*** can also be specified with another colon, in the form (first:step:last).
For example, to create a vector with all integers from 1 to 9 in steps of 2:

```
>> nv = 1:2:9
nv =
1        3        5        7        9
```

Similarly, the **linspace** function creates a linearly spaced vector; **linspace(x,y,n)** creates a vector with n values in the inclusive range from x to y. For example, the following creates a vector with five values linearly spaced between 3 and 15, including the 3 and 15:

```
>> ls = linspace(3,15,5)
ls =
3        6        9        12        15
```

Vector variables can also be created using existing variables. For example, a new vector is created here consisting first of all the values from *nv* followed by all values from *ls*:

```
>> newvec = [nv ls]
newvec =
1        3        5        7        9        3        6        9        12        15
```

Putting two vectors together like this to create a new one is called ***concatenating*** the vectors.

1.5.1.2 Referring to and Modifying Elements

A particular element in a vector is accessed using the name of the vector variable and the element number (or ***index***, or ***subscript***) in parentheses. In MATLAB, the indices start at 1. Normally, diagrams of vectors and matrices show the indices; for example, for the variable *newvec* created earlier the indices 1–10 of the elements are shown above the vector:

1	2	3	4	5	6	7	8	9	10
1	3	5	7	9	3	6	9	12	15

For example, the fifth element in the vector *newvec* is a 9.

```
>> newvec(5)
ans =
9
```

A subset of a vector, which would be a vector itself, can also be obtained using the colon operator. For example, the following statement would get the fourth through sixth elements of the vector *newvec*, and store the result in a vector variable *b*:

```
>> b = newvec(4:6)
b =
7         9         3
```

Any vector can be used for the indices in another vector, not just one created using the colon operator. For example, the following would get the first, fifth, and tenth elements of the vector *newvec*:

```
>> newvec([1 5 10])
ans =
1         9         15
```

The vector [1 5 10] is called an ***index vector***; it specifies the indices in the original vector that are being referenced. The value stored in a vector element can be changed by specifying the index or subscript. For example, to change the second element from the vector *b* to now store the value 11 instead of 9:

```
>> b(2) = 11
b =
7         11        3
```

By using an index, a vector can also be extended. For example, the following creates a vector that has three elements. By then referring to the fourth element in an assignment statement, the vector is extended to have four elements.

```
>> rv = [3 55 11]
rv =
3         55        11
>> rv(4) = 2
rv =
3         55        11        2
```

If there is a gap between the end of the vector and the specified element, 0's are filled in. For example, the following extends the variable created earlier again:

```
>> rv(6) = 13
rv =
3         55        11        2         0         13
```

Creating Column Vectors

One way to create a column vector is by explicitly putting the values in square brackets, separated by semicolons:

```
>> c = [1; 2; 3; 4]
c =
1
2
3
4
```

There is no direct way to use the colon operator described earlier to get a column vector. However, any row vector created using any of these methods can be ***transposed*** to get a column vector. In general, the transpose of a matrix is a new matrix in which the rows and columns are interchanged. For vectors, transposing a row vector results in a column vector, and transposing a column vector results in a row vector. MATLAB has a built-in operator, the apostrophe, to get a transpose.

```
>> r = 1:3;
>> c = r
c =
1
2
3
```

Creating Matrix Variables

Creating a matrix variable is really just a generalization of creating row and column vector variables. That is, the values within a row are separated by either spaces or commas, and the different rows are separated by semicolons. For example, the matrix variable *mat* is created by explicitly typing values:

```
>> mat = [4 3 1; 2 5 6]
mat =
4       3       1
2       5       6
```

There must always be the same number of values in each row. If you attempt to create a matrix in which there are different numbers of values in the rows, the result will be an error message; for example:

```
>> mat = [3 5 7; 1 2]
??? Error using ==> vertcat
```

CAT arguments dimensions are not consistent. Iterators can also be used for the values on the rows using the colon operator; for example:

```
>> mat = [2:4; 3:5]
mat =
2       3       4
3       4       5
```

Different rows in the matrix can also be specified by pressing the Enter key after each row instead of typing a semicolon when entering the matrix values; for example:

```
>> newmat = [2 6 88
33 5 2]

newmat =
2       6       88
33      5       2
```

Matrices of random numbers can be created using the **rand** and **randint** functions. The first two arguments to the **randint** function specify the size of the matrix of random integers. For example, the following will create a 2 × 4 matrix of random integers, each in the range from 10 to 30:

```
>> randint(2,4,[10,30])
ans =
29      22      28      19
14      20      26      10
```

For the **rand** function, if a single value *n* is passed to it, an *n* × *n* matrix will be created, or passing two arguments will specify the number of rows and columns:

```
>> rand(2)

ans =
0.2311          0.4860
0.6068          0.8913

>> rand(1,3)
ans =
0.7621          0.4565          0.0185
```

MATLAB also has several functions that create special matrices. For example, the **zeros** function creates a matrix of all zeros. Like **rand**, either one argument can be passed (which will be both the number of rows and columns), or two arguments (first the number of rows and then the number of columns).

```
>> zeros(3)
ans =
0       0       0
0       0       0
0       0       0

>> zeros(2,4)
ans =
0       0       0       0
0       0       0       0
```

Referring to and Modifying Matrix Elements

To refer to matrix elements, the row and then the column indices are given in parentheses (always the row index first and then the column). For example, this creates a matrix variable *mat*, and then refers to the value in the second row, third column of *mat*:

```
>> mat = [2:4; 3:5]
mat =
2       3       4
3       4       5
```

```
>> mat(2,3)
ans =
5
```

It is also possible to refer to a subset of a matrix. For example, this refers to the first and second rows, second and third columns:

```
>> mat(1:2,2:3)
ans =
3       4
4       5
```

Using a colon for the row index means all rows, regardless of how many, and using a colon for the column index means all columns. For example, this refers to the entire first row:

```
>> mat(1,:)
ans =
2       3       4
```

and this refers to the entire second column:

```
>> mat(:, 2)
ans =
3
4
```

If a single index is used with a matrix, MATLAB ***unwinds*** the matrix column by column. For example, for the matrix *intmat* created here, the first two elements are from the first column, and the last two are from the second column:

```
>> intmat = randint(2,2,[0 100])
intmat =
100         77
28          14

>> intmat(1)
ans =
100
>> intmat(2)
ans =
28
>> intmat(3)
ans =
77
>> intmat(4)
ans =
14
```

This is called ***linear indexing***. It is usually much better style when working with matrices to refer to the row and column indices, however. An individual element in a matrix can be modified by assigning a value.

```
>> mat = [2:4; 3:5];
>> mat(1,2) = 11
mat =
2       11      4
3       4       5
```

An entire row or column could also be changed. For example, the following replaces the entire second row with values from a vector:

```
>> mat(2,:) = 5:7
mat =
2       11      4
5       6       7
```

Notice that since the entire row is being modified, a vector with the correct length must be assigned. To extend a matrix, an individual element could not be added since that would mean there would no longer be the same number of values in every row. However, an entire row or column could be added. For example, the following would add a fourth column to the matrix:

```
>> mat(:,4) = [9 2]’
mat =
2       11      4       9
5       6       7       2
```

Just as we saw with vectors, if there is a gap between the current matrix and the row or column being added, MATLAB will fill in with zeros.

```
>> mat(4,:) = 2:2:8
mat =
2       11      4       9
5       6       7       2
0       0       0       0
2       4       6       8
```

Dimensions

The **length** and **size** functions in MATLAB are used to find array dimensions. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a matrix. For a matrix, the **length** function will return either the number of rows or the number of columns, whichever is largest. For example, the following vector, *vec*, has four elements so its length is 4. It is a row vector, so the size is 1 × 4.

```
>> vec = -2:1
vec =
-2      -1      0       1
>> length(vec)
ans =
4
>> size(vec)
```

```
ans =
1          4
```

For the matrix *mat* shown next, it has three rows and two columns, so the size is 3 × 2. The length is the larger dimension, 3.

```
>> mat = [1:3; 5:7]'
mat =
1         5
2         6
3         7
>> size(mat)
ans =
3         2
>> length(mat)
ans =
3
>> [r c] = size(mat)
r =
3
c =
2
```

Note: The last example demonstrates a very important and unique concept in MATLAB: the ability to have a vector of variables on the left-hand side of an assignment. The **size** function returns two values, so in order to capture these values in separate variables we put a vector of two variables on the left of the assignment. The variable *r* stores the first value returned, which is the number of rows, and *c* stores the number of columns.

MATLAB also has a function, **numel**, which returns the total number of elements in any array (vector or matrix):

```
>> vec = 9:–2:1
vec =
9 7 5 3 1

>> numel(vec)
ans =
5

>> mat = randint(2,3,[1,10])
mat =
7         9          8
4          6          5

>> numel(mat)
ans =
6
```

For vectors, this is equivalent to the length of the vector. For matrices, it is the product of the number of rows and columns. MATLAB also has a built-in expression **end** that can be used to refer to the last element in a vector; for example, v(end) is equivalent to v(length(v)). For matrices, it can refer to the last row or column. So, using **end** for the row index would refer to the last row. In this case, the element referred to is in the first column of the last row:

```
>> mat = [1:3; 4:6]'
mat =
1        4
2        5
3        6

>> mat(end,1)
ans =
3
```

Using **end** for the column index would refer to the last column (e.g., the last column of the second row):

```
>> mat(2,end)
ans =
5
```

This can be used only as an index.

Changing Dimensions

In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices, including **reshape**, **fliplr**, **flipud**, and **rot90**. The **reshape** function changes the dimensions of a matrix. The following matrix variable *mat* is 3 4, or in other words it has 12 elements.

```
>> mat = randint(3,4,[1 100])
mat =
14       61       2        94
21       28       75       47
20       20       45       42
```

These 12 values instead could be arranged as a 2 x 6 matrix, 6 x 2, 4 x3, 1x 12, or 12 x1. The **reshape** function iterates through the matrix columnwise. For example, when reshaping *mat* into a 2 6 matrix, the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

```
>> reshape(mat,2,6)
ans =
14       20       28       2        45       47
21       61       20       75       94       42
```

The **fliplr** function “flips” the matrix from left to right (in other words the left-most column, the first column, becomes the last column and so forth), and the **flipud** functions flips up to down. Note that in these examples *mat* is unchanged; instead, the results are stored in the default variable *ans* each time.

```
>> mat = randint(3,4,[1 100])
mat =
14      61      2       94
21      28      75      47
20      20      45      42

>> fliplr(mat)
ans =
94      2       61      14
47      75      28      21
42      45      20      20

>> mat
mat =
14      61      2       94
21      28      75      47
20      20      45      42

>> flipud(mat)
ans =
20      20      45      42
21      28      75      47
14      61      2       94
```

The **rot90** function rotates the matrix counterclockwise 90 degrees, so for example the value in the top-right corner becomes instead the top-left corner and the last column becomes the first row:

```
>> mat
mat =
14      61      2       94
21      28      75      47
20      20      45      42

>> rot90(mat)
ans =
94      47      42
2       75      45
61      28      20
14      21      20
```

The function **repmat** can also be used to create a matrix; **repmat(mat,m,n)** creates a larger matrix, which consists of an *m* × *n* matrix of copies of mat. For example, here is a 2 × 2 random matrix:

```
>> intmat = randint(2,2,[0 100])
intmat =
100     77
28      14
```

The function **repmat** can be used to replicate this matrix six times as a 3 × 2 matrix of the variable *intmat*.

```
>> repmat(intmat,3,2)
ans =
100     77      100     77
28      14      28      14
100     77      100     77
28      14      28      14
28      14      28      14
```

Empty Vectors

An ***empty vector***, or, in other words, a vector that stores no values, can be created using empty square brackets:

```
>> evec = [ ]
evec =
[ ]
>> length(evec)
ans =
0
```

Then, values can be added to the vector by ***concatenating***, or adding values to the existing vector. The following statement takes what is currently in *evec*, which is nothing, and adds a 4 to it.

```
>> evec = [evec 4]
evec =
4
```

The following statement takes what is currently in *evec*, which is 4, and adds
an 11 to it.

```
>> evec = [evec 11]
evec =
4       11
```

This can be continued as many times as desired, in order to build a vector up from nothing. Empty vectors can also be used to ***delete elements from arrays***. For example, to remove the third element from an array, the empty vector is assigned to it:

```
>> vec = 1:5
vec =
1       2       3       4       5
>> vec(3) = [ ]
vec =
1       2       4       5
```

The elements in this vector are now numbered 1 through 4. Subsets of a vector could also be removed; for example:

```
>> vec = 1:8
vec =
1       2       3       4       5       6       7       8
>> vec(2:4) = [ ]
```

```
vec =
1         5         6         7         8
```

Individual elements cannot be removed from matrices, since matrices always have to have the same number of elements in every row.

```
>> mat = [7 9 8; 4 6 5]
mat =
7         9         8
4         6         5
```

```
>> mat(1,2) = [ ];
```

??? Indexed empty matrix assignment is not allowed. However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

```
>> mat(:,2) = [ ]
mat =
7         8
4         5
```

Creating String Variables

A string consists of any number of characters (including, possibly, none). These are examples of strings: ‘’
‘x’
‘cat’
‘Hello there’
‘123’

A ***substring*** is a subset or part of a string. For example, ‘there’ is a substring within the string ‘Hello there’. ***Characters*** include letters of the alphabet, digits, punctuation marks, white space, and control characters. ***Control characters*** are characters that cannot be printed, but accomplish a task (such as a backspace or tab). ***Whitespace characters*** include the space, tab, newline (which moves the cursor down to the next line), and carriage return (which moves the cursor to the beginning of the current line). ***Leading blanks*** are blank spaces at the beginning of a string, for example, ‘ hello’, and ***trailing blanks*** are blank spaces at the end of a string. There are several ways that string variables can be created. One is using assignment statements:

```
>> word = ‘cat’;
```

Another method is to read into a string variable. Recall that to read into a string variable using the **input** function, the second argument ‘s’ must be included:

```
>> strvar = input(‘Enter a string: ’, ‘s’)
Enter a string: xyzabc
strvar =
xyzabc
```

If leading or trailing blanks are typed by the user, these will be stored in the string. For example, in the following the user entered four blanks and then ‘xyz’:

```
>> s = input('Enter a string: ','s')
Enter a string: xyz
s =
xyz
```

Strings as Vectors

Strings are treated as ***vectors of characters***—or in other words, a vector in which every element is a single character—so many vector operations can be performed. For example, the number of characters in a string can be found using the **length** function:

```
>> length('cat')
ans =
3
>> length(' ')
ans =
1
>> length('')
ans =
0
```

Notice that there is a difference between an ***empty string***, which has a length of zero, and a string consisting of a blank space, which has a length of one. Expressions can refer to an individual element (a character within the string), or a subset of a string or a transpose of a string:

```
>> mystr = 'Hi';
>> mystr(1)
ans =
H
>> mystr'
ans =
H
i
>> sent = 'Hello there';
>> length(sent)
ans =
11
>> sent(4:8)
ans =
lo th
```

Notice that the blank space in the string is a valid character within the string. A matrix can be created, which consists of strings in each row. So, essentially it is created as a column vector of strings, but the end result is that this would be treated as a matrix in which every element is a character:

```
>> wordmat = ['Hello';'Howdy']
wordmat =
Hello
Howdy
```

```
>> size(wordmat)
ans =
2         5
```

This created a 2 5 matrix of characters. With a character matrix, we can refer to an individual element, which is a character, or an individual row, which is one of the strings:

```
>> wordmat(2,4)
ans =
d
>> wordmat(1,:)
ans =
Hello
```

Since rows within a matrix must always be the same length, the shorter strings must be padded with blanks so that all strings have the same length, otherwise an error will occur.

```
>> greetmat = ['Hello'; 'Goodbye']
??? Error using ==> vertcat

CAT arguments dimensions are not consistent.
>> greetmat = ['Hello  '; 'Goodbye']
greetmat =
Hello
Goodbye

>> size(greetmat)
ans =
2         7
```

Operations on Strings

MATLAB has many built-in functions that work with strings. Some of the string manipulation functions that perform the most common operations will be described here.

Concatenation

String concatenation means to join strings together. Of course, since strings are just vectors of characters, the method of concatenating vectors works for strings, also. For example, to create one long string from two strings, it is possible to join them by putting them in square brackets:

```
>> first = 'Bird';
>> last = 'house';
>> [first last]
ans =
Birdhouse
```

The function **strcat** does this also horizontally, meaning that it creates one longer string from the inputs.

```
>> first = 'Bird';
>> last = 'house';
>> strcat(first,last)
ans =
Birdhouse
```

There is a difference between these two methods of concatenating, however, if there are leading or trailing blanks in the strings. The method of using the square brackets will concatenate the strings, including all leading and trailing blanks.

```
>> str1 = 'xxx ';
>> str2 = ' yyy';
>> [str1 str2]
ans =
xxx yyy
>> length(ans)
ans =
12
```

The **strcat** function, however, will remove trailing blanks (but not leading blanks) from strings before concatenating. Notice that in these examples, the trailing blanks from *str1* are removed, but the leading blanks from *str2* are not:

```
>> strcat(str1,str2)
ans =
xxx yyy
>> length(ans)
ans =
9
>> strcat(str2,str1)
ans =
yyyxxx
>> length(ans)
ans =
9
```

The function **strvcat** will concatenate vertically, meaning that it will create a
column vector of strings.

```
>> strvcat(first,last)
ans =
Bird
house
>> size(ans)
ans =
2       5
```

Note that **strvcat** will pad with extra blanks automatically, in this case to make both strings have a length of 5.

