Moi University

COM 325

Computer Applications II

Mr. Brian
mwotie@gmail.com

CHAPTER 1

INTRODUCTION

MATLAB (matrix laboratory) is a fourth-generation high-level programming language and interactive environment for numerical computation, visualization and programming. MATLAB is developed by MathWorks.

It allows matrix manipulations; plotting of functions and data; implementation of algorithms; creation of user interfaces; interfacing with programs written in other languages, including C, C++, Java, and FORTRAN; analyze data; develop algorithms; and create models and applications.

It has numerous built-in commands and math functions that help you in mathematical calculations, generating plots, and performing numerical methods.

MATLAB's Power of Computational Mathematics

MATLAB is used in every facet of computational mathematics. Following are some commonly used mathematical calculations where it is used most commonly:

- Dealing with Matrices and Arrays
- 2-D and 3-D Plotting and graphics
- Linear Algebra
- Algebraic Equations
- Non-linear Functions
- Statistics
- Data Analysis
- Calculus and Differential Equations
- Numerical Calculations
- Integration
- Transforms
- Curve Fitting
- Various other special functions

Features of MATLAB

Following are the basic features of MATLAB:

- It is a high-level language for numerical computation, visualization and application development.
- It also provides an interactive environment for iterative exploration, design and problem solving.
- It provides vast library of mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration and solving ordinary differential equations.
- It provides built-in graphics for visualizing data and tools for creating custom plots.

- MATLAB's programming interface gives development tools for improving code quality, maintainability, and maximizing performance.
- It provides tools for building applications with custom graphical interfaces.
- It provides functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET and Microsoft Excel.

Uses of MATLAB

MATLAB is widely used as a computational tool in science and engineering encompassing the fields of physics, chemistry, math and all engineering streams. It is used in a range of applications including:

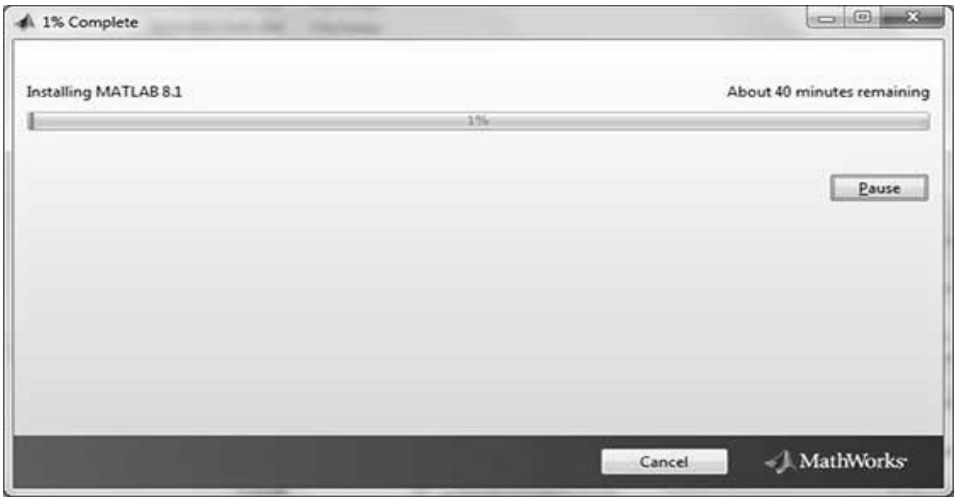
- Signal processing and Communications
- Image and video Processing
- Control systems
- Test and measurement
- Computational finance
- Computational biology

Local Environment Setup

Setting up MATLAB environment is a matter of few clicks. The installer can be downloaded from http://in.mathworks.com/downloads/web_downloads:

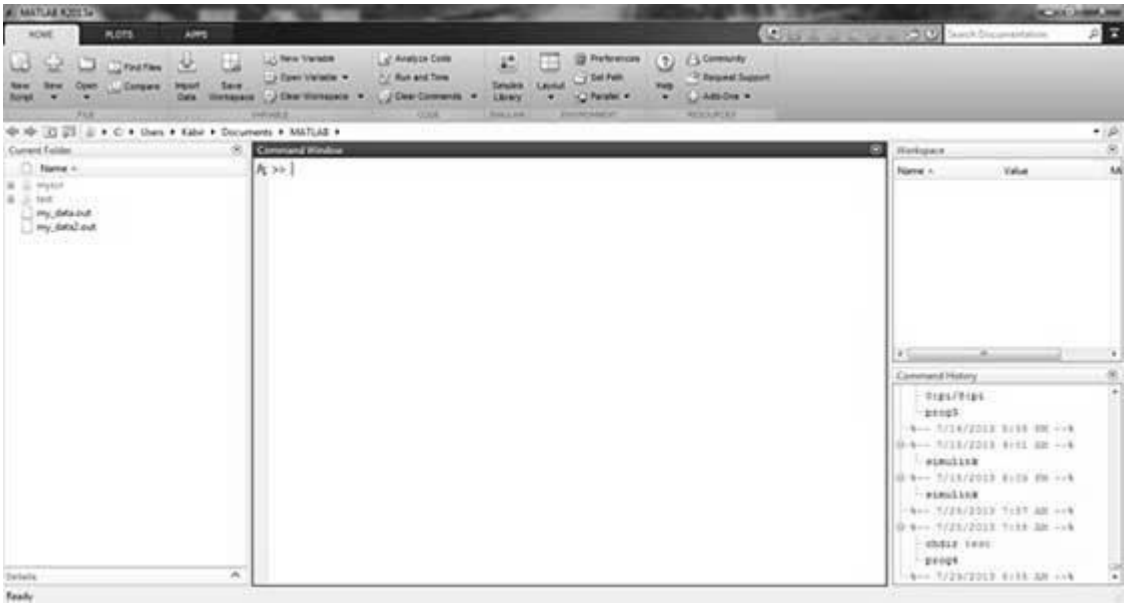
MathWorks provides the licensed product, a trial version and a student version as well. You need to log into the site and wait a little for their approval.