Creating Customized Strings

There are several built-in functions that create customized strings, including **char, blanks**, and **sprintf**. We have seen already that the **char** function can be used to convert from an ASCII code to a character, for example:

```
>> char(97)
ans =
a
```

The **char** function can also be used to create a matrix of characters. When using the **char** function to create a matrix, it will automatically pad the strings within the rows with blanks as necessary so that they are all the same length, just like **strvcat**.

```
>> clear greetmat
>> greetmat = char(‘Hello’,‘Goodbye’)
greetmat =
Hello
Goodbye
>> size(greetmat)
ans =
2 7
```

The **blanks** function will create a string consisting of n blank characters which are kind of hard to see here! However, in MATLAB if the mouse is moved to highlight the result in *ans*, the blanks can be seen.

```
>> blanks(4)
ans =
>> length(ans)
ans =
4
```

Usually this function is most useful when concatenating strings, and you want a number of blank spaces in between. For example, this will insert five blank spaces in between the words:

```
>> [first blanks(5) last]
ans =
Bird house
```

Displaying the transpose of the **blanks** function can also be used to move the cursor down. In the Command Window, it would look like this:

```
>> disp(blanks(4)’)
```

This is useful in a script or function to create space in output, and is essentially equivalent to printing the newline character four times. The **sprintf** function works exactly like the **fprintf** function, but instead of printing it creates a string. Here are several examples in which the output is not suppressed so the value of the string variable is shown:

```
>> sent1 = sprintf('The value of pi is %.2f', pi)
sent1 =
The value of pi is 3.14
>> sent2 = sprintf('Some numbers: %5d, %2d', 33, 6)
sent2 =
Some numbers: 33, 6
>> length(sent2)
ans =
23
```

In the following example, on the other hand, the output of the assignment is suppressed so the string is created including a random integer and stored in the string variable. Then, some exclamation points are concatenated to that string.

```
>> phrase = sprintf('A random integer is %d', ›.
randint(1,1,[5,10]));
>> strcat(phrase, '!!!')
ans =
A random integer is 7!!!
```

All the conversion specifiers that can be used in the **fprintf** function can also be used in the **sprintf** function.

Removing Whitespace Characters

MATLAB has functions that will remove trailing blanks from the end of a string and/or leading blanks from the beginning of a string. The **deblank** function will remove blank spaces from the end of a string. For example, if some strings are padded in a string matrix so that all are the same length, it is frequently preferred to then remove those extra blank spaces in order to actually use the string.

```
>> names = char('Sue', 'Cathy','Xavier')
names =
Sue
Cathy
Xavier
>> name1 = names(1,:)
name1 =
Sue
>> length(name1)
ans =
6
>> name1 = deblank(name1);
>> length(name1)
ans =
3
```

Note: The **deblank** function removes only trailing blanks from a string, not leading blanks. The **strtrim** function will remove both leading and trailing blanks from a string, but not blanks in the middle of the string. In the following example, the three blanks in the beginning and four blanks in the end are removed, but not the two blanks in the middle. Selecting the result in MATLAB with the mouse would show the blank spaces.

```
>> strvar = [blanks(3) ‘xx’ blanks(2) ‘yy’ blanks(4)]
strvar =
xx yy
>> length(strvar)
ans =
13
>> strtrim(strvar)
ans =
xx yy
>> length(ans)
ans =
6
```

Changing Case

MATLAB has two functions that convert strings to all uppercase letters, or all lowercase, called **upper** and **lower**.

```
>> mystring = ‘AbCDEfgh’;
>> lower(mystring)
ans =
abcdefgh

>> upper(ans)
ans =
ABCDEFGH
```

Comparing Strings

There are several functions that compare strings and return logical true if they are equivalent, or logical false if not. The function **strcmp** compares strings, character by character. It returns logical true if the strings are completely identical (which infers that they must be of the same length, also) or logical false if the strings are not the same length or any corresponding characters are not identical. Here are some examples of these comparisons:

```
>> word1 = ‘cat’;
>> word2 = ‘car’;
>> word3 = ‘cathedral’;
>> word4 = ‘CAR’;
>> strcmp(word1,word2)
ans =
0
>> strcmp(word1,word3)
ans =
0
>> strcmp(word1,word1)
```

```
ans =
1
>> strcmp(word2,word4)
ans =
0
```

The function **strncmp** compares only the first *n* characters in strings and ignores the rest. The first two arguments are the strings to compare, and the third argument is the number of characters to compare (the value of *n*).

```
>> strncmp(word1,word3,3)
ans =
1
>> strncmp(word1,word3,4)
ans =
0
```

There is also a function **strncmpi** that compares *n* characters, ignoring the case.

Finding, Replacing, and Separating Strings

There are several functions that find and replace strings, or parts of strings, within other strings and functions that separate strings into substrings. The function **findstr** receives two strings as input arguments. It finds all occurrences of the shorter string within the longer, and returns the subscripts of the beginning of the occurrences. The order of the strings does not matter with **findstr**; it will always find the shorter string within the longer, whichever that is. The shorter string can consist of one character, or any number of characters. If there is more than one occurrence of the shorter string within the longer one, **findstr** returns a vector with all indices. Note that what is returned is the index of the beginning of the shorter string.

```
>> findstr('abcde', 'd')
ans =
4
>> findstr('d','abcde')
ans =
4
>> findstr('abcde', 'bc')
ans =
2
>> findstr('abcdeabcdedd', 'd')
ans =
4       9       11      12
```

The function **strfind** does essentially the same thing, except that the order of the arguments does make a difference. The general form is **strfind(string, substring)**; it finds all occurrences of the substring within the string, and returns the subscripts.

```
>> strfind('abcdeabcde','e')
ans =
5       10
```

For both **strfind** and **findstr**, if there are no occurrences, the empty vector is returned.

```
>> strfind('abcdeabcde','ef')
ans =
[ ]
```

The function **strrep** finds all occurrences of a substring within a string, and replaces them with a new substring. The order of the arguments matters. The format is: strrep(string, oldsubstring, newsubstring) The following example replaces all occurrences of the substring 'e' with the substring 'x':

```
>> strrep('abcdeabcde','e','x')
ans =
abcdxabcdx
```

All strings can be any length, and the lengths of the old and new substrings do not have to be the same. In addition to the string functions that find and replace, there is a function that separates a string into two substrings. The **strtok** function breaks a string into pieces; it can be called several ways. The function receives one string as an input argument. It looks for the first ***delimiter***, which is a character or set of characters that act as a separator within the string. By default, the delimiter is any whitespace character. The function returns a ***token***, which is the beginning of the string, up to (but not including) the first delimiter. It also returns the rest of the string, which includes the delimiter. Assigning the returned values to a vector of two variables will capture both of these. The format is

[token rest] = strtok(string)

where *token* and *rest* are variable names. For example,

```
>> sentence1 = 'Hello there'
sentence1 =
Hello there
>> [word rest] = strtok(sentence1)
word =
Hello
rest =
there
>> length(word)
ans =
5
>> length(rest)
ans =
6
```

Notice that the rest of the string includes the blank space delimiter. By default, the delimiter for the token is a whitespace character (meaning that the token is defined as everything up to the blank space), but alternate delimiters can be defined. The format

[token rest] = strtok(string, delimeters)

returns a token that is the beginning of the string, up to the first character contained within the delimiters string, and also the rest of the string. In the following example, the delimiter is the character 'l'.

```
>> [word rest] = strtok(sentence1,'l')
```

```
word =
He
rest =
llo there
```

Leading delimiter characters are ignored, whether it is the default whitespace or a specified delimiter. For example, the leading blanks are ignored here:

```
>> [firstpart lastpart] = strtok(' materials science')
firstpart =
materials
lastpart =
science
```

Evaluating a String

The function **eval** is used to evaluate a string as a function.For example, in the following, the string 'plot(x)' is interpreted to be a call to the **plot** function, and it produces the plot shown in Figure 6.2.

```
>> x = [2 6 8 3];
>> eval('plot(x)')
```

This would be useful if the user entered the name of the type of plot to use. In this example, the string that the user enters (in this case 'bar') is concatenated with the string '(x)' to create the string 'bar(x)'; this is then evaluated as a call to the **bar** function as seen in Figure 6.3. The name of the plot type is also used in the title.

The is functions for strings

There are several **is** functions for strings, which return logical true or false. The function **isletter** returns logical true if the character is a letter of the alphabet. The function **isspace** returns logical true if the character is a whitespace character. If strings are passed to these functions, they will return logical true or false for every element, or, in other words, every character.

```
>> isletter('a')
ans =
1
>> isletter('EK127')
ans =
1 1 0 0 0
>> isspace('a b')
ans =
0 1 0
```

The **ischar** function will return logical true if an array is a character array, or logical false if not.

```
>> vec = 'EK127';
>> ischar(vec)
ans =
1
>> vec = 3:5;
>> ischar(vec)
```

```
ans =
0
```

Converting between string and number types

MATLAB has several functions that convert numbers to strings in which each character element is a separate digit, and vice versa. (Note: these are different from the functions **char, double**, etc., that convert characters to ASCII equivalents and vice versa.) To convert numbers to strings, MATLAB has the functions **int2str** for integers and **num2str** for real numbers (which also works with integers). The function **int2str** would convert, for example, the integer 4 to the string ‘4’.

```
>> rani = randint(1,1,50)
rani =
38
>> s1 = int2str(rani)
s1 =
38
>> length(rani)
ans =
1
>> length(s1)
ans =
2
```

The variable *rani* is a scalar that stores one number, whereas *s1* is a string that stores two characters, ‘3’ and ‘8’. Even though the result of the first two assignments is 38, notice that the indentation in the Command Window is different for the number and the string. The **num2str** function, which converts real numbers, can be called in several ways. If only the real number is passed to the **num2str** function, it will create a string that has four decimal places, which is the default in MATLAB for displaying real numbers. The precision can also be specified (which is the number of digits), and format strings can also be passed, as shown:

```
>> str2 = num2str(3.456789)
str2 =
3.4568
>> length(str2)
ans =
6
>> str3 = num2str(3.456789,3)
str3 =
3.46
>> str = num2str(3.456789,‘%6.2f’)
str =
3.46
```

Note that in the last example, MATLAB removed the leading blanks from the string. The function **str2num** does the reverse; it takes a string in which a number is stored and converts it to the type **double**:

```
>> num = str2num(‘123.456’)
num =
123.4560
```

If there is a string in which there are numbers separated by blanks, the **str2num** function will convert this to a vector of numbers (of the default type double). For example,

```
>> mystr = ‘66 2 111’;
>> numvec = str2num(mystr)
numvec =
66 2 111
>> sum(numvec)
ans =
179
```

Input and Output

The previous script would be much more useful if it were more general; for example, if the value of the radius could be read from an external source rather than being assigned in the script. Also, it would be better to have the script print the output in a nice, informative way. Statements that accomplish these tasks are called ***input/output*** statements, or ***I/O*** for short. Although for simplicity examples of input and output statements will be shown here from the Command Window, these statements will make the most sense in scripts.

Input Function

Input statements read in values from the default or ***standard input device***. In most systems, the default input device is the keyboard, so the input statement reads in values that have been entered by the ***user***, or the person who is running the script. In order to let the user know what he or she is supposed to enter, the script must first ***prompt*** the user for the specified values. The simplest input function in MATLAB is called **input**. The **input** function is used in an assignment statement. To call it, a string is passed, which is the prompt that will appear on the screen, and whatever the user types will be stored in the variable named on the left of the assignment statement. To make it easier to read the prompt, put a colon and then a space after the prompt. For example,

```
>> rad = input(‘Enter the radius: ’)
Enter the radius: 5
rad =
5
```

If character or string input is desired, ‘s’ must be added after the prompt:

```
>> letter = input(‘Enter a char: ’,‘s’)
Enter a char: g
letter =
g
```

Notice that although this is a string variable, the quotes are not shown. However, they are shown in the Workspace Window. If the user enters only spaces or tabs before pressing the Enter key, they are ignored and an ***empty string*** is stored in the variable:

```
>> mychar = input(‘Enter a character: ’, ‘s’)
Enter a character:
mychar =
‘’
```

Notice that in this case the quotes are shown, to demonstrate that there is nothing inside of the string. However, if blank spaces are entered before other characters, they are included in the string. In this example, the user pressed the space bar four times before entering “go”:

```
>> mystr = input('Enter a string: ', 's')
Enter a string: go
mystr =
go
>> length(mystr)
ans =
6
```

It is also possible for the user to type quotation marks around the string rather than including the second argument ‘s’ in the call to the **input** function:

```
>> name = input('Enter your name: ');
```

Enter your name: *‘Stormy’* However, it is better to signify that character input is desired in the **input** function itself. Normally, the results from **input** statements are suppressed with a semicolon at the end of the assignment statements, as shown here. Notice what happens if string input has not been specified, but the user enters a letter rather than a number:

```
>> num = input('Enter a number: ')
Enter a number: t
??? Error using ==> input
Undefined function or variable 't'.
Enter a number: 3
num =
3
```

MATLAB gave an ***error message*** and repeated the prompt. However, if *t* is the name of a variable, MATLAB will take its value as the input:

```
>> t = 11;
>> num = input('Enter a number: ')
Enter a number: t
num =
11
```

Separate **input** statements are necessary if more than one input is desired. For example

```
>> x = input('Enter the x coordinate: ');
>> y = input('Enter the y coordinate: ');
```

Output Statements: disp **and** fprintf

Output statements display strings and the results of expressions, and can allow for ***formatting***, or customizing how they are displayed. The simplest output function in MATLAB is **disp**, which is used to display the result of an expression or a string without assigning any value to the default variable *ans*. However, **disp** does not allow formatting. For example,

```
>> disp('Hello')
```

```
Hello
>> disp(4^3)
64
```

Formatted output can be printed to the screen using the **fprintf** function. For example,

```
>> fprintf('The value is %d, for sure!\n',4^3)
The value is 64, for sure!
```

To the **fprintf** function, first a string (called the ***format string***) is passed, which contains any text to be printed as well as formatting information for the expressions to be printed. In this example, the %d is an example of format information. The %d is sometimes called a ***placeholder***; it specifies where the value of the expression that is after the string is to be printed. The character in the placeholder is called the ***conversion character***, and it specifies the type of value that is being printed. There are others, but what follows is a list of the simple placeholders:

%d	integers (it actually stands for decimal integer)
%f	floats
%c	single characters
%s	strings

Don't confuse the % in the placeholder with the symbol used to designate a comment. The character '\n' at the end of the string is a special character called the ***newline*** character; when it is printed the output moves down to the next line. A ***field width*** can also be included in the placeholder in **fprintf**, which specifies how many characters total are to be used in printing. For example, %5d would indicate a field width of 5 for printing an integer and %10â•›s would indicate a field width of 10 for a string. For floats, the number of decimal places can also be specified; for example, %6.2f means a field width of 6 (including the decimal point and the decimal places) with two decimal places. For floats, just the number of decimal places can also be specified; for example, %.3f if indicates three decimal places.

```
>> fprintf('The int is %3â•›d and the float is %6.2f\n',5,4.9)
```

The int is 5 and the float is 4.90 Note that if the field width is wider than necessary, ***leading*** blanks are printed, and if more decimal places are specified than necessary, ***trailing*** zeros are printed. There are many other options for the format string. For example, the value being printed can be left-justified within the field width using a minus sign. The following example shows the difference between printing the integer 3 using %5d and using %–5d. The x's are just used to show the spacing.

```
>> fprintf('The integer is xx%5dxx and xx%-5dxx\n',3,3)
The integer is xx 3xx and xx3 xx
```

Also, strings can be truncated by specifying decimal places:

```
>> fprintf('The string is %s or %.4s\n', 'truncate',...
'truncate')
```

The string is truncate or trun. There are several special characters that can be printed in the format string in addition to the newline character. To print a slash, two slashes in a row are used, and also to print a single quote two single quotes in a row are used. Additionally, \t is the tab character.

```
>> fprintf('Try this out: tab\t quote " slash \\ \n')
Try this out: tab quote ' slash \
```

Scripts with Input and Output

Putting all this together, we can implement the algorithm from the beginning of this chapter. The following script calculates and prints the area of a circle. It first prompts the user for a radius, reads in the radius, and then calculates and prints the area of the circle based on this radius.

```
script2.m
% This script calculates the area of a circle
% It prompts the user for the radius
% Prompt the user for the radius and calculate
% the area based on that radius
radius = input('Please enter the radius: ');
area = pi * (radius^2);
% Print all variables in a sentence format
fprintf('For a circle with a radius of %.2f,',radius)
fprintf('the area is %.2f\n',area)
```

Executing the script produces the following output:

```
>> script2
Please enter the radius: 3.9
For a circle with a radius of 3.90, the area is 47.78
```

Notice that the output from the first two assignment statements is suppressed by putting semicolons at the end. That is frequently done in scripts, so that the exact format of what is displayed by the program is controlled by the **fprintf** functions.

Introduction to File Input/Output (Load and Sav e)

In many cases, input to a script will come from a data file that has been created by another source. Also, it is useful to be able to store output in an external file that can be manipulated and/or printed later. In this section, we will demonstrate how to read from an external data file, and also how to write to an external data file. There are basically three different operations, or ***modes***, on files. Files can be:

- Read from
- Written to
- *Appended* to

Writing to a file means writing to a file, from the beginning. Appending to a file is also writing, but starting at the end of the file rather than the beginning. In other words, appending to a file means adding to what was already there. There are many different file types, which use different filename extensions. For now, we will keep it simple and just work with .dat or .txt files when working with data or text files. There are several methods for reading from files and writing to files; for now we will use the **load** function to read and the **save** function to write to files.

Writing Data to a File

The **save** function can be used to write data from a matrix to a data file, or to append to a data file. The format is:

save filename matrixvariablename –ascii.

The -ascii qualifier is used when creating a text or data file. The following creates a matrix and then saves the values of the matrix variable to a data file called testfile.dat:

```
>> mymat = rand(2,3)
mymat =
0.4565        0.8214        0.6154
0.0185        0.4447        0.7919
>> save testfile.dat mymat –ascii
```

This creates a file called testfile.dat that stores the numbers

```
0.4565        0.8214        0.6154
0.0185        0.4447        0.7919
```

The **type** command can be used to display the contents of the file; notice that scientific notation is used:

```
>> type testfile.dat
4.5646767e–001     8.2140716e–001     6.1543235e–001
1.8503643e–002    4.4470336e–001     7.9193704e–001
```

Note: If the file already exists, the **save** function will overwrite it; **save** always begins writing from the beginning of a file.

Appending Data to a Data File

Once a text file exists, data can be appended to it. The format is the same as previously, with the addition of the qualifier -append. For example, the following creates a new random matrix and appends it to the file just created:

```
>> mymat = rand(3,3)
mymat =
0.9218 0.4057 0.4103
0.7382 0.9355 0.8936
0.1763 0.9169 0.0579
>> save testfile.dat mymat –ascii –append
```

This results in the file testfile.dat containing

```
0.4565        0.8214        0.6154
0.0185        0.4447        0.7919
0.9218        0.4057        0.4103
0.7382        0.9355        0.8936
0.1763        0.9169        0.0579
```

Note: Although technically any size matrix could be appended to this data file, in order to be able to read it back into a matrix later there would have to be the same number of values on every row.