After downloading the installer the software can be installed through few clicks.



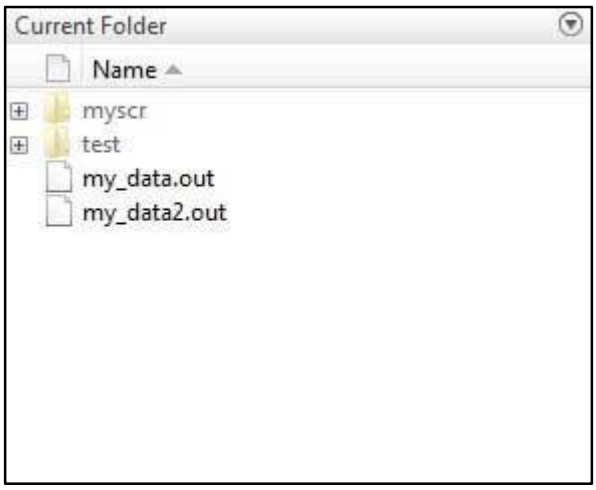
Understanding the MATLAB Environment

MATLAB development IDE can be launched from the icon created on the desktop. The main working window in MATLAB is called the desktop. When MATLAB is started, the desktop appears in its default layout:

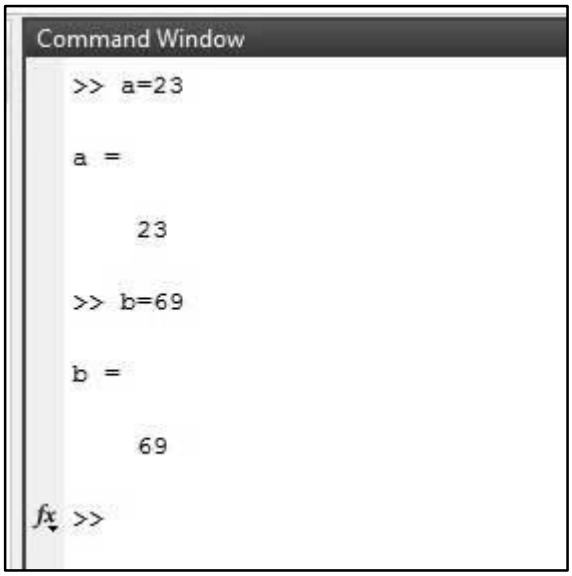


The desktop has the following panels:

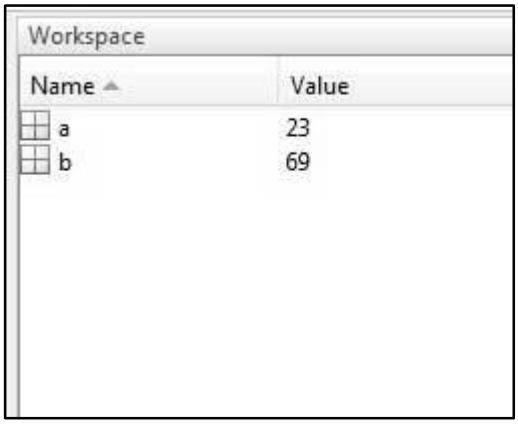
Current Folder - This panel allows you to access the project folders and files.



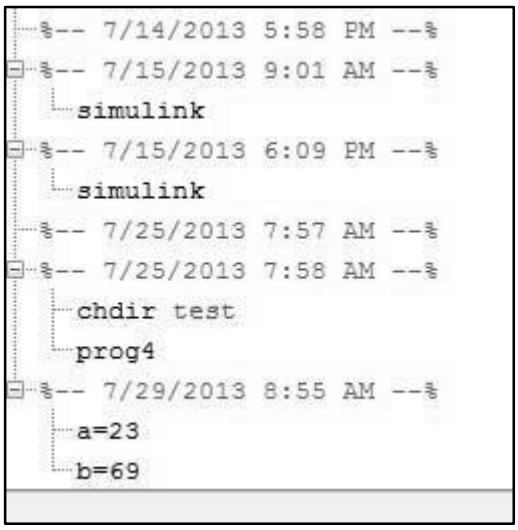
Command Window - This is the main area where commands can be entered at the command line. It is indicated by the command prompt (>>).



Workspace - The workspace shows all the variables created and/or imported from files.



Command History - This panel shows or rerun commands that are entered at the command line.