Reading from a File

Once a file has been created (as previously), it can be read into a matrix variable. If the file is a data file, the **load** function will read from the file filename.ext (e.g., the extension might be .dat) and create a matrix with the same name as the file. For example, if the data file testfile.dat had been created as shown in the previous section, this would read from it:

```
>> clear
>> load testfile.dat
>> who
Your variables are:
testfile
>> testfile
testfile =
0.4565          0.8214          0.6154
0.0185          0.4447          0.7919
0.9218          0.4057          0.4103
0.7382          0.9355          0.8936
0.1763          0.9169          0.0579
```

Note: The **load** command works only if there are the same number of values in each line, so that the data can be stored in a matrix, and the **save** command only writes from a matrix to a file. If this is not the case, lower-level file I/O functions must be used

Cell Arrays

One type of data structure that MATLAB has but is not found in many programming languages is a ***cell array***. A cell array in MATLAB is an array, but unlike the vectors and matrices we have used so far, elements in cell arrays can store different types of values.

Creating Cell Arrays

There are several ways to create cell arrays. For example, we will create a cell array in which one element will store an integer, one element will store a character, one element will store a vector, and one element will store a string. Just as with the arrays we have seen so far, this could be a 1x 4 row vector, a 4x 1 column vector, or a 2 x2 matrix. The syntax for creating vectors and matrices is the same as before. Values within rows are separated by spaces or commas, and rows are separated by semicolons. However, for cell arrays, curly braces are used rather than square brackets. For example, the following creates a row vector cell array with the four different types of values:

```
>> cellrowvec = {23, ‘a’, 1:2:9, ‘hello’}
cellrowvec =
[23]     ‘a’      [1x5 double]     ‘hello’
```

To create a column vector cell array, the values are instead separated by semicolons:

```
>> cellcolvec = {23; ‘a’; 1:2:9; ‘hello’}
cellcolvec =
[ 23]
```

```
'a'
[1x5 double]
'hello'
```

This method creates a 2 2 cell array matrix:

```
>> cellmat = {23 'a'; 1:2:9 'hello'}
cellmat =
[ 23]           'a'
[1x5 double]    'hello'
```

Another method of creating a cell array is simply to assign values to specific array elements and build it up element by element. However, as explained before, extending an array element by element is a very inefficient and time-consuming method. It is much more efficient, if the size is known ahead of time, to preallocate the array. For cell arrays, this is done with the **cell** function. For example, to preallocate a variable *mycellmat* to be a 2 2 cell array, the **cell** function would be called as follows:

```
>> mycellmat = cell(2,2)
mycellmat =
[ ] [ ]
[ ] [ ]
```

Note that this is a function call so the arguments to the function are in parentheses. This creates a matrix in which all the elements are empty vectors. Then, each element can be replaced by the desired value. How to refer to each element in order to accomplish this will be explained next.

Referring to and Displaying Cell Array Elements and Attributes

Just as with the other vectors we have seen so far, we can refer to individual elements of cell arrays. The only difference is that curly braces are used for the subscripts. For example, this refers to the second element of the cell array *cellrowvec*:

```
> cellrowvec{2}
ans =
a
```

Row and column indices are used to refer to an element in a matrix (again using curly braces), for example,

```
>> cellmat{1,1}
ans =
23
```

Values can be assigned to cell array elements. For example, after preallocating the variable *mycellmat* in the previous section, the elements can be initialized:

```
>> mycellmat{1,1} = 23
mycellmat =
[23] [ ]
[   ] [ ]
```

When an element of a cell array is itself a data structure, only the type of the element is displayed when the cell array contents are shown. For example, in the cell arrays just created, the vector is shown just as *1 5* **double**. Referring to that element specifically would display its contents, for example,

```
>> cellmat{2,1}
ans =
1       3       5       7       9
```

Since this element is a vector, parentheses are used to refer to its elements. For example, the fourth element of the preceding vector is:

```
>> cellmat{2,1}(4)
ans =
7
```

Notice that the index into the cell array is given in curly braces, and then parentheses
are used to refer to an element of the vector.
We can also refer to subsets of cell arrays, for example,

```
>> cellcolvec{2:3}
ans =
a
ans =
1       3       5       7       9
```

Notice, however, that MATLAB stored *cellcolvec{2}* in the default variable *ans*, and then replaced that with the value of *cellcolvec{3}*. This is because the two values are different types, and therefore cannot be stored together in *ans*. However, they could be stored in two separate variables by having a vector of variables on the left-hand side of an assignment.

```
>> [c1 c2] = cellcolvec{2:3}
c1 =
a
c2 =
1       3       5       7       9
```

There are several methods of displaying cell arrays. The **celldisp** function displays all elements of the cell array:

```
>> celldisp(cellrowvec)
cellrowvec{1} =
23
cellrowvec{2} =
a
cellrowvec{3} =
1       3       5       7       9
cellrowvec{4} =
hello
```

The function **cellplot** puts a graphical display of the cell array in a Figure Window; however, it is a high-level view and basically just displays the same information as typing the name of the variable (e.g., it wouldn't show the contents of the vector in the previous example). Many of the functions and operations on arrays that we have already seen also work with cell arrays. For example, here are some related to dimensioning:

```
>> length(cellrowvec)
ans =
 4
>> size(cellcolvec)
ans =
4          1
>> cellrowvec{end}
ans =
hello
```

It is not possible to delete an individual element from a cell array. For example, assigning an empty vector to a cell array element does not delete the element, it just replaces it with the empty vector:

```
>> cellrowvec
mycell =
[23]       [1x5 double]      'hello'
>> length(cellrowvec)
ans =
4
>> cellrowvec{2} = [ ]
mycell =
[23]â      []          [1x5 double]      'hello'

>> length(cellrowvec)
ans =
4
```

However, it is possible to delete an entire row or column from a cell array by assigning the empty vector (**Note**: use parentheses rather than curly braces to refer to the row or column):

```
>> cellmat
mycellmat =
[           23]       'a'
[1x5 double]       'hello'
>> cellmat(1,:) = []
mycellmat =
[1x5 double]      'hello'
```

Storing Strings in Cell Arrays

One good application of a cell array is to store strings of different lengths. Since cell arrays can store different types of values in the elements, that means strings of different lengths can be stored in the elements.

```
>> names = {'Sue', 'Cathy', 'Xavier'}
names =
 'Sue'    'Cathy'             'Xavier'
```

This is extremely useful, because unlike vectors of strings created using **char** or **strvcat**, these strings do not have extra trailing blanks. The length of each string can be displayed using a **for** loop to loop through the elements of the cell array:

```
>> for i = 1:length(names)
disp(length(names{i}))
end
3
5
6
```

It is possible to convert from a cell array of strings to a character array, and vice versa. MATLAB has several functions that facilitate this. For example, the function **cellstr** converts from a character array padded with blanks to a cell array in which the trailing blanks have been removed.

```
>> greetmat = char('Hello','Goodbye');
>> cellgreets = cellstr(greetmat)
cellgreets =
'Hello'
'Goodbye'
```

The **char** function can convert from a cell array to a character matrix:

```
>> names = {'Sue', 'Cathy', 'Xavier'};
>> cnames = char(names)
cnames =
Sue
Cathy
Xavier

>> size(cnames)
ans =
3        6
```

The function **iscellstr** will return logical true if a cell array is a cell array of all strings, or logical false if not.

```
>> iscellstr(names)
ans =
1

>> iscellstr(cellcolvec)
ans =
0
```

We will see several examples of cell arrays containing strings of varying lengths in the coming chapters, including advanced file input functions and customizing plots.

Structures

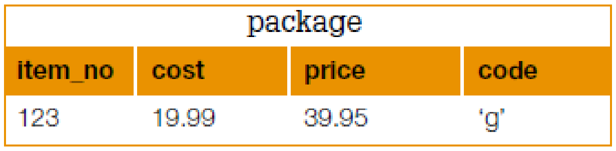
Structures are data structures that group together values that are logically related in what are called ***fields*** of the structure. An advantage of structures is that the fields are named, which helps to make it clear what the values are that are stored in the structure. However, structure variables are not arrays. They do not have elements, so it is not possible to loop through the values in a structure.

Creating and Modifying Structure Variables

Creating structure variables can be accomplished by simply storing values in fields using assignment statements, or by using the **struct** function. The first example that will be used is that the local Computer Super Mart wants to store information on the software packages that it sells. For every one, they will store:

- The item number
- The cost to the store
- The price to the customer
- A code indicating the type of software

An individual structure variable for one software package might look like this:



package			
item_no	**cost**	**price**	**code**
123	19.99	39.95	‘g’

The name of the structure variable is *package*; it has four fields called item_no, cost, price, and code. One way to initialize a structure variable is to use the **struct** function, which preallocates the structure. The field names are passed to the struct in quotes, following each one with the value for that field:

```
>> package = struct('item_no',123,'cost',19.99,…
'price',39.95,'code','g')
package =
          item_no:      123
          cost:         19.9900
          price:        39.9500
          code:         'g'
```

Typing the name of the structure variable will display the names and contents of all fields:

```
>> package
package =
          item_no:      123
          cost:         19.9900
          price:        39.9500
          code:         'g'
```

Notice that in the Workspace Window, the variable *package* is listed as a 1 x1 struct. MATLAB, since it is written to work with arrays, assumes the array format. Just as a single number is treated as a 1 x1 double, a single structure is treated as a 1x1 struct. Later in this chapter we will see how to work more generally with vectors of structs. An alternative method of creating this structure, which is not as efficient, involves using the ***dot operator*** to refer to fields within the structure. The name of the structure variable is followed by a dot, or period, and then the name of the field within that structure. Assignment statements can be used to assign values to the fields.

```
>> package.item_no = 123;
>> package.cost = 19.99;
>> package.price = 39.95;
>> package.code = ‘g’;
```

By using the dot operator in the first assignment statement, a structure variable is created with the field *item_no*. The next three assignment statements add more fields to the structure variable. Adding a field to a structure later is accomplished as shown earlier, by using an assignment statement. An entire structure variable can be assigned to another. This would make sense, for example, if the two structures had some values in common. Here, for example, the values from one structure are copied into another and then two fields are selectively changed.

```
>> newpack = package;
>> newpack.item_no = 111;
>> newpack.price = 34.95
newpack =
        item_no:        111
        cost:           19.9900
        price:          34.9500
        code:           ‘g’
```

To print from a structure, the **disp** function will display either the entire structure or a field.

```
>> disp(package)
item_no: 123
cost: 19.9900
price: 39.9500
code: ‘g’

>> disp(package.cost)
19.9900
```

However, using **fprintf**, only individual fields can be printed; the entire structure cannot be printed.

```
>> fprintf(‘%d %c\n’, package.item_no, package.code)
123g
```

The function **rmfield** removes a field from a structure. It returns a new structure with the field removed, but does not modify the original structure (unless the returned structure is assigned to that variable). For example, the following would remove the *code* field from the *newpack* structure, but store the resulting structure in the default variable *ans*. The value of *newpack* remains unchanged.

```
>> rmfield(newpack, ‘code’)
ans =
        item_no: 111
        cost: 19.9900
        price: 34.9500
>> newpack
newpack =
        item_no: 111
        cost: 19.9000
        price: 34.9500
        code: ‘g’
```

To change the value of *newpack*, the structure that results from calling **rmfield** must be assigned to *newpack*.

```
>> newpack = rmfield(newpack, ‘code’)
newpack =
        item_no:        111
        cost:           19.9000
        price:          34.9500
```

Passing Structures to Functions

An entire structure can be passed to a function, or individual fields can be passed. For example, here are two different versions of a function that calculates the profit on a software package. The profit is defined as the price minus the cost. In the first version, the entire structure variable is passed to the function, so the function must use the dot operator to refer to the *price* and *cost* fields of the input argument.

```
calcprof.m
function profit = calcprof(packstruct)
% Calculates the profit for a software package
% The entire structure is passed to the function
profit = packstruct.price – packstruct.cost;

>> calcprof(package)
ans =
19.9600
```

In the second version, just the *price* and *cost* fields are passed to the function using the dot operator in the function call. These are passed to two scalar input arguments in the function header, so there is no reference to a structure variable in the function itself, and the dot operator is not needed in the function.

```
calcprof2.m
function profit = calcprof2(oneprice, onecost)
% Calculates the profit for a software package
% The individual fields are passed to the function
profit = oneprice – onecost;
>> calcprof2(package.price, package.cost)
ans =
19.9600
```

It is important, as always with functions, to make sure that the arguments in the function call correspond one-to-one with the input arguments in the function header. In the case of *calcprof*, a structure variable is passed to an input argument, which is a structure. For the second function *calcprof2*, two individual fields, which are **double** values, are passed to two **double** arguments.

Related Structure Functions

There are several functions that can be used with structures in MATLAB. The function **isstruct** will return 1 for logical true if the variable argument is a structure variable, or 0 if not. The **isfield** function returns logical true if a fieldname (as a string) is a field in the structure argument, or logical false if not

```
>> isstruct(package)
ans =
1
>> isfield(package,'cost')
ans =
1
```

The **fieldnames** function will return the names of the fields that are contained in a structure variable.

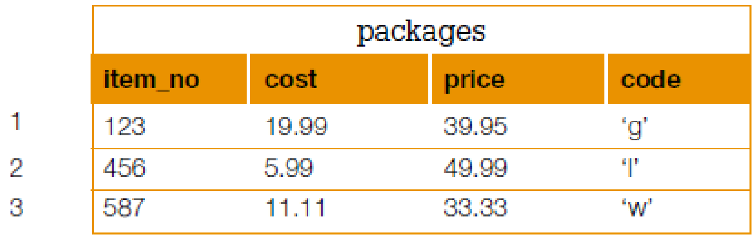
```
>> pack_fields = fieldnames(package)
pack_fields =
'item_no'
'cost'
'price'
'code'
```

Since the names of the fields are of varying lengths, the **fieldnames** function returns a cell array with the names of the fields. Curly braces are used to refer to the elements, since pack_fields is a cell array. For example, we can refer to the length of one of the strings:

```
>> length(pack_fields{2})
ans =
4
```

Vectors of Structures

In many applications, including database applications, information normally would be stored in a ***vector of structures***, rather than in individual structure variables. For example, if the Computer Super Mart is storing information on all the software packages that it sells, it would likely be in a vector of structures, for example,



	packages			
	item_no	cost	price	code
1	123	19.99	39.95	'g'
2	456	5.99	49.99	'l'
3	587	11.11	33.33	'w'

In this example, *packages* is a vector that has three elements. It is shown as a column vector. Each element is a structure consisting of four fields, item_no, cost, price, and code. It may look like a matrix with rows and columns, but it is instead a vector of structures. This can be created several ways. One method is to create a structure variable, as shown earlier, to store information on one software package. This can then be expanded to be a vector of structures.

```
>> packages = struct('item_no',123,'cost',19.99,…
'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,…
'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,…
'price',33.33,'code','w');
```

The first assignment statement shown here creates the first structure in the structure vector, the next one creates the second structure, and so on. This actually creates a 1x3 row vector. Alternatively, the first structure could be treated as a vector to begin with, for example,

```
>> packages(1) = struct('item_no',123,'cost',19.99,…
'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,…
'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,…
'price',33.33,'code','w');
```

Both of these methods, however, involve extending the vector. As we have already seen, preallocating any vector in MATLAB is more efficient than extending it. There are several methods of preallocating the vector. By starting with the last element, MATLAB would create a vector with that many elements. Then, the elements from 1 through end-1 could be initialized. For example, for a vector of structures that has three elements, start with the third element.

```
>> packages(3) = struct('item_no',587,'cost',11.11,.â•›.â•›.
'price',33.33,'code','w');
>> packages(1) = struct('item_no',123,'cost',19.99,.â•›.â•›.
'price',39.95,'code','g');
>> packages(2) = struct('item_no',456,'cost', 5.99,.â•›.â•›.
'price',49.99,'code','l');
```

Another method is to create one element with the values from one structure, and use **repmat** to replicate it to the desired size. Then, the remaining elements can be modified. The following creates one structure and then replicates this into a 1x3 matrix.

```
>> packages = repmat(struct('item_no',587,'cost',…
11.11, 'price',33.33,'code','w'), 1,3);
>> packages(2) = struct('item_no',456,'cost', 5.99,…
'price',49.99,'code','l');
>> packages(3) = struct('item_no',587,'cost',11.11,…
'price',33.33,'code','w');
```

Typing the name of the variable will display only the size of the structure vector and the names of the fields:

```
>> packages
packages =
1x3 struct array with fields:
item_no
cost
price
code
```

The variable *packages* is now a vector of structures, so each element in the vector is a structure. To display one element in the vector (one structure), an index into the vector would be specified. For example, to refer to the second element:

```
>> packages(2)
ans =
item_no: 456
cost: 5.9900
price: 49.9900
code: 'l'
```

To refer to a field, it is necessary to refer to the particular structure, and then the field within it. This means using an index into the vector to refer to the structure, and then the dot operator to refer to a field. For example:

```
>> packages(1).code
ans =
g
```

So, there are essentially three levels to this data structure. The variable *packages* is the highest level, which is a vector of structures. Each of its elements is an individual structure. The fields within these individual structures are the lowest level. The following loop displays each element in the *packages* vector.

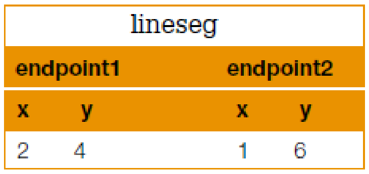
```
>> for i = 1:length(packages)
disp(packages(i))
end

item_no: 123
cost: 19.9900
price: 39.9500
code: 'g'

item_no: 456
cost: 5.9900
price: 49.9900
code: 'l'

item_no: 587
cost: 11.1100
```

```
price: 33.3300
code: ‘w’
```

To refer to a particular field for all structures, in most programming languages it would be necessary to loop through all elements in the vector and use the dot operator to refer to the field for each element. However, this is not the case in MATLAB.

Nested Structures

A ***nested structure*** is a structure in which at least one member is itself a structure. For example, a structure for a line segment might consist of fields representing the two points at the ends of the line segment. Each of these points would be represented as a structure consisting of the x- and y-coordinates.



lineseg			
endpoint1		**endpoint2**	
x	**y**	**x**	**y**
2	4	1	6

This shows a structure variable called *lineseg* that has two fields, *endpoint1* and *endpoint2*. Each of these is a structure consisting of two fields, *x* and *y*. One method of defining this is to nest calls to the **struct** function:

```
>> lineseg = struct(‘endpoint1’,struct(‘x’,2,‘y’,4), …
‘endpoint2’,struct(‘x’,1,‘y’,6))
```

This method is the most efficient. However, another method is to build the nested structure one field at a time. Since this is a nested structure with one structure inside of another, the dot operator must be used twice here to get to the actual x- and y-coordinates.

```
>> lineseg.endpoint1.x = 2;
>> lineseg.endpoint1.y = 4;
>> lineseg.endpoint2.x = 1;
>> lineseg.endpoint2.y = 6;
```

Once the nested structure has been created, we can refer to different parts of the variable *lineseg*. Just typing the name of the variable shows only that it is a structure consisting of two fields, *endpoint1* and *endpoint2*, each of which is a structure.

```
>> lineseg
lineseg =
endpoint1: [1x1 struct]
endpoint2: [1x1 struct]
```

Typing the name of one of the nested structures will display the field names and values within that structure:

```
>> lineseg.endpoint1
```

```
ans =
x: 2
y: 4
```

Using the dot operator twice will refer to an individual coordinate, for example,

```
>> lineseg.endpoint1.x
ans =
            2
```

A nested structure variable for a line segment could also be created by creating structure variables for the points first, and then storing these in the two fields of a line segment variable. For example:

```
>> pointone = struct('x', 5, 'y', 11);
>> pointtwo = struct('x', 7, 'y', 9);
>> myline = struct('endpoint1', pointone,…
'endpoint2', pointtwo)

myline =
endpoint1: [1x1 struct]
endpoint2: [1x1 struct]
```

Then, referring to different parts of the variable would work the same:

```
>> myline.endpoint1
ans =
 x: 5
 y: 11

>> myline.endpoint2.x
ans =
7
```

UNIT II **PROGRAMMING IN MATLAB** **10 hrs.**

Relational and logical operators – Control statements IF-END, IF-ELSE – END, ELSEIF, SWITCH CASE – FOR loop – While loop – Debugging – Applications to Simulation – miscellaneous MAT lab functions & Variables.

UNIT II

Relational Expressions

Conditions in **if** statements use expressions that are conceptually, or logically, either true or false. These expressions are called ***relational expressions***, or sometimes ***Boolean*** or ***logical*** expressions. These expressions can use both ***relational operators***, which relate two expressions of compatible types, and ***logical operators***, which operate on logical operands.

The relational operators in MATLAB are:

Operator	**Meaning**
>	greater than
<	less than
>=	greater than or equals
<=	less than or equals
==	equality
~=	inequality

All concepts should be familiar, although the operators used may be different from those used in other programming languages, or in mathematics classes. In particular, it is important to note that the operator for equality is two consecutive equal signs, not a single equal sign (recall that the single equal sign is the assignment operator). For numerical operands, the use of these operators is straightforward.
For example,
3 < 5 means "3 less than 5,"

which is conceptually a true expression. However, in MATLAB, as in many programming languages, ***logical true*** is represented by the integer 1, and ***logical false*** is represented by the integer 0. So, the expression

3 < 5 actually has the value 1 in MATLAB.

Displaying the result of expressions like this in the Command Window demonstrates the values of the expressions.

```
>> 3 < 5
ans =
1
>> 9 < 2
ans =
0
```

However, in the Workspace Window, the value shown for the result of these expressions would be true or false. The type of the result is **logical**.

Mathematical operations could be performed on the resulting 1 or 0.

```
>> 5 < 7
ans =
1
>> ans + 3
ans =
4
```

Comparing characters, for example ‘a’ < ‘c’, is also possible. Characters are compared using their ASCII equivalent values. So, ‘a’ < ‘câ•›’ is conceptually a true expression, because the character ‘a’ comes before the character ‘c’.

```
>> ‘a’ < ‘c’
ans =
1
```

The logical operators are:

Operator	**Meaning**
||	or for scalars
&&	and for scalars
~	not

All logical operators operate on logical or Boolean operands. The **not** operator is a unary operator; the others are binary. The **not** operator will take a Boolean expression, which is conceptually true or false, and give the opposite value. For example, (3 < 5) is conceptually false since (3 < 5) is true. The **or** operator has two Boolean expressions as operands. The result is true if either or both of the operands are true, and false only if both operands are false. The **and** operator also operates on two Boolean operands. The result of an **and** expression is true only if both operands are true; it is false if either or both are false. In addition to these logical operators, MATLAB also has a function **xor**, which is the exclusive or function. It returns logical true if one (and only one) of the arguments is true. For example, in the following only the first argument is true,

so the result is true:

```
>> xor(3 < 5, ‘a’ > ‘c’)
ans =
1
```

In this example, both arguments are true so the result is false:

```
>> xor(3 < 5, ‘a’ < ‘c’)
ans =
0
```

Given the logical values of true and false in variables x and y, the ***truth table*** shows how the logical operators work for all combinations. Note that the logical operators are commutative (e.g., x || y is the same as y || x).

Table 3.1 Truth Table for Logical Operators

x	y	~x	x || y	x && y	xor(x,y)
true	true	false	true	true	false
true	false	false	true	false	true
false	false	true	false	false	false

As with the numerical operators, it is important to know the operator precedence rules. Table 3.2 shows the rules for the operators that have been covered so far, in the order of precedence.

Table 3.2 Operator Precedence Rules

Operators	Precedence
parentheses ( )	highest
transpose and power', ^	
unary negation (–), not (~)	
multiplication, division *,/,\	
addition, subtraction +, –	
colon operator:	
relational <, <=, >, >=, ==, ~=	
and &&	
or | |	
assignment =	lowest

T he If Statement

The **if** statement chooses whether or not another statement, or group of statements, is executed. The general form of the **if** statement is:

```
if condition
        action
end
```

A ***condition*** is a relational expression that is conceptually, or logically, either true or false. The ***action*** is a statement, or a group of statements, that will be executed if the condition is true. When the **if** statement is executed, first the condition is evaluated. If the value of the condition is conceptually true, the action will be executed, and if not, the action will not be executed. The action can be any number of statements until the reserved word **end**; the action is naturally bracketed by the reserved words **if** and **end**. (Note: This is different from the **end** that is used as an index into a vector or matrix.)

For example, the following **if** statement checks to see whether the value of a variable is negative. If it is, the value is changed to a positive number by using the absolute value function; otherwise nothing is changed.

```
if num < 0
        num = abs(num)
end
```

If statements can be entered in the Command Window, although they generally make more sense in scripts or functions. In the Command Window, the **if** line would be entered, then the Enter key, then the action, the Enter key, and finally **end** and Enter; the results will immediately follow. For example, the previous **if**

statement is shown twice here. Notice that the output from the assignment is not suppressed, so the result of theÂ€action will be shown if the action is executed. The first time the value of the variable is negative so the action is executed and the variable is modified, but in the second case the variable is positive so the action is skipped.

```
>> num = −4;
>> if num < 0
        num = abs(num)
end
num =
4

>> num = 5;
>> if num < 0
        num = abs(num)
end
>>
```

This may be used, for example, to make sure that the square root function is not used on a negative number. The following script prompts the user for a number, and prints the square root. If the user enters a negative number, the **if** statement changes it to positive before taking the square root.

```
sqrtifexamp.m
% Prompt the user for a number and print its sqrt
num = input(‘Please enter a number: ’);
% If the user entered a negative number, change it
if num < 0
num = abs(num);
end
fprintf(‘The sqrt of %.1f is %.1f\n’,num,sqrt(num))
```

Here are two examples of running this script:

```
>> sqrtifexamp
Please enter a number: −4.2
The sqrt of 4.2 is 2.0
>> sqrtifexamp
Please enter a number: 1.44
The sqrt of 1.4 is 1.2
```

In this case, the action of the **if** statement was a single assignment statement. The action can be any number of valid statements. For example, we may wish to print a note to the user to say that the number entered was being changed.

```
sqrtifexampii.m
% Prompt the user for a number and print its sqrt
num = input(‘Please enter a number: ’);
```

```
% If the user entered a negative number, tell
% the user and change it
if num < 0
disp('OK, we''ll use the absolute value')
num = abs(num);
end
fprintf('The sqrt of %.1f is %.1f\n',num,sqrt(num))
```

```
>> sqrtifexampii
Please enter a number: −25
OK, we'll use the absolute value
The sqrt of 25.0 is 5.0
```

Notice the use of two single quotes in the **disp** statement in order to print one single quote

Representing Logical True and False

It has been stated that expressions that are conceptually true actually have the integer value of 1, and expressions that are conceptually false actually have the integer value of 0. Representing the concepts of logical true and false in MATLAB is slightly different: the concept of false is represented by the integer value of 0, but the concept of true can be represented by any nonzero value (not just the integer 1). This can lead to some strange Boolean expressions. For example, consider the following **if** statement:

```
>> if 5
disp('Yes, this is true!')
end
Yes, this is true!
```

Since 5 is a nonzero value, it is a way of saying true. Therefore, when this Boolean expression is evaluated, it will be true, so the **disp** function will be executed and "Yes, this is true" is displayed. Of course, this is a pretty bizarre **if** statement, one that hopefully would not ever be encountered! However, a simple mistake in an expression can lead to this kind of result. For example, let's say that the user is prompted for a choice of Y or N for a yes/no

question:

```
letter = input('Choice (Y/N): ','s');
```

In a script we might want to execute a particular action if the user responded with 'Y'. Most scripts would allow the user to enter either lower- or uppercase (e.g., either 'y' or 'Y') to indicate yes. The proper expression that would return true if the value of letter was 'y' or 'Y' would be

```
letter == 'y' || letter == 'Y'
```

However, if by mistake this was written as:

```
letter == 'y' || 'Y'
```

this expression would *always* be true, regardless of the value of the variable *letter*. This is because 'Y' is a nonzero value, so it is a true expression. The first part of the expression may be false, but since the second expression is true the entire expression would be true.

The If-Else statement

The **if** statement chooses whether an action is executed or not. Choosing between two actions, or choosing from several actions, is accomplished using **if-else**, nested **if**, and **switch** statements. The **if-else** statement is used to choose between two statements, or sets of statements.

The general form is:

```
if condition
action1
else
action2
end
```

First, the condition is evaluated. If it is conceptually true, then the set of statements designated as action1 is executed, and that is it for the **if-else** statement. If instead the condition is conceptually false, the second set of statements designated as action2 is executed, and that's it. The first set of statements is called the action of the **if** clause; it is what will be executed if the expression is true. The second set of statements is called the action of the **else** clause; it is what will be executed if the expression is false. One of these actions, and only one, will be executed—which one depends on the value of the condition. For example, to determine and print whether or not a random number in the range from 0 to 1 is less than 0.5, an **if-else** statement could be used:

```
if rand < 0.5
disp('It was less than .5!')
else
disp('It was not less than .5!')
end
```

One application of an **if-else** statement is to check for errors in the inputs to a script. For example, an earlier script prompted the user for a radius, and then used that to calculate the area of a circle. However, it did not check to make sure that the radius was valid (e.g., a positive number). Here is a modified script that checks the radius:

```
checkradius.m
% This script calculates the area of a circle
% It error-checks the user's radius
radius = input('Please enter the radius: ');
if radius <= 0
        fprintf('Sorry; %.2f is not a valid radius\n',radius)
else
        area = calcarea(radius);
        fprintf('For a circle with a radius of %.2f,',radius)
        fprintf('the area is %.2f\n',area)
end
```

Examples of running this script when the user enters invalid and then valid radii are shown here:

```
>> checkradius
Please enter the radius: −4
```

```
Sorry; −4.00 is not a valid radius

>> checkradius
Please enter the radius: 5.5
For a circle with a radius of 5.50, the area is 95.03
```

The **if-else** statement in this example chooses between two actions: printing an error message, or actually using the radius to calculate the area, and then printing out the result. Notice that the action of the **if** clause is a single statement, whereas the action of the **else** clause is a group of three statements.

Nested If-Else **Statements**

The **if-else** statement is used to choose between two statements. In order to choose from more than two statements, the **if-else** statements can be nested, one inside of another. For example, consider implementing the following continuous

mathematical function y = f(x):
$y = 1$ for $x < -1$
$y = x2$ for $-1 \leq x \leq 2$
$y = 4$ for $x > 2$

The value of y is based on the value of x, which could be in one of three possible ranges. Choosing which range could be accomplished with three separate **if** statements, as follows:

```
if x < −1
            y = 1;
end
if x > = −1 && x < = 2
             y = x^2;
end
if x > 2
             y = 4;
end
```

Since the three possibilities are mutually exclusive, the value of *y* can be determined by using three separate **if** statements. However, this is not very efficient code: all three Boolean expressions must be evaluated, regardless of the range in which *x* falls. For example, if *x* is less than –1, the first expression is true and 1 would be assigned to *y*. However, the two expressions in the next two **if** statements are still evaluated. Instead of writing it this way, the expressions can be ***nested*** so that the statement ends when an expression is found to be true:

```
if x < −1
             y = 1;
else
             % If we are here, x must be > = −1
             % Use an if-else statement to choose
             % between the two remaining ranges
             if x > = −1 && x < = 2
```

```
        y = x^2;
        else
        % No need to check
        % If we are here, x must be > 2
        y = 4;
        end
end
```

By using a nested **if-else** to choose from among the three possibilities, not all conditions must be tested as they were in the previous example. In this case, if *x* is less than –1, the statement to assign 1 to *y* is executed, and the **if-else** statement is completed so no other conditions are tested. If, however, *x* is not less than –1, then the else clause is executed. If the else clause is executed, then we already know that *x* is greater than or equal to –1 so that part does not need to be tested. Instead, there are only two remaining possibilities: either *x* is less than or equal to 2, or it is greater than 2. An **if-else** statement is used to choose between those two possibilities. So, the action of the **else** clause was another **if-else** statement. Although it is long, this is one **if-else** statement, a nested **if-else** statement. The actions are indented to show the structure. Nesting **if-else** statements in this way can be used to choose from among three, four, five, six, or more options—the possibilities are practically endless! This is actually an example of a particular kind of nested **if-else** called a cascading **if-else** statement. In this type of nested **if-else** statement, the conditions and actions cascade in a stair-like pattern.

For example, if there are *n* choices (where *n* > 3 in this example), the following general form would be used:

```
if condition1
        action1
elseif condition2
        action2
elseif condition3
        action3
% etc: there can be many of these
else
        actionn % the nth action
end
```

The actions of the **if**, **elseif**, and **else** clauses are naturally bracketed by the reserved words **if**, **elseif**, **else**, and **end**. For example, the previous example could be written using the **elseif** clause rather than nesting **if-else** statements:

```
if x < −1
        y = 1;
elseif x > = −1 && x < = 2
        y = x^2;
else
        y = 4;
end
```

So, there are three ways of accomplishing this task: using three separate **if** statements, using nested **if-else** statements, and using an **if** statement with **elseif** clauses, which is the simplest. This could be implemented in a function that receives a value of x and returns the corresponding value of y:

```
calcy.m
function y = calcy(x)
% This function calculates y based on x:
% y = 1 for x < −1
% y = x2 for −1 ≤ x ≤ 2
% y = 4 for x > 2
if x < −1
        y = 1;
elseif x >= −1 && x <=2
        y = x^2;
else
        y = 4;
end
>> x = 1.1;
>> y = calcy(x)
y =
        1.2100
```

Another example demonstrates choosing from more than just a few options. The following function receives an integer quiz grade, which should be in the range from 0 to 10. The program then returns a corresponding letter grade, according to the following scheme: a 9 or 10 is an ‘A’, an 8 is a ‘B’, a 7 is a ‘C’, a 6 is a ‘D’, and anything below that is an ‘F’. Since the possibilities are mutually exclusive, we could implement the grading scheme using separate **if** statements. However, it is more efficient to have one **if-else** statement with multiple **elseif** clauses. Also, the function returns the value ‘X’ if the quiz grade is not valid. The function does assume that the input is an integer.

```
letgrade.m
function grade = letgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument
% First, error-check
if quiz < 0 || quiz > 10
        grade = ‘X’;
% If here, it is valid so figure out the
% corresponding letter grade
elseif quiz == 9 || quiz == 10
grade = ‘A’;
elseif quiz == 8
        grade = ‘B’;
elseif quiz == 7
        grade = ‘C’;
elseif quiz == 6
        grade = ‘D’;
else
```

```
        grade = 'F';
end
```

Here are three examples of calling this function:

```
>> quiz = 8;
>> lettergrade = letgrade(quiz)
lettergrade =
B
>> quiz = 4;
>> letgrade(quiz)
ans =
F
>> quiz = 22;
>> lg = letgrade(quiz)
lg =
X
```

In the part of this **if** statement that chooses the appropriate letter grade to return, all the Boolean expressions are testing the value of the variable *quiz* to see if it is equal to several possible values, in sequence (first 9 or 10, then 8, then 7, etc.). This part can be replaced by a **switch** statement.

The Switch Statement

A **switch** statement can often be used in place of a nested **if-else** or an **if** statement with many **elseif** clauses. Switch statements are used when an expression is tested to see whether it is *equal to* one of several possible values. The general form of the **switch** statement is:

```
switch switch_expression
case caseexp1
        action1
case caseexp2
        action2
 case caseexp3
        action3
 % etc: there can be many of these
otherwise
        actionn
end
```

The **switch** statement starts with the reserved word **switch**, and ends with the reserved word **end**. The switch_expression is compared, in sequence, to the case expressions (caseexp1, caseexp2, etc.). If the value of the switch_expression matches caseexp1, for example, then action1 is executed and the **switch** statement ends. If the value matches caseexp3, then action3 is executed, and in general if the value matches caseexpi, where i can be any integer from 1 to n, then actioni is executed. If the value of the switch_expression does not match any of the case expressions, the action after the word **otherwise** is executed. For the previous example, the **switch** statement can be used as follows:

```
switchletgrade.m
function grade = switchletgrade(quiz)
% This function returns the letter grade corresponding
% to the integer quiz grade argument using switch
% First, error-check
if quiz < 0 || quiz > 10
        grade = 'X';
else
        % If here, it is valid so figure out the
        % corresponding letter grade using a switch
        switch quiz
        case 10
                grade = 'A';
        case 9
                grade = 'A';
        case 8
                grade = 'B';
        case 7
                grade = 'C';
        case 6
                grade = 'D';
        otherwise
                grade = 'F';
        end
end
```

Here are two examples of calling this function:

```
>> quiz = 22;
>> lg = switchletgrade(quiz)
lg =
X
>> quiz = 9;
>> switchletgrade(quiz)
ans =
A
```

Note that it is assumed that the user will enter an integer value. If the user does not, either an error message will be printed or an incorrect result will be returned. Since the same action of printing 'A' is desired for more than one case, these can be combined as follows:

```
switch quiz
        case {10,9}
                grade = 'A';
        case 8
                grade = 'B';
        % etc.
```

(The curly braces around the case expressions 10 and 9 are necessary.) In this example, we error-checked first using an **if-else** statement, and then if the grade was in the valid range, used a **switch** statement to find the corresponding letter grade.