%-- 7/14/2013 5:58 PM --%
%-- 7/15/2013 9:01 AM --%
simulink
%-- 7/15/2013 6:09 PM --%
simulink
%-- 7/25/2013 7:57 AM --%
%-- 7/25/2013 7:58 AM --%
chdir test
prog4
%-- 7/29/2013 8:55 AM --%
a=23
b=69

CHAPTER 2

BASIC SYNTAX

MATLAB environment behaves like a super-complex calculator. You can enter commands at the >> command prompt.

MATLAB is an interpreted environment. In other words, you give a command and MATLAB executes it right away.

Hands on Practice

Type a valid expression, for example,

5 + 5

And press ENTER

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

ans = 10

Let us take up few more examples:

```
3 ^ 2        % 3 raised to the power of 2
```

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

ans = 9

Another example,

```
sin(pi /2)   % sine of angle 90o
```

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

ans = 1

Another example,

```
7/0                  % Divide by zero
```

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

ans = Inf warning: division by zero

Another example,

732 * 20.3

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

ans = 1.4860e+04

MATLAB provides some special expressions for some mathematical symbols, like pi for π, Inf for ∞, i (and j) for √-1 etc. **Nan** stands for 'not a number'.

Use of Semicolon (;) in MATLAB

Semicolon (;) indicates end of statement. However, if you want to suppress and hide the MATLAB output for an expression, add a semicolon after the expression.

For example,

```
x = 3; y = x + 5
```

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is:

```
y = 8
```

Adding Comments

The percent symbol (%) is used for indicating a comment line. For example,

```
x = 9   % assign the value 9 to x
```

You can also write a block of comments using the block comment operators % { and % }.

The MATLAB editor includes tools and context menu items to help you add, remove, or change the format of comments.

Commonly used Operators and Special Characters

MATLAB supports the following commonly used operators and special characters:

Operator	Purpose
+	Plus; addition operator.
-	Minus; subtraction operator.
*	Scalar and matrix multiplication operator.
.*	Array multiplication operator.
^	Scalar and matrix exponentiation operator.
.^	Array exponentiation operator.
\	Left-division operator.
/	Right-division operator.
.\	Array left-division operator.
./	Array right-division operator.

:	Colon; generates regularly spaced elements and represents an entire row or column.
( )	Parentheses; encloses function arguments and array indices; overrides precedence.
[ ]	Brackets; enclosures array elements.
.	Decimal point.
…	Ellipsis; line-continuation operator
,	Comma; separates statements and elements in a row
;	Semicolon; separates columns and suppresses display.
%	Percent sign; designates a comment and specifies formatting.
_	Quote sign and transpose operator.
._	Non-conjugated transpose operator.
=	Assignment operator.

CHAPTER 3

VARIABLES

Special Variables and Constants

MATLAB supports the following special variables and constants:

Name	Meaning
ans	Most recent answer.
eps	Accuracy of floating-point precision.
i,j	The imaginary unit √-1.
Inf	Infinity.
NaN	Undefined numerical result (not a number).
pi	The number π

Naming Variables

Variable names consist of a letter followed by any number of letters, digits or underscore.

MATLAB is case-sensitive.

Variable names can be of any length, however, MATLAB uses only first N characters, where N is given by the function **namelengthmax**.

Saving Your Work

The **save** command is used for saving all the variables in the workspace, as a file with .mat extension, in the current directory.

For example,

```
save myfile
```

You can reload the file anytime later using the **load** command.

```
load myfile
```

In MATLAB environment, every variable is an array or matrix.

You can assign variables in a simple way. For example,

```
x = 3     % defining x and initializing it with a value
```

MATLAB will execute the above statement and return the following result:

```
x = 3
```

It creates a 1-by-1 matrix named *x* and stores the value 3 in its element. Let us check another example,

```
x = sqrt(16)                % defining x and initializing it with an expression
```

MATLAB will execute the above statement and return the following result:

x = 4

Please note that:

Once a variable is entered into the system, you can refer to it later.

Variables must have values before they are used.

When an expression returns a result that is not assigned to any variable, the system assigns it to a variable named ans, which can be used later.

For example,

sqrt(78)

MATLAB will execute the above statement and return the following result:

ans =

8.8318

You can use this variable ***ans***:

9876/ans

MATLAB will execute the above statement and return the following result:

ans =

1.1182e+03

Let's look at another example:

```
x      = 7 * 8; y = x * 7.89
```

MATLAB will execute the above statement and return the following result:

```
y      =
```

441.8400

Multiple Assignments

You can have multiple assignments on the same line. For example,

a = 2; b = 7; c = a * b;

MATLAB will execute the above statement and return the following result:

c =

14

I have forgotten the Variables!

The **who** command displays all the variable names you have used.

who

MATLAB will execute the above statement and return the following result:

Your variables are: a ans b c x y

The **whos** command displays little more about the variables:

Variables currently in memory

Type of each variables

Memory allocated to each variable

Whether they are complex variables or not

whos

MATLAB will execute the above statement and return the following result:

Name	Size	Bytes	Class Attributes
a	1x1	8	double
ans	1x1	8	double
b	1x1	8	double
c	1x1	8	double
x	1x1	8	double
y	1x1	8	double

The **clear** command deletes all (or the specified) variable(s) from the memory.

```
clear x     % it will delete x,
```

won't display anything clear   % it will delete all variables in the workspace

% peacefully and unobtrusively

Long Assignments

Long assignments can be extended to another line by using an ellipses (...). For example,

```
initial_velocity = 0; acceleration = 9.8; time = 20; final_velocity = initial_velocity ...
   + acceleration * time
```

MATLAB will execute the above statement and return the following result:

```
final_velocity =    196
```

The format Command

By default, MATLAB displays numbers with four decimal place values. This is known as ***short format***.

However, if you want more precision, you need to use the **format** command.

The **format long** command displays 16 digits after decimal.

For example:

```
format long x = 7 + 10/3 + 5 ^ 1.2
```

MATLAB will execute the above statement and return the following result:

```
x =
  17.231981640639408
```

Another example,

```
format short x = 7 + 10/3 + 5 ^ 1.2
```

MATLAB will execute the above statement and return the following result:

```
x =
```

17.2320

The **format bank** command rounds numbers to two decimal places. For example,

```
format bank daily_wage = 177.45; weekly_wage = daily_wage * 6
```

MATLAB will execute the above statement and return the following result:

```
weekly_wage =
      1064.70
```

MATLAB displays large numbers using exponential notation.

The **format short e** command allows displaying in exponential form with four decimal places plus the exponent.

For example,

```
format short e
4.678 * 4.9
```

MATLAB will execute the above statement and return the following result:

```
ans =
  2.2922e+01
```

The **format long e** command allows displaying in exponential form with four decimal places plus the exponent. For example,

```
format long e x = pi
```

MATLAB will execute the above statement and return the following result:

```
x =
   3.141592653589793e+00
```

The **format rat** command gives the closest rational expression resulting from a calculation. For example,

```
format rat
16
4.678 * 4.9
```

MATLAB will execute the above statement and return the following result:

```
ans =
  2063/90
```

Creating Vectors

A vector is a one-dimensional array of numbers. MATLAB allows creating two types of vectors:

- Row vectors
- Column vectors

Row vectors are created by enclosing the set of elements in square brackets, using space or comma to delimit the elements.

For example,

```
r = [7 8 9 10 11]
```

MATLAB will execute the above statement and return the following result:

```
r =
  Columns 1 through 4
     7     8     9    10
  Column 5
    11
```

Another example,

```
r = [7 8 9 10 11]; t = [2, 3, 4, 5, 6]; res = r + t
```

MATLAB will execute the above statement and return the following result:

```
res =
  Columns 1 through 4
     9    11    13    15
  Column 5
    17
```

Column vectors are created by enclosing the set of elements in square brackets, using semicolon (;) to delimit the elements.

```
c = [7;  8;  9;  10; 11]
```

MATLAB will execute the above statement and return the following result:

```
c =
     7
     8
     9
    10
    11
```

Creating Matrices

A matrix is a two-dimensional array of numbers.

In MATLAB, a matrix is created by entering each row as a sequence of space or comma separated elements, and end of a row is demarcated by a semicolon. For example, let us create a 3-by-3 matrix as:

```
m = [1 2 3; 4 5 6; 7 8 9]
```

MATLAB will execute the above statement and return the following result:

```
m =
     1     2     3
     4     5     6
```

7 8 9

CHAPTER 4

COMMANDS

MATLAB is an interactive program for numerical computation and data visualization. You can enter a command by typing it at the MATLAB prompt '>>' on the **Command Window**.

In this section, we will provide lists of commonly used general MATLAB commands.

Commands for Managing a Session

MATLAB provides various commands for managing a session. The following table provides all such commands:

Command	Purpose
clc	Clears command window.
clear	Removes variables from memory.
exist	Checks for existence of file or variable.
global	Declares variables to be global.
help	Searches for a help topic.
lookfor	Searches help entries for a keyword.
quit	Stops MATLAB.
who	Lists current variables.
whos	Lists current variables (long display).

Commands for Working with the System

MATLAB provides various useful commands for working with the system, like saving the current work in the workspace as a file and loading the file later.

It also provides various commands for other system-related activities like, displaying date, listing files in the directory, displaying current directory, etc.

The following table displays some commonly used system-related commands:

Command	Purpose
cd	Changes current directory.

date	Displays current date.
delete	Deletes a file.
diary	Switches on/off diary file recording.
dir	Lists all files in current directory.
load	Loads workspace variables from a file.
path	Displays search path.
pwd	Displays current directory.
save	Saves workspace variables in a file.
type	Displays contents of a file.
what	Lists all MATLAB files in the current directory.
wklread	Reads .wk1 spreadsheet file.

Input and Output Commands

MATLAB provides the following input and output related commands:

Command	Purpose
disp	Displays contents of an array or string.
fscanf	Read formatted data from a file.
format	Controls screen-display format.
fprintf	Performs formatted writes to screen or file.
input	Displays prompts and waits for input.
;	Suppresses screen printing.

The **fscanf** and **fprintf** commands behave like C scanf and printf functions. They support the following format codes:

Format Code	Purpose

%s	Format as a string.
%d	Format as an integer.
%f	Format as a floating point value.
%e	Format as a floating point value in scientific notation.
%g	Format in the most compact form: %f or %e.
\n	Insert a new line in the output string.
\t	Insert a tab in the output string.

The format function has the following forms used for numeric display:

Format Function	Display up to
format short	Four decimal digits (default).
format long	16 decimal digits.
format short e	Five digits plus exponent.
format long e	16 digits plus exponents.
format bank	Two decimal digits.
format +	Positive, negative, or zero.
format rat	Rational approximation.
format compact	Suppresses some line feeds.
format loose	Resets to less compact display mode.