Sometimes the **otherwise** clause is used instead for the error message. For example, if the user is supposed to enter only a 1, 3, or 5, the script might be organized as follows:

```
switcherror.m
% Example of otherwise for error message
choice = input('Enter a 1, 3, or 5: ');
switch choice
        case 1
                disp('It''s a one!!')
        case 3
                disp('It''s a three!!')
        case 5
                disp('It''s a five!!')
        otherwise
                disp('Follow directions next time!!')
end
```

In this case, actions are taken if the user correctly enters one of the valid options. If the user does not, the **otherwise** clause handles printing an error message. Note the use of two single quotes within the string to print one.

```
>> switcherror
Enter a 1, 3, or 5: 4
Follow directions next time!!
```

The for Loop

The **for** statement, or the **for** loop, is used when it is necessary to repeat statement(s) in a script or function, and when it *is* known ahead of time how many times the statements will be repeated. The statements that are repeated are called the action of the loop. For example, it may be known that the action of the loop will be repeated five times. The terminology used is that we ***iterate*** through the action of the loop five times. The variable that is used to iterate through values is called a ***loop variable***, or an ***iterator variable***. For example, the variable might iterate through the integers 1 through 5 (e.g., 1, 2, 3, 4, and then 5). Although variable names in general should be mnemonic, it is common for an iterator variable to be given the name *i* (and if more than one iterator variable is needed, *i*, *j*, *k*, *l*, etc.) This is historical, and is because of the way integer variables were named in Fortran. However, in MATLAB both **i** and **j** are built-in values for -1 , so using either as a loop variable will override that value. If that is not an issue, then it is acceptable to use *i* as a loop variable. The general form of the **for** loop is:

```
for loopvar = range
        action
end
```

where *loopvar* is the loop variable, *range* is the range of values through which the loop variable is to iterate, and the action of the loop consists of all statements up to the **end**. The range can be specified using any vector, but normally the easiest way to specify the range of values is to use the colon operator. As an example, to print a column of numbers from 1 to 5:

```
for i = 1:5
        fprintf('%d\n',i)
end
```

This loop could be entered in the Command Window, although like **if** and **switch** statements, loops will make more sense in scripts and functions. In the Command Window, the results would appear after the **for** loop:

```
>> for i = 1:5
fprintf('%d\n',i)
end
1
2
3
4
5
```

What the **for** statement accomplished was to print the value of *i* and then the newline character for every value of *i*, from 1 through 5 in steps of 1. The first thing that happens is that *i* is initialized to have the value 1. Then, the action of the loop is executed, which is the **fprintf** statement that prints the value of *i* (1), and then the newline character to move the cursor down. Then, *i* is incremented to have the value of 2. Next, the action of the loop is executed, which prints 2 and the newline. Then, *i* is incremented to 3 and that is printed, then *i* is incremented to 4 and that is printed, and then finally *i* is incremented to 5 and that is printed. The final value of *i* is 5; this value can be used once the loop has finished.

Finding Sums and Products

A very common application of a **for** loop is to calculate sums and products. For example, instead of just printing the integers 1 through 5, we could calculate the sum of the integers 1 through 5 (or, in general, 1 through *n*, where *n* is any positive integer). Basically, we want to implement or calculate the sum 1 + 2 + 3 + … + n. In order to do this, we need to add each value to a ***running sum***. A running sum is a sum that will keep changing; we keep adding to it. First the sum has to be initialized to 0, then in this case it will be 1 (0 + 1), then 3 (0 + 1 + 2), then 6 (0 + 1 + 2 + 3), and so forth. In a function to calculate the sum, we need a loop or iterator variable *i*, as before, and also a variable to store the running sum. In this case we will use the output argument *runsum* as the running sum. Every time through the loop, the next value of *i* is added to the value of *runsum*. This function will return the end result, which is the sum of all integers from 1 to the input argument *n* stored in the output argument *runsum*.

```
sum_1_to_n.m
function runsum = sum_1_to_n(n)
% This function returns the sum of
% integers from 1 to n
runsum = 0;
for i = 1:n
        runsum = runsum + i;
end
```

As an example, if 5 is passed to be the value of the input argument *n*, the function will calculate and return 1 + 2 + 3 + 4 + 5, or 15:

```
>> sum_1_to_n(5)
          ans =
         15
```

Note that the output was suppressed when initializing the sum to 0 and when adding to it during the loop. Another very common application of a **for** loop is to find a ***running product***. For example, instead of finding the sum of the integers 1 through n, we could find the product of the integers 1 through n. Basically, we want to implement or calculate the product 1 * 2 * 3 * 4 *… * n, which is called the ***factorial*** of n, written n!.

For Loops that Do Not Use the Iterator Variable in the Action

In all the examples that we have seen so far, the value of the loop variable has been used in some way in the action of the **for** loop: we have printed the value of *i*, or added it to a sum, or multiplied it by a running product, or used it as an index into a vector. It is not always necessary to actually use the value of the loop variable, however. Sometimes the variable is simply used to iterate, or repeat, a statement a specified number of times. For example,

```
for i = 1:3
          fprintf('I will not chew gum\n')
end
```

produces the output:

```
I will not chew gum
I will not chew gum
I will not chew gum
```

The variable *i* is necessary to repeat the action three times, even though the value of *i* is not used in the action of the loop.

Nested for Loops

The action of a loop can be any valid statement(s). When the action of a loop is another loop, this is called a ***nested loop***. As an example, a nested **for** loop will be demonstrated in a script that will print a box of *'s. Variables in the script will specify how many rows and columns to print. For example, if *rows* has the value 3, and *columns* has the value 5, the

output would be:

```
*****
*****
*****
```

Since lines of output are controlled by printing the newline character, the basic algorithm is:

- For every row of output,
- Print the required number of *'s
- Move the cursor down to the next line (print the '\n')

```
printstars.m
% Prints a box of stars
% How many will be specified by 2 variables
% for the number of rows and columns
rows = 3;
columns = 5;
% loop over the rows
for i=1:rows
          % for every row loop to print *'s and then one \n
          for j=1:columns
fprintf('*')
          end
          fprintf('\n')
end
```

Running the script displays the output:

```
>> printstars
*****
*****
*****
```

The variable *rows* specifies the number of rows to print, and the variable *columns* specifies how many *'s to print in each row. There are two loop variables: *i* is the loop variable for the rows, and *j* is the loop variable for the columns. Since the number of rows and columns are known (given by the variables *rows* and *columns*), **for** loops are used. There is one **for** loop to loop over the rows, and another to print the required number of *'s. The values of the loop variables are not used within the loops, but are used simply to iterate the correct number of times. The first **for** loop specifies that the action will be repeated *rows* times. The action of this loop is to print *'s and then the newline character. Specifically, the action is to loop to print *columns* *'s across on one line. Then, the newline character is printed after all five stars to move the cursor down for the next line. The first **for** loop is called the ***outer loop***; the second **for** loop is called the ***inner loop***. So, the outer loop is over the rows, and the inner loop is over the columns. The outer loop must be over the rows because the program is printing a certain number of rows of output. For each row, a loop is necessary to print the required number of *'s; this is the inner **for** loop. When this script is executed, first the outer loop variable *i* is initialized to 1. Then, the action is executed. The action consists of the inner loop, and then printing the newline character. So, while the outer loop variable has the value 1, the inner loop variable *j* iterates through all its values. Since the value of *columns* is 5, the inner loop will print a * five times. Then, the newline character is printed and the outer loop variable *i* is incremented to 2. The action of the outer loop is then executed again, meaning the inner loop will print five *'s, and then the newline character will be printed. This continues, and in all, the action of the outer loop will be executed *rows* times.
Notice the action of the outer loop consists of two statements (the **for** loop and an **fprintf** statement). The action of the inner loop, however, is only a single statement. The **fprintf** statement to print the newline character must be separate from the other **fprintf** statement that prints the *. If we simply had fprintf('*\n') as the action of the inner loop, this would print a long column of 15 *'s, not a box In these examples, the loop variables were used just to specify the number of times the action is to be repeated. These same loops could be used instead to produce a multiplication table by multiplying the values of the loop variables. The following function *multtable* calculates and returns a matrix that is a multiplication table. Two arguments are passed to the function, which are the number of rows and columns for this matrix

```
multtable.m
function outmat = multtable (rows, columns)
% Creates a matrix which is a multiplication table
% Preallocate the matrix
outmat = zeros(rows,columns);
for i = 1:rows
        for j = 1:columns
                outmat(i,j) = i * j;
        end
end
```

In the following example, the matrix has three rows and five columns:

```
>> multtable(3,5)
ans =
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
```

Notice that this is a function that returns a matrix; it does not print anything. It preallocates the matrix to zeros, and then replaces each element. Since the number of rows and columns are known, **for** loops are used. The outer loop loops over the rows, and the inner loop loops over the columns. The action of the nested loop calculates i * j for all values of *i* and *j*. First, when *i* has the value 1, *j* iterates through the values 1 through 5, so first we are calculating 1 * 1, then 1 * 2, then 1 * 3, then 1 * 4, and finally 1 * 5. These are the values in the first row (first in element (1,1), then (1,2), then (1,3), then (1,4), and finally (1,5)). Then, when *i* has the value 2, the elements in the second row of the output matrix are calculated, as *j* again iterates through the values from 1 through 5. Finally, when *i* has the value 3, the values in the third row are calculated (3 * 1, 3 * 2, 3 * 3, 3 * 4, and 3 * 5). This function could be used in a script that prompts the user for the number of rows and columns, calls this function to return a multiplication table, and writes the resulting matrix to a file:

```
createmulttab.m
% Prompt the user for rows and columns and
% create a multiplication table to store in
% a file mymulttable.dat
num_rows = input('Enter the number of rows: ');
num_cols = input('Enter the number of columns: ');
multmatrix = multtable(num_rows, num_cols);
save mymulttable.dat multmatrix –ascii
```

Here is an example of running this script, and then loading from the file into a matrix in order to verify that the file was created:

```
>> createmulttab
Enter the number of rows: 6
Enter the number of columns: 4
>> load mymulttable.dat
>> mymulttable
mymulttable =
1       2       3       4
2       4       6       8
3       6       9       12
4       8       12      16
5       10      15      20
6       12      18      24
```

Logical Vectors

The relational operators can also be used with vectors and matrices. For example, let's say that there is a vector, and we want to compare every element in the vector to 5 to determine whether it is greater than 5 or not. The result would be a vector (with the same length as the original) with logical true or false values. Assume a variable vec as shown here.

```
>> vec = [5 9 3 4 6 11];
```
In MATLAB, this can be accomplished automatically by simply using the relational operator >.
```
>> isg = vec > 5
isg =
0       1       0       0       1       1
```
Notice that this creates a vector consisting of all logical true or false values. Although this is a vector of ones and zeros, and numerical operations can be done on the vector *isg*, its type is **logical** rather than **double**.
```
>> doubres = isg + 5
ans =
5       6       5       5       6       6
>> whos
Name            Size            Bytes   Class
doubres         1x6 4           8       double array
isg             1x6             6       logical array
vec             1x6             48      double array
```
To determine how many of the elements in the vector *vec* were greater than 5, the **sum** function could be used on the resulting vector *isg*:

```
>> sum(isg)
ans =
3
```

The ***logical vector*** *isg* can also be used to index into the vector. For example, if only the elements from the vector that are greater than 5 are desired:

```
>> vec(isg)
ans =
9       6       11
```
Because the values in the vector must be **logical** 1's and 0's, the following function that appears at first to accomplish the same operation using the programming method, actually does not. The function receives

two input arguments: the vector, and an integer with which to compare (so it is somewhat more general). It loops through every element in the input vector, and stores in the result vector either a 1 or 0 depending on whether vec(i) > n is true or false.

```
testvecgtn.m
function outvec = testvecgtn(vec,n)
% Compare each element in vec to see whether it
% is greater than n or not
% Preallocate the vector
outvec = zeros(size(vec));
for i = 1:length(vec)
        % Each element in the output vector stores 1 or 0
         if vec(i) > n
         outvec(i) = 1;
         else
         outvec(i) = 0;
         end
end
```

Calling the function appears to return the same vector as simply vec > 5, and summing the result still works to determine how many elements were greater than 5.

```
>> notlog = testvecgtn(vec,5)
notlog =
0        1        0        0        1        1
>> sum(notlog)
ans =
         3
```

However, as before, it could not be used to index into a vector because the elements are **double**, not **logical**:

```
>> vec(notlog)
??? Subscript indices must either be real positive integers or logicals.
```

While **Loops**

The **while** statement is used as the conditional loop in MATLAB; it is used to repeat an action when ahead of time it is *not* known *how many* times the action will be repeated. The general form of the **while** statement is:

```
while condition
        action
end
```

The action, which consists of any number of statement(s), is executed as long as the condition is true. The condition must eventually become false to avoid an ***infinite loop***. (If this happens, Ctrl-C will exit the loop.) The way it works is that first the condition is evaluated. If it is logically true, the action is executed. So, to begin with it is just like an **if** statement. However, at that point the condition is evaluated again. If it is still true, the action is executed again. Then, the action is evaluated again. If it is still true, the action is executed again. Then, the action is… eventually, this has to stop! Eventually something in the action has to change something in the condition so it becomes false. As an example of a conditional loop, we will write a function that will find the first factorial that is greater than the input argument *high*. Previously, we wrote a function to calculate a particular factorial. For example, to calculate 5! we found the product 1 * 2 * 3 * 4 * 5. In that case a **for** loop was used, since it was known that the loop would be repeated five times. Now, we do not know how many times the loop will be repeated. The basic algorithm is to have two variables, one that iterates through the values 1, 2, 3, and so on, and one that stores the factorial of the iterator at each step. We start with 1, and 1 factorial, which is 1. Then, we check the factorial. If it is not greater than *high*, the iterator variable will then increment to 2, and find its factorial (2). If this is not greater than *high*, the iterator will then increment to 3, and the function will find its factorial (6). This continues until we get to the first factorial that is greater than *high*. So, the process of incrementing a variable and finding its factorial is repeated until we get to the first value greater than *high*. This is implemented using a **while** loop:

```
factgthigh.m
function facgt = factgthigh(high)
% Finds the first factorial > high
i=0;
fac=1;
while fac <= high
        i=i+1;
        fac = fac * i;
end
facgt = fac;
```

Here is an example of calling the function, passing 5000 for the value of the input argument *high*.

```
>> factgthigh(5000)
ans =
5040
```

The iterator variable *i* is initialized to 0, and the running product variable *fac*, which will store the factorial of each value of *i*, is initialized to 1. The first time the **while** loop is executed, the condition is conceptually true: 1 is less than or equal to 5000. So, the action of the loop is executed, which is to increment *i* to 1 and *fac* to 1 (1 * 1). After the execution of the action of the loop, the condition is evaluated again. Since it will still be true, the action is executed: *i* is incremented to 2, and *fac* will get the value 2 (1 * 2). The value 2 is still <= 5000, so the action will be executed again: *i* will be incremented to 3, and *fac* will get the value 6 (2 * 3). This continues until the first value of *fac* is found that is greater than 5000. As soon as *fac* gets to this value, the condition will be false and the **while** loop will end. At that point the factorial is assigned to the output argument, which returns the value. The reason that *i* is initialized to 0 rather than 1 is that the first time the loop action is executed, *i* becomes 1 and *fac* becomes 1 so we have 1 and 1!, which is 1. Notice that the output of all assignment statements is suppressed in the function.

Multiple Conditions in a While Loop

In the previous section, we wrote a function *myany* that imitated the built-in **any** function by returning logical true if any value in the input vector was logical true, and logical false otherwise. The function was inefficient because it looped through all the elements in the input vector, even though once one logical true value is found it is no longer necessary to examine any other elements. A **while** loop will improve on this. Instead of looping through all the elements, what we really want to do is to loop until either a logical true value is found, or until we've gone through the entire vector. Thus, we have two parts to the condition in the **while** loop. In the following function, we initialize the output argument to logical false, and an iterator variable *i* to 1. The action of the loop is to examine an element from the input vector: if it is logical true, we change the output argument to be logical true. Also in the action the iterator variable is incremented. The action of the loop is continued as long as the index has not yet reached the end of the vector, and as long as the output argument is still logical false.

```
myanywhile.m
function logresult = myanywhile(vec)
% Simulates the built-in function any
% Uses a while loop so that the action halts
% as soon as any true value is found
logresult = logical(0);
i = 1;
while i <= length(vec) && logresult == 0
        if vec(i) = 0
        logresult = logical(1);
        end
        i = i + 1;
end
```

The output produced by this function is the same as the *myany* function, but it is more efficient because now as soon as the output argument is set to logical true, the loop ends.

Debugging Techniques

Any error in a computer program is called a ***bug***. This term is thought to date back to the 1940s, when a problem with an early computer was found to have been caused by a moth in the computer's circuitry! The process of finding errors in a program, and correcting them, is still called ***debugging***.

Types of Errors

There are several different kinds of errors that can occur in a program, which fall into the categories of ***syntax errors, run-time errors***, and ***logical errors***. Syntax errors are mistakes in using the language. Examples of syntax errors are missing a comma or a quotation mark, or misspelling a word. MATLAB itself will flag syntax errors and give an error message. For example, the following string is missing the end quote:

```
>> mystr = 'how are you;
??? mystr = 'how are you;
            |
Error: A MATLAB string constant is not terminated properly.
```

Another common mistake is to spell a variable name incorrectly, which MATLAB will also catch.

```
>> value = 5;
>> newvalue = valu + 3;
??? Undefined function or variable 'valu'.
```

Run-time, or execution-time, errors are found when a script or function is executing. With most languages, an example of a run-time error would be attempting to divide by zero. However, in MATLAB, this will generate a warning message. Another example would be attempting to refer to an element in an array that does not exist.

runtime_ex.m

```
% This script shows an execution-time error
vec = 3:5;
for i = 1:4
        disp(vec(i))
end
```

This script initializes a vector with three elements, but then attempts to refer to a fourth. Running it prints the three elements in the vector, and then an error message is generated when it attempts to refer to the fourth element. Notice that it gives an explanation of the error, and it gives the line number in the script in which the error occurred.

```
>> runtime_ex
3
4
5
??? Attempted to access vec(4); index out of bounds because
numel(vec)=3.
Error in ==> runtime_ex at 6
        disp(vec(i))
```

Logical errors are more difficult to locate, because they do not result in any error message. A logical error is a mistake in reasoning by the programmer, but it is not a mistake in the programming language. An example of a logical error would be dividing by 2.54 instead of multiplying in order to convert inches to centimeters. The results printed or returned would be incorrect, but this might not be obvious. All programs should be robust and should wherever possible anticipate potential errors, and guard against them. For example, whenever there is input into a program, the program should error-check and make sure that the input is in the correct range of values. Also, before dividing, the denominator should be checked to make sure that it is not zero. Despite the best precautions, there are bound to be errors in programs.