Vector, Matrix, and Array Commands

The following table shows various commands used for working with arrays, matrices and vectors:

Command	Purpose
cat	Concatenates arrays.
find	Finds indices of nonzero elements.

length	Computes number of elements.
linspace	Creates regularly spaced vector.

logspace	Creates logarithmically spaced vector.
max	Returns largest element.
min	Returns smallest element.
prod	Product of each column.
reshape	Changes size.
size	Computes array size.
sort	Sorts each column.
sum	Sums each column.
eye	Creates an identity matrix.
ones	Creates an array of ones.
zeros	Creates an array of zeros.
cross	Computes matrix cross products.
dot	Computes matrix dot products.
det	Computes determinant of an array.
inv	Computes inverse of a matrix.
pinv	Computes pseudoinverse of a matrix.
rank	Computes rank of a matrix.
rref	Computes reduced row echelon form.
cell	Creates cell array.
celldisp	Displays cell array.
cellplot	Displays graphical representation of cell array.

num2cell	Converts numeric array to cell array.
deal	Matches input and output lists.
iscell	Identifies cell array.

Plotting Commands

MATLAB provides numerous commands for plotting graphs. The following table shows some of the commonly used commands for plotting:

Command	Purpose
axis	Sets axis limits.
fplot	Intelligent plotting of functions.
grid	Displays gridlines.
plot	Generates xy plot.
print	Prints plot or saves plot to a file.
title	Puts text at top of plot.
xlabel	Adds text label to x-axis.
ylabel	Adds text label to y-axis.
axes	Creates axes objects.
close	Closes the current plot.
close all	Closes all plots.
figure	Opens a new figure window.
gtext	Enables label placement by mouse.
hold	Freezes current plot.
legend	Legend placement by mouse.
refresh	Redraws current figure window.
set	Specifies properties of objects such as axes.

subplot	Creates plots in sub windows.
text	Places string in figure.
bar	Creates bar chart.
loglog	Creates log-log plot.
polar	Creates polar plot.
semilogx	Creates semi log plot. (logarithmic abscissa).
semilogy	Creates semi log plot. (logarithmic ordinate).
stairs	Creates stairs plot.
stem	Creates stem plot.

CHAPTER 5

THE M-FILES

So far, we have used MATLAB environment as a calculator. However, MATLAB is also a powerful programming language, as well as an interactive computational environment.

In previous chapters, you have learned how to enter commands from the MATLAB command prompt. MATLAB also allows you to write series of commands into a file and execute the file as complete unit, like writing a function and calling it.

The M Files

MATLAB allows writing two kinds of program files:

Scripts - script files are program files with **.m extension**. In these files, you write series of commands, which you want to execute together. Scripts do not accept inputs and do not return any outputs. They operate on data in the workspace.

Functions - functions files are also program files with **.m extension**. Functions can accept inputs and return outputs. Internal variables are local to the function.

You can use the MATLAB editor or any other text editor to create your **.m** files. In this section, we will discuss the script files. A script file contains multiple sequential lines of MATLAB commands and function calls. You can run a script by typing its name at the command line.

Creating and Running Script File

To create scripts files, you need to use a text editor. You can open the MATLAB editor in two ways:

Using the command prompt

Using the IDE

If you are using the command prompt, type **edit** in the command prompt. This will open the editor. You can directly type **edit** and then the filename (with .m extension)

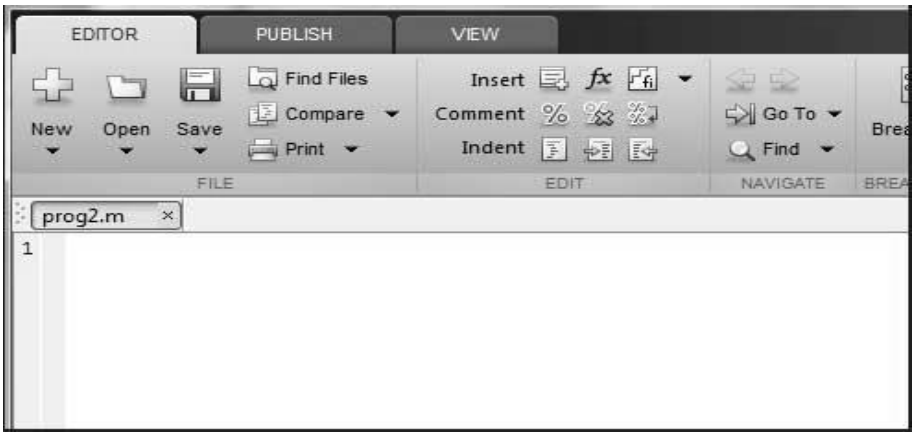
edit Or edit <filename>

The above command will create the file in default MATLAB directory. If you want to store all program files in a specific folder, then you will have to provide the entire path.

Let us create a folder named progs. Type the following commands at the command prompt(>>):

mkdir progs % create directory progs under default directory chdir progs % changing the current directory to progs edit prog1.m % creating an m file named prog1.m

If you are creating the file for first time, MATLAB prompts you to confirm it. Click Yes.



Alternatively, if you are using the IDE, choose NEW -> Script. This also opens the editor and creates a file named Untitled. You can name and save the file after typing the code.