Tracing

Many times, when a program has loops and/or selection statements and is not running properly, it is useful in the debugging process to know exactly which statements have been executed. For example, here is a function that attempts to display In Middle Of Range if the argument passed to it is in the range from 3 to 6, and Out Of Range otherwise.

```
testifelse.m
function testifelse(x)
% This function will test the debugger
if 3 < x < 6
        disp('In middle of range')
else
        disp('Out of range')
end
```

However, it seems to print In Middle Of Range for all values of x:

```
>> testifelse(4)
In middle of range
>> testifelse(7)
```

```
In middle of range
>> testifelse(–2)
In middle of range
```

One way of following the flow of the function, or ***tracing*** it, is to use the **echo** function. The **echo** function, which is a toggle, will display every statement as it is executed as well as results from the code. For scripts, just **echo** can be typed, but for functions, the name of the function must be specified, for example, echo function name on/off

```
>> echo testifelse on
```

Editor/Debugger

MATLAB has many useful functions for debugging, and debugging can also be done through its editor, called the Editor/Debugger. Typing **help debug** at the prompt in the Command Window will show some of the debugging functions. Also, in the Help Browser, clicking the Search tab and then typing **debugging** will display basic information about the debugging processes. It can be seen in the previous example that the action of the **if** clause was executed and it printed In Middle Of Range, but just from that it cannot be determined why this happened. There are several ways to set ***breakpoints*** in a file (script or function) so that the variables or expressions can be examined. These can be done from the Editor/Debugger, or commands can be typed from the Command Window. For example, the following **dbstop** command will set a breakpoint in the fifth line of this function (which is the action of the **if** clause), which allows us to type variable names and/or expressions to examine their values at that point in the execution. The function **dbcont** can be used to continue the execution, and **dbquit** can be used to quit the debug mode. Notice that the prompt becomes K>> in debug mode.

```
>> dbstop testifelse 5
>> testifelse(–2)
5 disp('In middle of range')
K>> x
x =
       –2
K>> 3 < x
ans =
       0
K>> 3 < x < 6
ans =
       1
K>> dbcont
In middle of range
end
>>
```

By typing the expressions 3 < x and then 3 < x < 6, we can determine that the expression 3 < x will return either 0 or 1. Both 0 and 1 are less than 6, so the expression will always be true, regardless of the value of x!

Function Stubs

Another common debugging technique, which is used when there is a script main program that calls many functions, is to use ***function stubs***. A function stub is a placeholder, used so that the script will work even though that particular function hasn't been written yet. For example, a programmer might start with a script main program that consists of calls to three function that accomplish all the tasks.

```
mainmfile.m
% This program gets values for x and y, and
% calculates and prints z
[x, y] = getvals;
z = calcz(x,y);
printall(x,y,z)
```

The three functions have not yet been written, however, so function stubs are put in place so that the script can be executed and tested. The function stubs consist of the proper function headers, followed by a simulation of what the function will eventually do (e.g., it puts arbitrary values in for the output arguments).

```
getvals.m
function [x, y] = getvals
x = 33;
y = 11;
calcz.m
function z = calcz(x,y)
z = 2.2;

printall.m
function printall(x,y,z)
disp('Something')
```

Then, the functions can be written and debugged one at a time. It is much easier to write a working program using this method than to attempt to write everything at once—then, when errors occur, it is not always easy to determine where the problem is!

UNIT III **PLOTTING IN MATLAB** **10 hrs.**

Basic 2D plots – modifying line styles – markers and colors – grids – placing text on a plot – Various / Special Mat Lab 2D plot types – SEMILOGX – SEMILOGY – LOG- LOG – POLAR – COMET – Example frequency response of filter circuits.

UNIT III

Plot Functions

So far, we have used **plot** to create two-dimensional plots and **bar** to create bar charts. We have seen how to clear the Figure Window using **clf**, and how to create and number Figure Windows using **figure**. Labeling plots has been accomplished using **xlabel**, **ylabel**, **title**, and **legend**, and we also have seen how to customize the strings passed to these functions using **sprintf**. The **axis** function changes the axes from the defaults that would be taken from the data in the *x* and *y* vectors to the values specified. Finally, the **grid** and **hold** toggle functions print grids or not, or lock the current graph in the Figure Window so that the next plot will be superimposed.

Matrix of Plots

Another function that is very useful with any type of plot is **subplot**, which creates a matrix of plots in the current Figure Window. Three arguments are passed to it in the form **subplot(r,c,n)**; where *r* and *c* are the dimensions of the matrix and *n* is the number of the particular plot within this matrix. The plots are numbered rowwise starting in the upper left corner. In many cases, it is useful to create a **subplot** in a **for** loop so the loop variable can iterate through the integers 1 through *n*. When the **subplot** function is called in a loop, the first two arguments will always be the same since they give the dimensions of the matrix. The third argument will iterate through the numbers assigned to the elements of the matrix. When the **subplot** function is called, it makes that element the active plot; then, any plot function can be used complete with axis labeling, titles, and such within that element. For example, the following **subplot** shows the difference, in one Figure Window, between using 10 points and 20 points to plot **sin(x)** between 0 and 2 * . The **subplot** function creates a 1 × 2 row vector of plots in the Figure Window, so that the two plots are shown side-by-side. The loop variable *i* iterates through the values 1 and then 2. The first time through the loop, when *i* has the value 1, 10*1 or 10 points are used, and the value of the third argument to the **subplot** function is 1. The second time through the loop, 20 points are used and the third argument to **subplot** is 2. Note that **sprintf** is used to print how many points were used in the plot titles. The resulting Figure Window with both plots is shown in Figure 10.1.

subplotex.m

```
%demonstrates subplot using a for loop
for i = 1:2
x = linspace(0,2*pi,10*i);
y = sin(x);
subplot(1,2,i)
plot(x,y,‘ko’)
```

```
ylabel('sin(x)')
title(sprintf('%d Points',10*i))
end
```

Plot Types

Besides **plot** and **bar**, there are other plot types such as ***histograms, stem plots, area plots*** and ***pie charts***, as well as other functions that customize graphs. Described in this section are some of the other plotting functions. The functions **bar**, **barh**, **area**, and **stem** essentially display the same data as the **plot** function, but in different forms. The **bar** function draws a bar chart (as we have seen before), **barh** draws a horizontal bar chart, **area** draws the plot as a continuous curve and fills in under the curve that is created, and **stem** draws a stem plot. For example, the following script creates a Figure Window that uses a 2 × 2 **subplot** to demonstrate these four plot types using the same x and y points (see Figure 10.2).

```
subplottypes.m

% Subplot to show plot types

x = 1:6;
y = [33 11 5 9 22 30];
subplot(2,2,1)
bar(x,y)
title('bar')
subplot(2,2,2)
barh(x,y)
title('barh')
subplot(2,2,3)
area(x,y)
title('area')
subplot(2,2,4)
stem(x,y)
title('stem')
```

Notice that the third argument in the call to the **subplot** function is a single index into the matrix created in the Figure Window; the numbering is rowwise (in contrast to the normal columnwise unwinding that MATLAB uses for matrices).

Animation

In this section we will examine a couple of ways to ***animate*** a plot. These are visuals, so the results can't really be shown here; it is necessary to type these into MATLAB to see the results. We'll start by ***animating*** a plot of **sin(x)** with the vectors:

```
>> x = -2*pi : 1/100 : 2*pi;
>> y = sin(x);
```

This results in enough points that we'll be able to see the result using the built-in **comet** function, which shows the plot by first showing the point (x(1),y(1)), and then moving on to the point (x(2),y(2)), and so on, leaving a trail (like a comet!) of all the previous points.

```
>> comet(x,y)
```

The end result looks the same as **plot(x,y)**. Another way of animating is to use the built-in function **movie**, which displays recorded movie frames. The frames are captured in a loop using the built-in function **getframe**, and are stored in a matrix. For example, the following script again animates the **sin** function. The **axis** function is used so that MATLAB will use the same set of axes for all frames, and using the **min** and **max** functions on the data vectors x and y will allow us to see all points. It displays the movie once in the **for** loop, and then again when the **movie** function is called.

sinmovie.m

```
% Shows a movie of the sin function
Clear

x = -2*pi: 1/5 : 2*pi;
y = sin(x);
n = length(x);
for i = 1:n
plot(x(i),y(i),'r*')
axis([min(x)-1 max(x)+1 min(y)-1 max(y)+1])
M(i) = getframe;
end
movie(M)
```

The Plot Function For now, we'll start with a very simple graph of one point using the plot function. The following script, plotonepoint, plots one point. To do this, first values are given for the x and y coordinates of the point in separate variables. The point is then plotted using a red*. The plot is then customized by specifying the minimum and maximum values on first the x- and then y-axis. Labels are then put on the x-axis, the y-axis, and the graph itself using the function xlabel, ylabel, and title. All this can be done from the Command Window, but it is much easier to use a script. The following shows the contents of the script plotonepoint that accomplishes this. The x-coordinate represents the time of day (e.g., 11am) and the y-coordinate represents the temperature in degrees Fahrenheit at that time: plotonepoint.m

```
% This is a really simple plot of just one point!
% Create coordinate variables and plot a red '*'
 x = 11;
y = 48;
plot(x,y,'r*')

% Change the axes and label them
axis([9 12 35 55])
xlabel('Time')
ylabel('Temperature')
```

% Put a title on the plot title('Time and Temp')

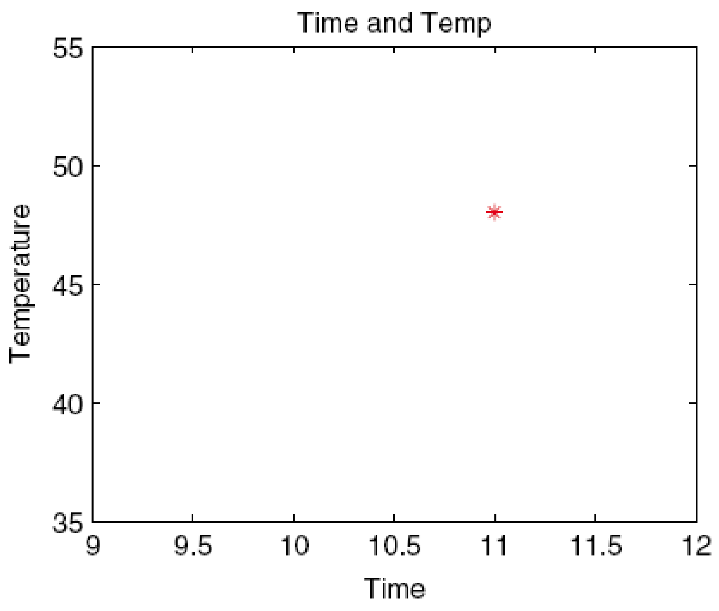
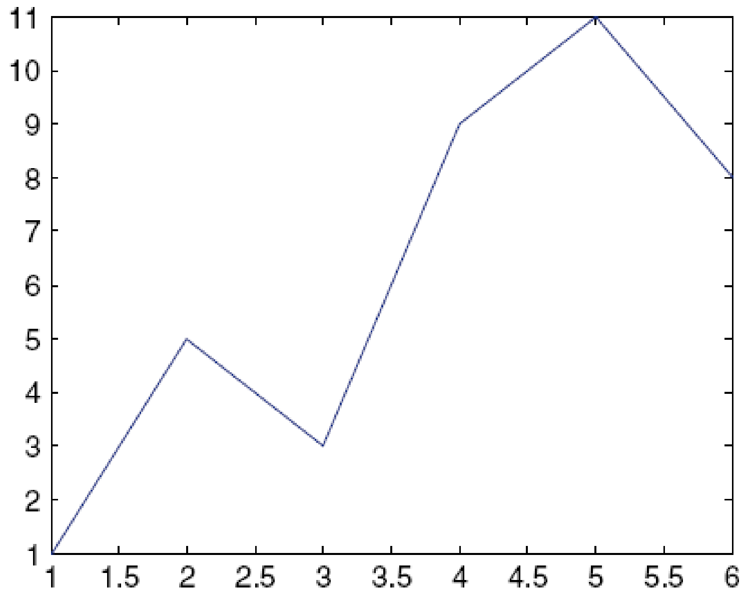


FIGURE 2.1
Plot of one data point.

FIGURE 2.2
Plot of data points from vectors.



In the call to the axis function, one vector is passed. The first two values are the minimum and maximum for the x-axis, and the last two are the minimum and maximum for the y-axis. Executing this script brings up a

Figure Window with the plot (see Figure 2.1). To be more general, the script could prompt the user for the time and temperature, rather than just assigning values. Then, the axis function could be used based on whatever the values of x and y are, for example,

```
axis([x–2 x+2 y–10 y+10])
```

In order to plot more than one point, x and y vectors are created to store the values of the (x,y) points. For example, to plot the points

(1,1)
(2,5)
(3,3)
(4,9)
(5,11)
(6,8)

first an x vector is created that has the x values (since they range from 1 to 6 in steps of 1, the colon operator can be used) and then a y vector is created with the y values. This will create (in the Command Window) x and y vectors and then plot them (see Figure 2.2).

```
>> x = 1:6;
>> y = [1 5 3 9 11 8];
 >> plot(x,y)
```

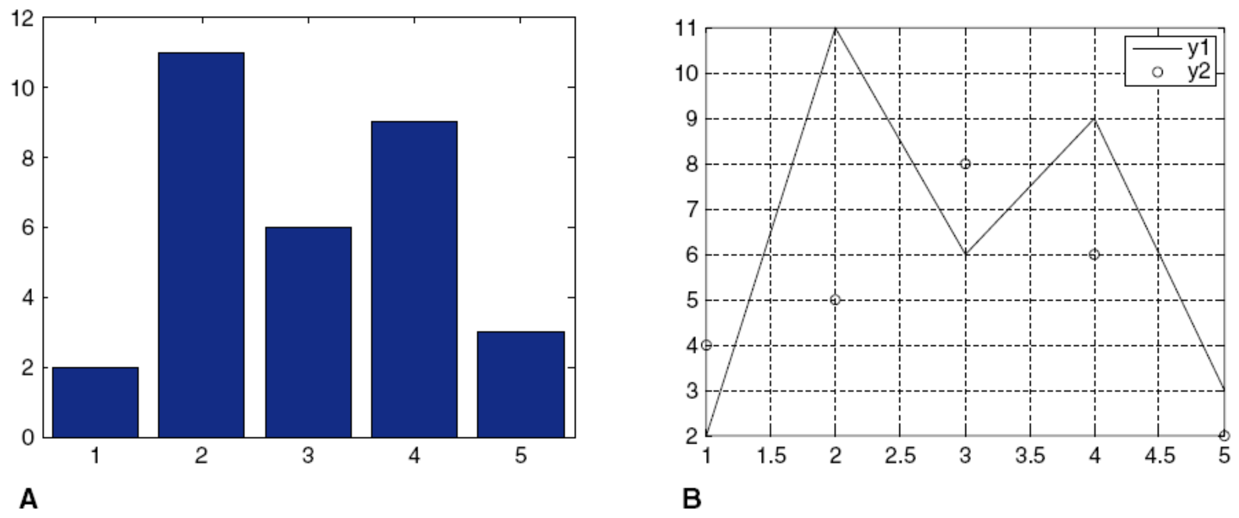


FIGURE 2.3
(A) Bar chart produced by script. (B) Plot produced by script, with a grid and legend.

Notice that the points are plotted with straight lines drawn in between. Also, the axes are set up according to the data; for example, the x values range from 1 to 6 and the y values from 1 to 11, so that is how the axes are set up. Also, notice that in this case the x values are the indices of the y vector (the y vector has six values in it, so the indices iterate from 1 to 6). When this is the case, it is not necessary to create the x vector. For example,

```
 >> plot(y)
```

will plot exactly the same figure without using an x vector.

Customizing a Plot: Color, Line Types, Marker Types

Plots can be done in the Command Window, as shown here, if they are really simple. However, many times it is desirable to customize the plot with labels, titles, and such, so it makes more sense to do this in a script. Using the help function for plot will show the many options for the line types, colors, and so on. In the script plotonepoint, earlier, the string ‘r*’ specified a red star for the point type.

The possible colors are:

b blue
c cyan
g green
k black
m magenta
r red
y yellow

The plot symbols, or markers, that can be used are:

o circle
d diamond
h hexagram
p pentagram
+ plus
. point
s square
* star
v down triangle
< left triangle
> right triangle
^ up triangle
x x-mark

Line types can also be specified by the following:

-- dashed

-. dash dot

: dotted

- solid

If no line type is specified, a solid line is drawn between the points, as seen in the last example.

Simple Related Plot Functions

Other functions that are useful in customizing plots are clf, figure, hold, legend, and grid. Brief descriptions of these functions are given here; use help to find out more about them:

clf clears the Figure Window by removing everything from it.

figure creates a new, empty Figure Window when called without any arguments. Calling it as figure(n) where n is an integer is a way of creating and maintaining multiple Figure Windows, and of referring to each individually.

hold is a toggle that freezes the current graph in the Figure Window, so that new plots will be superimposed on the current one. Just hold by itself is a toggle, so calling this function once turns the hold on, and then the next time turns it off. Alternatively, the commands hold on and hold off can be used.

legend displays strings passed to it in a legend box in the Figure Window, in order of the plots in the Figure Window.

grid displays grid lines on a graph. Called by itself, it is a toggle that turns the grid lines on and off. Alternatively, the commands grid on and grid off can be used.

Also, there are many plot types. Another simple plot type is a bar chart.

For example, the following script creates two separate Figure Windows. First, it clears the Figure Window. Then, it creates an x vector and two different y vectors (y1 and y2). In the first Figure Window, it plots the y1 values using a bar chart. In the second Figure Window, it plots the y1 values as black lines, puts hold on so that the next graph will be superimposed, and plots the y2 values as black o's. It also puts a legend on this graph and uses a grid. Labels and titles are omitted in this case since it is generic data.

plot2figs.m

```
% This creates 2 different plots, in 2 different
% Figure Windows, to demonstrate some plot features
clf x = 1:5; % Not necessary
y1 = [2 11 6 9 3];
y2 = [4 5 8 6 2];
% Put a bar chart in Figure 1

figure(1)
 bar(x,y1)
% Put plots using different y values on one plot
% with a legend
```

```
figure(2)
plot(x,y1,‘k’)
hold on
plot(x,y2,‘ko’)
grid on
legend(‘y1’,‘y2’)
```

Running this script will produce two separate Figure Windows. If there aren’t any other active Figure Windows, the first, which is the bar chart, will be in the one titled in MATLAB Figure 1. The second will be in Figure 2. See Figure 2.3 for both plots.