Type the following code in the editor:

```
NoOfStudents = 6000;
TeachingStaff = 150;
NonTeachingStaff = 20;
Total = NoOfStudents + TeachingStaff ...
   + NonTeachingStaff;
disp(Total);
```

After creating and saving the file, you can run it in two ways:

Clicking the **Run** button on the editor window or

Just typing the filename (without extension) in the command prompt: >> prog1 The command window prompt displays the result:

```
6170
```

Example

Create a script file, and type the following code:

```
a = 5; b = 7; c = a + b d = c + sin(b) e = 5 * d f = exp(-d)
```

When the above code is compiled and executed, it produces the following result:

```
c =     12 d =
   12.6570 e =
   63.2849 f =
   3.1852e-06
```

CHAPTER 6

DATA TYPES

MATLAB does not require any type declaration or dimension statements. Whenever MATLAB encounters a new variable name, it creates the variable and allocates appropriate memory space.

If the variable already exists, then MATLAB replaces the original content with new content and allocates new storage space, where necessary.

For example,

Total = 42

The above statement creates a 1-by-1 matrix named 'Total' and stores the value 42 in it.

Data Types Available in MATLAB

MATLAB provides 15 fundamental data types. Every data type stores data that is in the form of a matrix or array. The size of this matrix or array is a minimum of 0-by-0 and this can grow up to a matrix or array of any size.

The following table shows the most commonly used data types in MATLAB:

Data Type	Description
int8	8-bit signed integer
uint8	8-bit unsigned integer
int16	16-bit signed integer
uint16	16-bit unsigned integer
int32	32-bit signed integer
uint32	32-bit unsigned integer
int64	64-bit signed integer
uint64	64-bit unsigned integer
single	single precision numerical data
double	double precision numerical data
logical	logical values of 1 or 0, represent true and false respectively
char	character data (strings are stored as vector of characters)

cell array	array of indexed cells, each capable of storing an array of a different dimension and data type
structure	C-like structures, each structure having named fields capable of storing an array of a different dimension and data type
function handle	pointer to a function
user classes	objects constructed from a user-defined class
java classes	objects constructed from a Java class

Example

Create a script file with the following code:

```
str = 'Hello World!' n = 2345 d = double(n) un = uint32(789.50) rn = 5678.92347 c = int32(rn)
```

When the above code is compiled and executed, it produces the following result:

```
str =
Hello World!
n =    2345 d =    2345 un =    790
rn =
  5.6789e+03
c =
  5679
```

Data Type Conversion

MATLAB provides various functions for converting a value from one data type to another. The following table shows the data type conversion functions:

Function	Purpose
Char	Convert to character array (string)
int2str	Convert integer data to string
mat2str	Convert matrix to string
num2str	Convert number to string
str2double	Convert string to double-precision value

str2num	Convert string to number
native2unicode	Convert numeric bytes to Unicode characters
unicode2native	Convert Unicode characters to numeric bytes
base2dec	Convert base N number string to decimal number
bin2dec	Convert binary number string to decimal number
dec2base	Convert decimal to base N number in string
dec2bin	Convert decimal to binary number in string
dec2hex	Convert decimal to hexadecimal number in string
hex2dec	Convert hexadecimal number string to decimal number
hex2num	Convert hexadecimal number string to double-precision number
num2hex	Convert singles and doubles to IEEE hexadecimal strings
cell2mat	Convert cell array to numeric array
cell2struct	Convert cell array to structure array
cellstr	Create cell array of strings from character array
mat2cell	Convert array to cell array with potentially different sized cells
num2cell	Convert array to cell array with consistently sized cells
struct2cell	Convert structure to cell array

Determination of Data Types

MATLAB provides various functions for identifying data type of a variable.

Following table provides the functions for determining the data type of a variable:

Function	Purpose
is	Detect state
isa	Determine if input is object of specified class
iscell	Determine whether input is cell array

iscellstr	Determine whether input is cell array of strings
ischar	Determine whether item is character array
isfield	Determine whether input is structure array field
isfloat	Determine if input is floating-point array
ishghandle	True for Handle Graphics object handles
isinteger	Determine if input is integer array
isjava	Determine if input is Java object
islogical	Determine if input is logical array
isnumeric	Determine if input is numeric array
isobject	Determine if input is MATLAB object
isreal	Check if input is real array
isscalar	Determine whether input is scalar
isstr	Determine whether input is character array
isstruct	Determine whether input is structure array
isvector	Determine whether input is vector
class	Determine class of object
validateattributes	Check validity of array
whos	List variables in workspace, with sizes and types

Example

Create a script file with the following code:

x = 3 isinteger(x) isfloat(x) isvector(x) isscalar(x) isnumeric(x)

x = 23.54 isinteger(x) isfloat(x) isvector(x) isscalar(x) isnumeric(x)

x = [1 2 3] isinteger(x) isfloat(x) isvector(x) isscalar(x)

x = 'Hello' isinteger(x) isfloat(x) isvector(x) isscalar(x) isnumeric(x)

When you run the file, it produces the following result:

```
x =      3
ans =      0
ans =      1
ans =      1
 ans =      1
ans =      1
 x =
   23.5400
```

```
ans =      0
ans =      1
ans =      1
ans =      1
ans =      1
x =
     1     2     3
ans =      0
ans =      1
ans =      1
ans =      0
x =
Hello
ans =      0
ans =
     0
ans =      1
ans =      0
ans =
     0
```

CHAPTER 7

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. MATLAB is designed to operate primarily on whole matrices and arrays. Therefore, operators in MATLAB work both on scalar and nonscalar data. MATLAB allows the following types of elementary operations:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operations
- Set Operations

Arithmetic Operators

MATLAB allows two different types of arithmetic operations:

- Matrix arithmetic operations
- Array arithmetic operations

Matrix arithmetic operations are same as defined in linear algebra. Array operations are executed element by element, both on one-dimensional and multidimensional array.

The matrix operators and array operators are differentiated by the period (.) symbol. However, as the addition and subtraction operation is same for matrices and arrays, the operator is same for both cases. The following table gives brief description of the operators:

Operator	Description
+	Addition or unary plus. A+B adds the values stored in variables A and B. A and B must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.
-	Subtraction or unary minus. A-B subtracts the value of B from A. A and B must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.
*	Matrix multiplication. C = A*B is the linear algebraic product of the matrices A and B. More precisely, $C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$ For non-scalar A and B, the number of columns of A must be equal to the number of rows of B. A scalar can multiply a matrix of any size.

.*	Array multiplication. A.*B is the element-by-element product of the arrays A and B. A and B must have the same size, unless one of them is a scalar.
/	Slash or matrix right division. B/A is roughly the same as B*inv(A). More precisely, B/A = (A'\B')'.
./	Array right division. A./B is the matrix with elements A(i,j)/B(i,j). A and B must have the same size, unless one of them is a scalar.
\	Backslash or matrix left division. If A is a square matrix, A\B is roughly the same as inv(A)*B, except it is computed in a different way. If A is an n-by-n matrix and B is a column vector with n components, or a matrix with several such columns, then X = A\B is the solution to the equation *AX = B*. A warning message is displayed if A is badly scaled or nearly singular.
.\	Array left division. A.\B is the matrix with elements B(i,j)/A(i,j). A and B must have the same size, unless one of them is a scalar.
^	Matrix power. X^p is X to the power p, if p is a scalar. If p is an integer, the power is computed by repeated squaring. If the integer is negative, X is inverted first. For other values of p, the calculation involves eigenvalues and eigenvectors, such that if [V,D] = eig(X), then X^p = V*D.^p/V.
.^	Array power. A.^B is the matrix with elements A(i,j) to the B(i,j) power. A and B must have the same size, unless one of them is a scalar.
'	Matrix transpose. A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.
.'	Array transpose. A.' is the array transpose of A. For complex matrices, this does not involve conjugation.

Example

The following examples show the use of arithmetic operators on scalar data. Create a script file with the following code:

```
a = 10; b = 20; c = a + b d = a - b e = a * b f = a / b g = a \ b x = 7; y = 3; z = x ^ y
```

When you run the file, it produces the following result:

```
c =     30
d =    -10
e =    200
f =
   0.5000
g =      2
z =
   343
```

Functions for Arithmetic Operations

Apart from the above-mentioned arithmetic operators, MATLAB provides the following commands/functions used for similar purpose:

Function	Description
uplus(a)	Unary plus; increments by the amount a
plus (a,b)	Plus; returns a + b
uminus(a)	Unary minus; decrements by the amount a
minus(a, b)	Minus; returns a - b
times(a, b)	Array multiply; returns a.*b
mtimes(a, b)	Matrix multiplication; returns a* b
rdivide(a, b)	Right array division; returns a ./ b
ldivide(a, b)	Left array division; returns a.\ b
mrdivide(A, B)	Solve systems of linear equations $xA = B$ for x
mldivide(A, B)	Solve systems of linear equations $Ax = B$ for x
power(a, b)	Array power; returns a.^b
mpower(a, b)	Matrix power; returns a ^ b

cumprod(A)	Cumulative product; returns an array of the same size as the array A containing the cumulative product.

	If A is a vector, then cumprod(A) returns a vector containing the cumulative product of the elements of A. If A is a matrix, then cumprod(A) returns a matrix containing the cumulative products for each column of A. If A is a multidimensional array, then cumprod(A) acts along the first non-singleton dimension.
cumprod(A, dim)	Returns the cumulative product along dimension *dim*.
cumsum(A)	Cumulative sum; returns an array A containing the cumulative sum. If A is a vector, then cumsum(A) returns a vector containing the cumulative sum of the elements of A. If A is a matrix, then cumsum(A) returns a matrix containing the cumulative sums for each column of A. If A is a multidimensional array, then cumsum(A) acts along the first non-singleton dimension.
cumsum(A, dim)	Returns the cumulative sum of the elements along dimension *dim*.
diff(X)	Differences and approximate derivatives; calculates differences between adjacent elements of X. If X is a vector, then diff(X) returns a vector, one element shorter than X, of differences between adjacent elements: [X(2)-X(1) X(3)-X(2) ... X(n)X(n-1)] If X is a matrix, then diff(X) returns a matrix of row differences: [X(2:m,:)-X(1:m-1,:)]
diff(X,n)	Applies *diff* recursively n times, resulting in the nth difference.
diff(X,n,dim)	It is the nth difference function calculated along the dimension specified by scalar dim. If order n equals or exceeds the length of dimension dim, diff returns an empty array.