Notice that the first and last points are on the axes, which makes them difficult to see. That is why the axis function is frequently used—to create space around the points so that they are all visible.

The ability to pass a vector to a function and have the function evaluate every element of the vector can be very useful in creating plots. For example, the following script graphically

Âdisplays the difference between the **sin** and **cos** functions:

```
sinncos.m
% This script plots sin(x) and cos(x) in the same Figure
% Window for values of x ranging from 0 to 2*pi

clf
x = 0: 2*pi/40: 2*pi;
y = sin(x);
plot(x,y,‘ro’)
hold on
y = cos(x);
plot(x,y,‘b+’)
legend(‘sin’, ‘cos’)
title(‘sin and cos on one graph’)
```

The script creates an *x* vector; iterating through all the values from 0 to 2* in steps of 2* /40 gives enough points to get a good graph. It then finds the sine of each x value, and plots these points using red o’s. The command **hold on** freezes this in the Figure Window so the next plot will be superimposed. Next, it finds the cosine of each x value and plots these points using blue +’s. The **legend** function creates a legend; the first string is paired with the first plot, and the second string with the second plot. Running this script produces the plot seen in Figure 2.4.

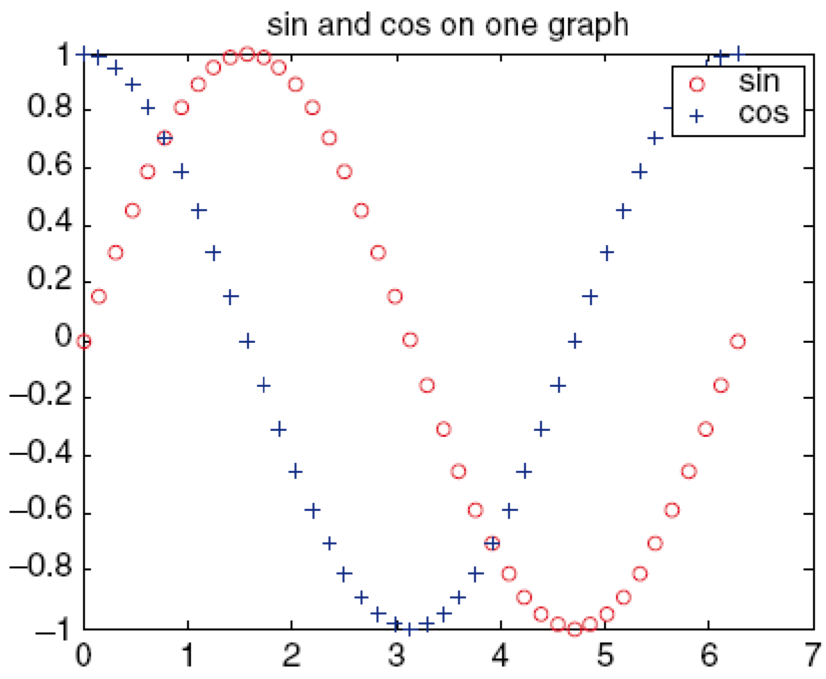


FIGURE 2.4

Plot of **sin** *and* **cos** *in one Figure Window with a legend.*

Text

- text(x,y,str)
- text(x,y,z,str)
- text(___,Name,Value)
- t = text(___)

Description

text(x,y,str) adds a text description to one or more data points in the current axes using the text specified by str. To add text to one point, specify x and y as scalars in data units. To add text to multiple points, specify x and y as vectors with equal length.

text(x,y,z,str) positions the text in 3-D coordinates.

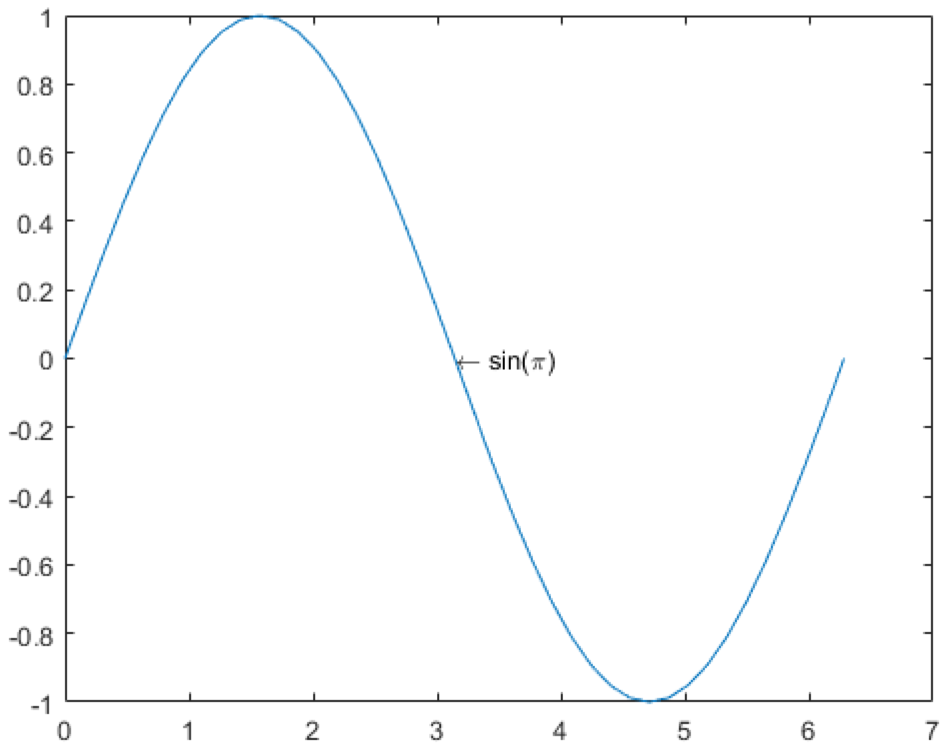
text(___,Name,Value) specifies text object properties using one or more name-value pairs. For example, 'FontSize',14 sets the font size to 14 points. You can specify text properties with any of the input argument combinations in the previous syntaxes. If you specify the Position and String properties as name-value pairs, then you do not need to specify the x, y, z, and str inputs.

t = text(___) returns one or more text objects. Use t to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text Properties. You can specify an output with any of the previous syntaxes.

Add Text Description to Data Point

Plot a sine curve. At the point $(\pi, 0)$, add the text description $\sin(\pi)$. Use the TeX markup \pi for the Greek letter π. Use \leftarrow to display a left-pointing arrow.

```
x = 0:pi/20:2*pi;
y = sin(x);
plot(x,y)
text(pi,0,'\leftarrow sin(\pi)')
```



For a list of Greek characters and other TeX markup, see the Interpreter property description.

gtext

Add text to figure using mouse

- gtext(str)
- gtext(str,Name,Value)
- t = gtext(___)

Description

gtext(str) inserts the text, str, at the location you select with the mouse. When you hover over the figure window, the pointer becomes a crosshair. gtext is waiting for you to select a location. Move the pointer to the location you want and either click the figure or press any key, except **Enter**.

gtext(str,Name,Value) specifies text properties using one or more name-value pair arguments. For example, 'FontSize',14 specifies a 14-point font.

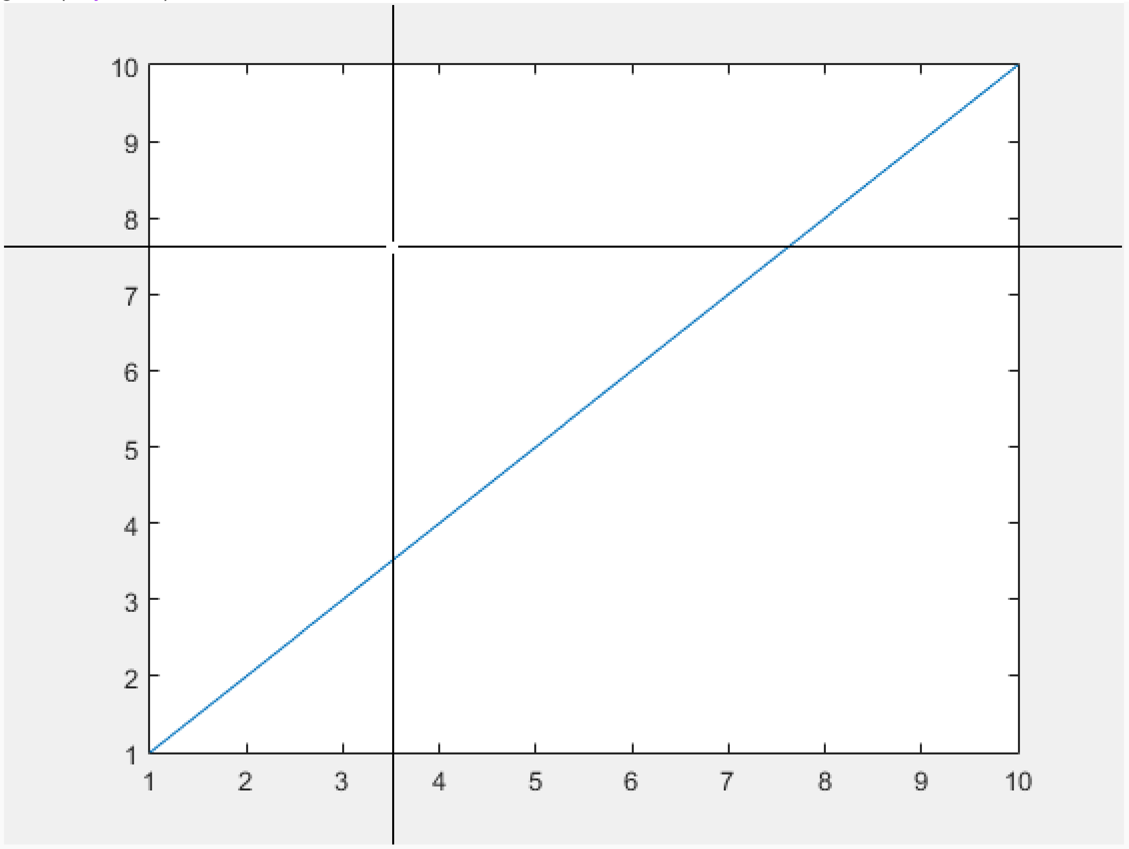
t = gtext(___) returns an array of text objects created by gtext. Use t to modify properties of the text objects after they are created. For a list of properties and descriptions, see Text Properties. You can return an output argument using any of the arguments from the previous syntaxes.

Example

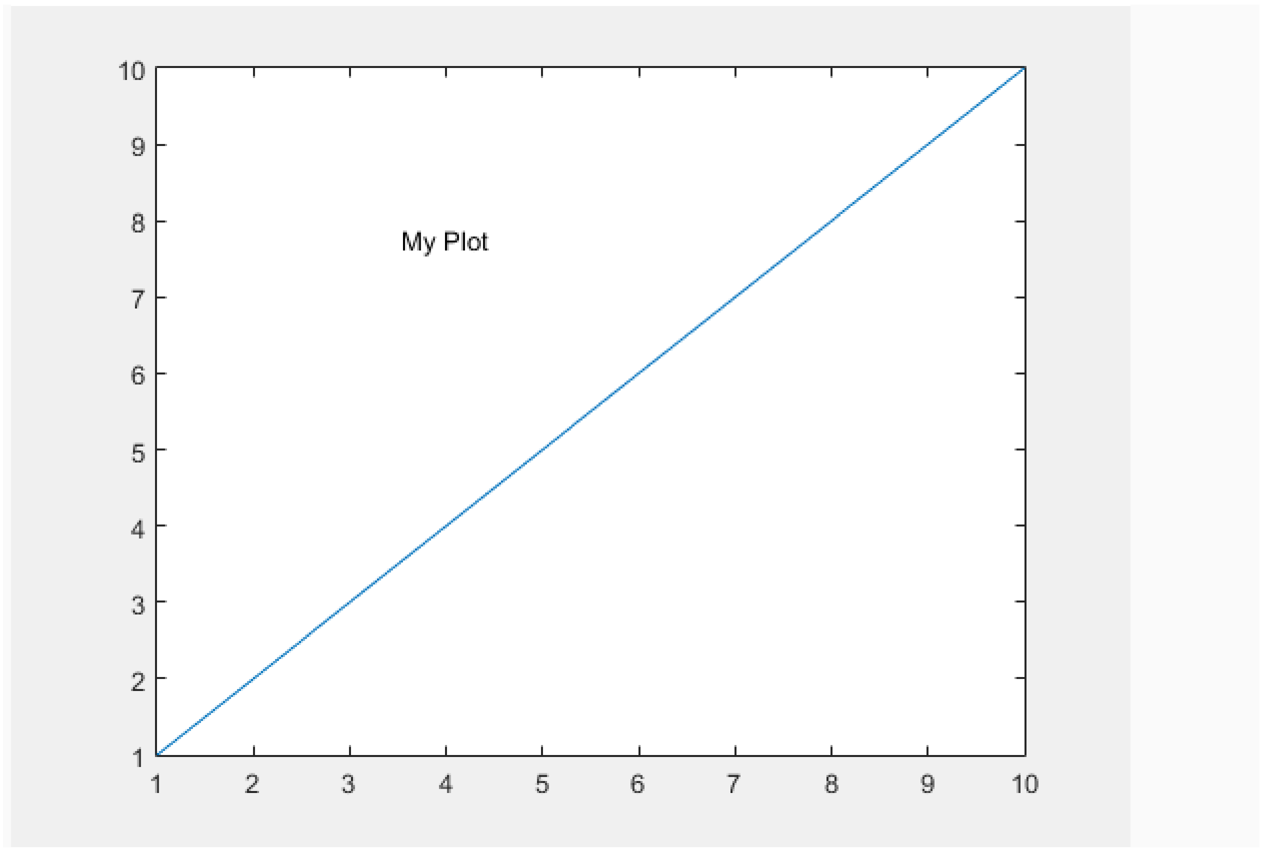
Add Text to Figure Using Mouse

Create a simple line plot and use gtext to add text to the figure using the mouse.

```
plot(1:10)
gtext('My Plot')
```



Click the figure to place the text at the selected location.



SEMILOG X

Syntax

```
semilogx(Y)
semilogx(X1,Y1,...)
semilogx(X1,Y1,LineSpec,...)
semilogx(...,'PropertyName',PropertyValue,...)
h = semilogx(...)
```

Description

semilogx plot data as logarithmic scales for the *x*-axis.

semilogx(Y) creates a plot using a base 10 logarithmic scale for the *x*-axis and a linear scale for the *y*-axis. It plots the columns of Y versus their index if Y contains real numbers.semilogx(Y) is equivalent to semilogx(real(Y),imag(Y)) if Y contains complex numbers. semilogx ignores the imaginary component in all other uses of this function.

semilogx(X1,Y1,...) plots all Yn versus Xn pairs. If only one of Xn or Yn is a matrix, semilogx plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

semilogx(X1,Y1,*LineSpec*,...) plots all lines defined by the Xn,Yn,*LineSpec* triples. LineSpec determines line style, marker symbol, and color of the plotted lines.

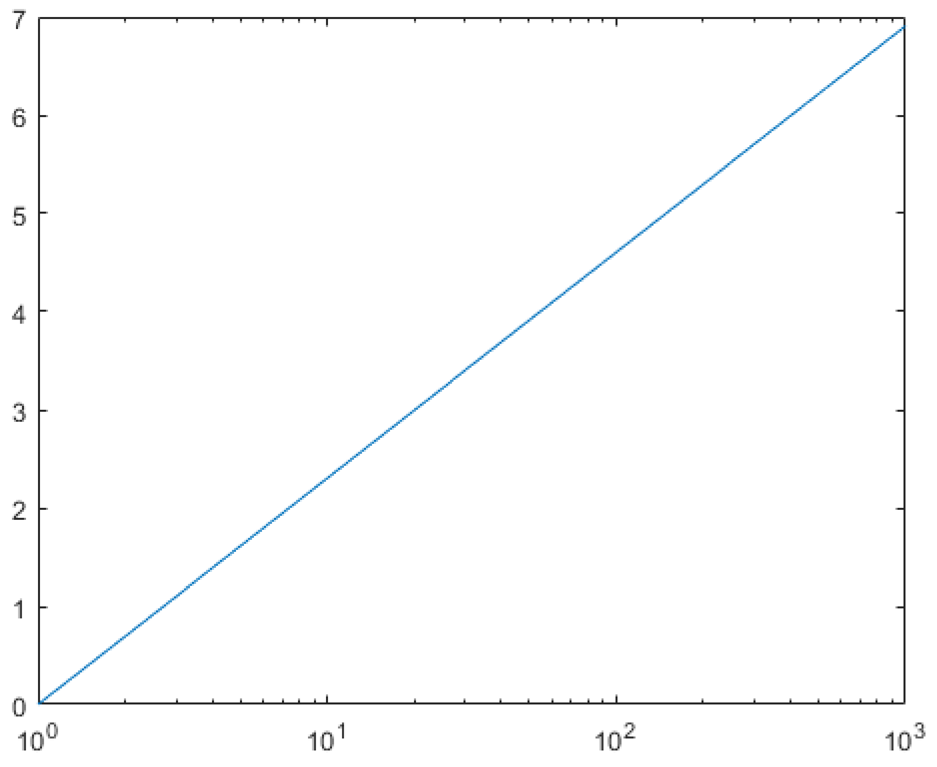
semilogx(...,'*PropertyName*',PropertyValue,...) sets property values for all charting lines created by semilogx. For a list of properties, see Chart Line Properties.

h = semilogx(...) return a vector of chart line handles, one handle per line.

Examples

Logarithmic Scale for x-Axis
Create a plot with a logarithmic scale for the x-axis and a linear scale for the y-axis.

```
x = 0:1000;
y = log(x);

figure
semilogx(x,y)
```



SEMILOGY

syntax

```
semilogy(Y)
semilogy(X1,Y1,...)
semilogy(X1,Y1,LineSpec,...)
semilogy(...,'PropertyName',PropertyValue,...)
h = semilogy(...)
```

Description

semilogy plots data with logarithmic scale for the *y*-axis.

semilogy(Y) creates a plot using a base 10 logarithmic scale for the *y*-axis and a linear scale for the *x*-axis. It plots the columns of Y versus their index if Y contains real numbers.semilogy(Y) is equivalent to semilogy(real(Y),imag(Y)) if Y contains complex numbers. semilogy ignores the imaginary component in all other uses of this function.

semilogy(X1,Y1,...) plots all Yn versus Xn pairs. If only one of Xn or Yn is a matrix, semilogy plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

semilogy(X1,Y1,*LineSpec*,...) plots all lines defined by the Xn,Yn,*LineSpec* triples. LineSpec determines line style, marker symbol, and color of the plotted lines.

semilogy(...,'*PropertyName*',PropertyValue,...) sets property values for all the charting lines created by semilogy. For a list of properties, see Chart Line Properties.

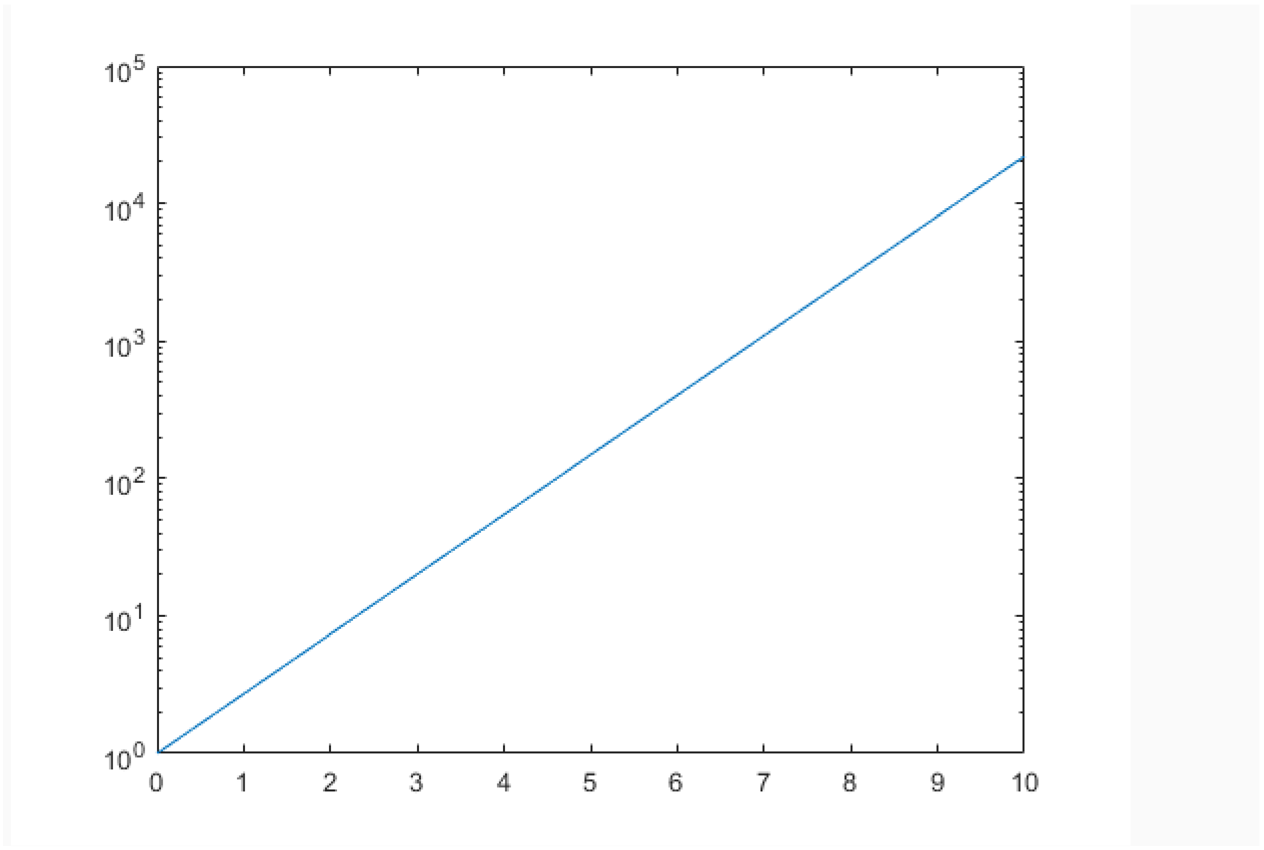
h = semilogy(...) returns a vector of chart line handles, one handle per line.

Examples

Logarithmic Scale for y-Axis

Create a plot with a logarithmic scale for the y-axis and a linear scale for the x-axis.

```
x = 0:0.1:10;
y = exp(x);

figure
semilogy(x,y)
```



LOG LOG

Syntax

loglog(Y)
loglog(X1,Y1,...)
loglog(X1,Y1,*LineSpec*,...)
loglog(...,'*PropertyName*',PropertyValue,...)
h = loglog(...)

Description

loglog(Y) plots the columns of Y versus their index if Y contains real numbers. If Y contains complex numbers, loglog(Y) and loglog(real(Y),imag(Y)) are equivalent. loglog ignores the imaginary component in all other uses of this function.

loglog(X1,Y1,...) plots all Yn versus Xn pairs. If only one of Xn or Yn is a matrix, loglog plots the vector argument versus the rows or columns of the matrix, along the dimension of the matrix whose length matches the length of the vector. If the matrix is square, its columns plot against the vector if their lengths match.

loglog(X1,Y1,*LineSpec*,...) plots all lines defined by the Xn,Yn,*LineSpec* triples, where LineSpec determines line type, marker symbol, and color of the plotted lines. You can mixXn,Yn,*LineSpec* triples with Xn,Yn pairs, for example, loglog(X1,Y1,X2,Y2,*LineSpec*,X3,Y3)

loglog(...,'*PropertyName*',PropertyValue,...) sets line property values for all the charting lines created. h = loglog(...) returns a column vector of chart line handles, one handle per line.

If you do not specify a color when plotting more than one line, loglog automatically cycles through the colors and line styles in the order specified by the current axes.

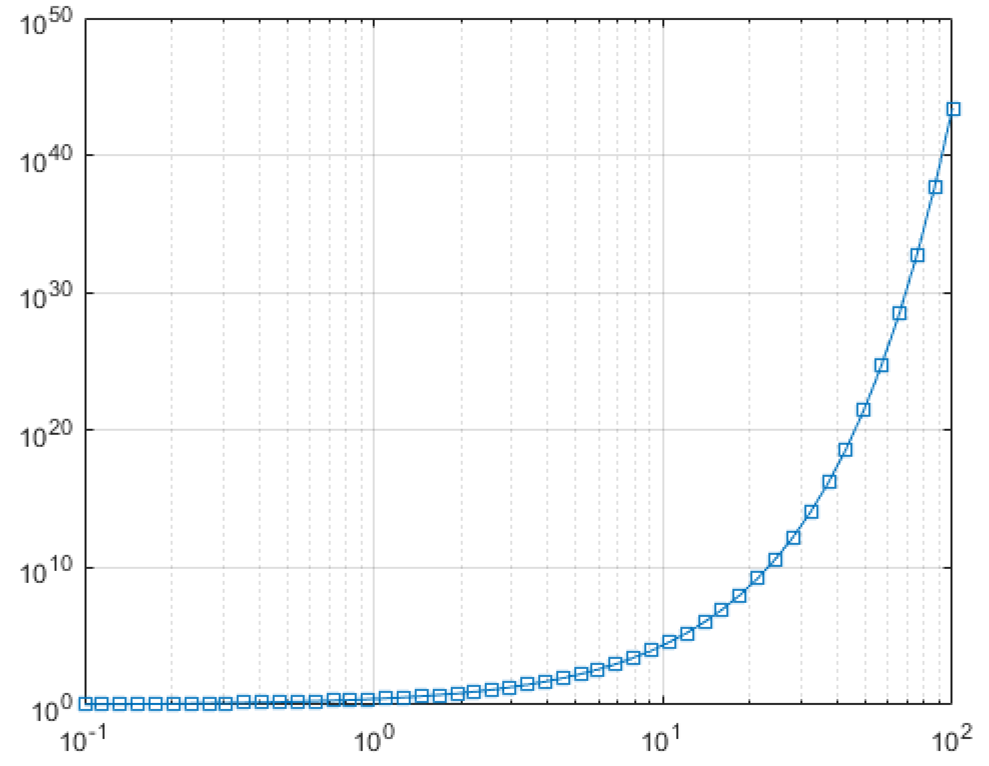
If you attempt to add a loglog, semilogx, or semilogy plot to a linear axis mode graph with hold on, the axis mode remains as it is and the new data plots as linear.

Examples

Logarithmic Scale for Both Axes

Create a plot using a logarithmic scale for both the x-axis and the y-axis. Set the LineSpec string so that loglog plots using a line with square markers. Display the grid.

```
x = logspace(-1,2);
y = exp(x);

figure
loglog(x,y,'-s')
grid on
```



POLAR

Syntax

```
polar(theta,rho)
polar(theta,rho,LineSpec)
polar(axes_handle,...)
h = polar(...)
```

DESCRIPTION

The polar function accepts polar coordinates, plots them in a Cartesian plane, and draws the polar grid on the plane.

polar(theta,rho) creates a polar coordinate plot of the angle theta versus the radius rho. theta is the angle from the *x*-axis to the radius vector specified in radians; rho is the length of the radius vector specified in dataspace units.

polar(theta,rho,LineSpec) LineSpec specifies the line type, plot symbol, and color for the lines drawn in the polar plot.

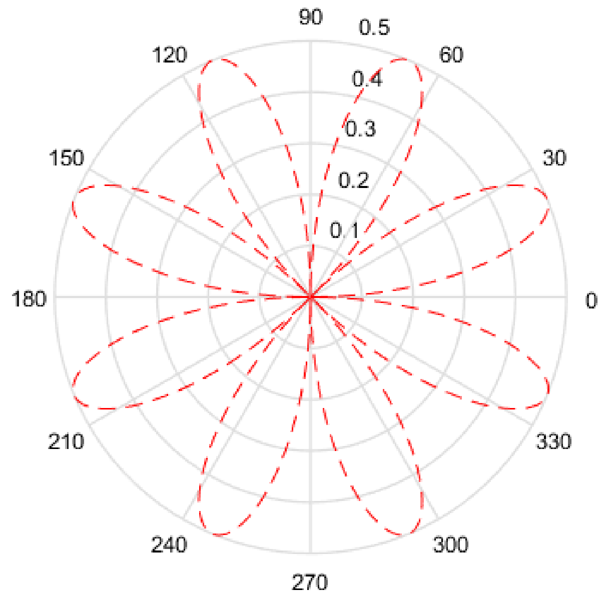
polar(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes (gca).

h = polar(...) returns the handle of a line object in h.

EXAMPLES

Simple Polar Plot
Create a simple polar plot using a dashed red line.

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);

figure
polar(theta,rho,'--r')
```



COMET

SYNTAX

comet(y)
comet(x,y)
comet(x,y,p)
comet(axes_handle,...)

DESCRIPTION

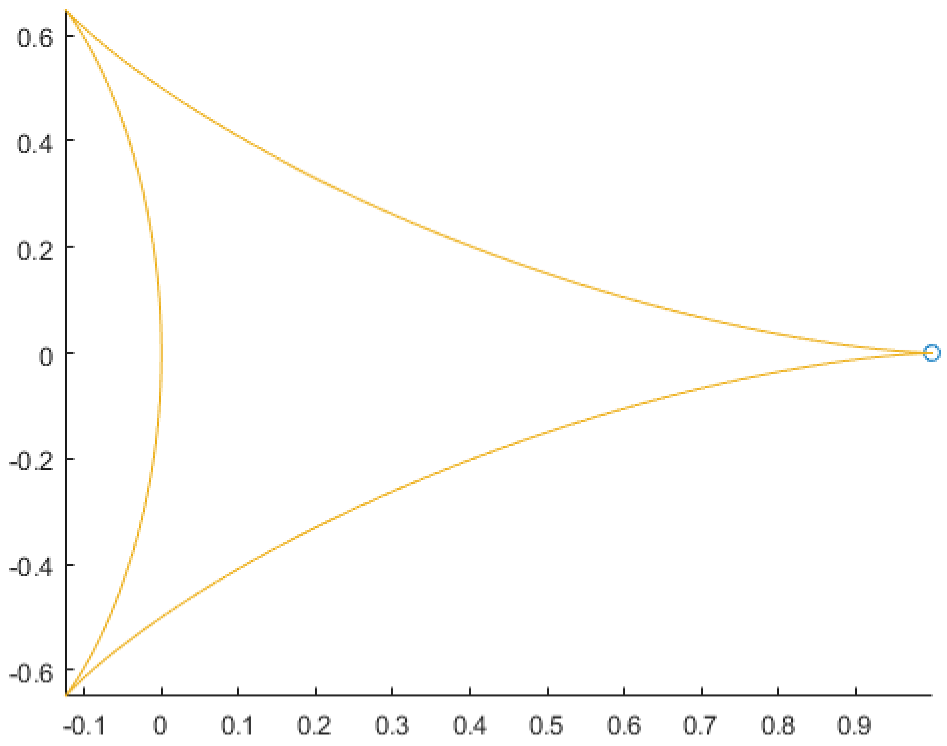
comet(y) displays a comet graph of the vector y. A comet graph is an animated graph in which a circle (the comet *head*) traces the data points on the screen. The comet *body* is a trailing segment that follows the head. The *tail* is a solid line that traces the entire function.

comet(x,y) displays a comet graph of vector y versus vector x.

comet(x,y,p) specifies a comet body of length p*length(y). p defaults to 0.1.

comet(axes_handle,...) plots into the axes with the handle axes_handle instead of into the current axes

CREATE COMET GRAPH
```
t = 0:.01:2*pi;
x = cos(2*t).*(cos(t).^2);
y = sin(2*t).*(sin(t).^2);
comet(x,y);
```



Derive and Plot a Low Pass Transfer Function on MATLAB

Introduction to Filters

A filter is a circuit that removes unwanted frequencies from a waveform. Filters can be used to remove noise from a system to make it cleaner. It consists of two main bands: the pass band and the stop band.

To understand the pass band and stop band in a filter, we need to understand Bode plots. A Bode plot is a graph that tracks the response of frequencies. It shows the magnitude of a signal with respect to the frequency. The magnitude or the amplitude is measured in decibels and plotted on the Y-axis of the Bode plot. The X-axis of the bode plot is the frequency of the filter.

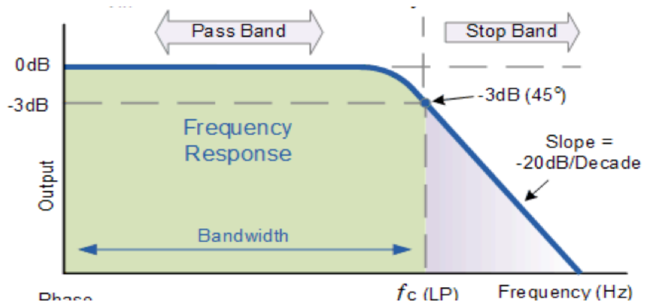


Figure 1. Example of a Low Pass Bode Plot

The above image is a bode plot for a low pass filter. The frequencies in the pass band are the frequencies with an amplitude of 0 decibels or above. The frequencies after the cutoff frequencies f_c are in the stop band. The frequencies that we want to remove would be in the stop band when the magnitude is less than zero.

Depending on which frequencies we want to remove, the location of the pass band will vary to create the main 4 filters types. The main four filter response types are:

- High pass filters
- Low pass filters
- Band pass filters
- Band stop filters

The order of a filter indicates how steep the slope is. For every raise in order of a filter, there is a 6db/octave increase in the filter's slope. An ideal perfect filter would have a slope of infinity. It would look like a square wave. Unfortunately, these ideal filters cannot be made in real life, and we can only make filters that have a roll-off or slope as close to this as we can.

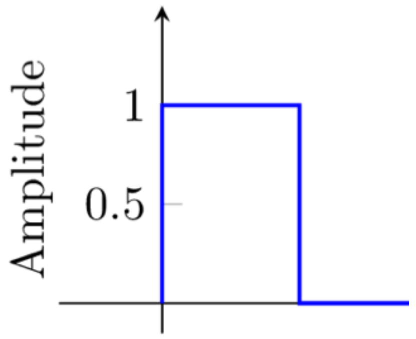


Figure 2. Frequency response of an Ideal Filter

Deriving a Low Pass Transfer Function
The transfer function for a low pass Akerberg-Mossberg filter is seen below in equation 2.

$$T_{\mathrm{LP}} = \frac{\omega_0^2}{s^2 + \frac{\omega_0}{Q}s + \omega_0^2}$$

Now we have to find the correct values for a,b,c, and d in equation 1 to end up with the transfer function of the low pass filter. We see that the numerator that we want only has w_0^2 so we have to get rid of the other monomials to only leave w_0^2 by itself. The first variable we want to get rid of is the first term, as^2. So we set a=0 in this case. The second terms in the transfer function of equation 1 has the variables "a" and "b". We want to equal it to zero since we only want the w_0^2 to be left in the numerator.

$$s(\omega_0/Q)\,[a - b(kQ)] = 0$$

In this case, our "a" had already been set to 0 by the first term and to make this one zero we also have to set “b” to 0. Doing this now gets rid of this second term by making it equal to zero.

Now to make the last term equal to w_0^2 we need to find our remaining values, “d” and “k”. Notice we had not found k previously because it was being multiplied by “b” which was zero.

$$\omega_0^2\,[a - (c-d)k] = \omega_0^2$$

Our “a” and “c” have already been set, so we are missing “d” and “k”. To make this configuration possible, we would have to make “d” and “k” both equal to 1. Once those variables are inserted, in the numerator we have the remaining transfer function.

Now that we have all of our values, we can insert them into MATLAB to plot the frequency response for this filter.

Graphing in MATLAB

To start off, we will do a new script in MATLAB. Since the frequency response or Bode plot is logarithmic, the first thing we will declare is a logarithmic spaced vector. We will use: w = logspace(0,9,200);

```
% THE FIRST TWO POINTS ARE THE BOUNDARIES OF THE GRAPH. THE 200 IS THE NUMBER OF POINTS THAT WILL
BE GENERATED

s=j.*w;

a=0;

b=0;

c=0;

d=1;

k=1;

Q=1;

w0=1000; % Chosen Cutoff Frequency

tn = -((a*(s.^2))+(s.*(w0/Q))*(a-(b*k*Q))+(w0^2*(a-(c-d)*k))); %numerator of transfer function

td = (s.^2)+(s.*(w0/Q))+(w0^2); % the denominator of the transfer function

t1 = tn./td; %numerator over the denominator

plot(log10(w), 20*log10(abs(t1)));grid on;title('Lowpass') % matlab will now plot our transfer function with
respect to the graph we declared
```