prod(A)	Product of array elements; returns the product of the array elements of A. If A is a vector, then prod(A) returns the product of the elements. If A is a nonempty matrix, then prod(A) treats the columns of A as vectors and returns a row vector of the products of each column. If A is an empty 0-by-0 matrix, prod(A) returns 1. If A is a multidimensional array, then prod(A) acts along the first non-singleton dimension and returns an array of products. The size of this dimension reduces to 1 while the sizes of all other dimensions remain the same. The prod function computes and returns B as single if the input, A, is single. For all other numeric and logical data types, prod computes and returns B as double
prod(A,dim)	Returns the products along dimension dim. For example, if A is a matrix, prod(A,2) is a column vector containing the products of each row.
prod(___,datatype)	Multiplies in and returns an array in the class specified by datatype.
sum(A)	Sum of array elements; returns sums along different dimensions of an array. If A is floating point, that is double or single, B is accumulated natively, that is in the same class as A, and B has the same class as A. If A is not floating point, B is accumulated in double and B has class double. If A is a vector, sum(A) returns the sum of the elements.
	If A is a matrix, sum(A) treats the columns of A as vectors, returning a row vector of the sums of each column. If A is a multidimensional array, sum(A) treats the values along the first non-singleton dimension as vectors, returning an array of row vectors.
sum(A,dim)	Sums along the dimension of *A* specified by scalar *dim*.
sum(..., 'double') sum(..., dim,'double')	Perform additions in double-precision and return an answer of type double, even if A has data type single or an integer data type. This is the default for integer data types.

sum(..., 'native') sum(..., dim,'native')	Perform additions in the native data type of A and return an answer of the same data type. This is the default for single and double.
ceil(A)	Round toward positive infinity; rounds the elements of A to the nearest integers greater than or equal to A.
fix(A)	Round toward zero
floor(A)	Round toward negative infinity; rounds the elements of A to the nearest integers less than or equal to A.
idivide(a, b) idivide(a, b,'fix')	Integer division with rounding option; is the same as a./b except that fractional quotients are rounded toward zero to the nearest integers.
idivide(a, b, 'round')	Fractional quotients are rounded to the nearest integers.
idivide(A, B, 'floor')	Fractional quotients are rounded toward negative infinity to the nearest integers.
idivide(A, B, 'ceil')	Fractional quotients are rounded toward infinity to the nearest integers.
mod (X,Y)	Modulus after division; returns X - n.*Y where n = floor(X./Y). If Y is not an integer and the quotient X./Y is within round off error of an integer, then n is that integer. The inputs X and Y must be real arrays of the same size, or real scalars (provided Y ~=0). Please note: mod(X,0) is X mod(X,X) is 0 mod(X,Y) for X~=Y and Y~=0 has the same sign as Y
rem (X,Y)	Remainder after division; returns X - n.*Y where n = fix(X./Y). If Y is not an integer and the quotient X./Y is within round off error of an integer, then n is that integer. The inputs X and Y must be real arrays of the same size, or real scalars (provided Y ~=0). Please note that: rem(X,0) is NaN rem(X,X) for X~=0 is 0 rem(X,Y) for X~=Y and Y~=0 has the same sign as X.
round(X)	Round to nearest integer; rounds the elements of X to the nearest integers. Positive elements with a fractional part of 0.5 round up to the nearest positive integer. Negative elements with a fractional part of 0.5 round down to the nearest negative integer.

Relational Operators

Relational operators can also work on both scalar and non-scalar data. Relational operators for arrays perform element-by-element comparisons between two arrays and return a logical array of the same size, with elements set to logical 1 (true) where the relation is true and elements set to logical 0 (false) where it is not.

The following table shows the relational operators available in MATLAB:

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Example

Create a script file and type the following code:

```
a = 100; b = 200;
if (a >= b)
max = a
else
max = b
end
```

When you run the file, it produces following result:

```
max =
  200
```

Apart from the above-mentioned relational operators, MATLAB provides the following commands/functions used for the same purpose:

Function	Description
eq(a, b)	Tests whether a is equal to b
ge(a, b)	Tests whether a is greater than or equal to b

gt(a, b)	Tests whether a is greater than b
le(a, b)	Tests whether a is less than or equal to b
lt(a, b)	Tests whether a is less than b
ne(a, b)	Tests whether a is not equal to b
isequal	Tests arrays for equality
isequaln	Tests arrays for equality, treating NaN values as equal

Example

Create a script file and type the following code:

```
% comparing two values a = 100; b = 200; if (ge(a,b)) max = a else
max = b end
% comparing two different values a = 340; b = 520;
if (le(a, b)) disp(' a is either less than or equal to b') else disp(' a is greater than b') end
```

When you run the file, it produces the following result:

```
max =    200  a is either less than or equal to b
```

Logical Operators

MATLAB offers two types of logical operators and functions:

Element-wise - These operators operate on corresponding elements of logical arrays.

Short-circuit - These operators operate on scalar and logical expressions.

Element-wise logical operators operate element-by-element on logical arrays. The symbols &, |, and ~ are the logical array operators AND, OR, and NOT.

Short-circuit logical operators allow short-circuiting on logical operations. The symbols && and || are the logical short-circuit operators AND and OR. **Example**

Create a script file and type the following code:

```
a = 5; b = 20;
if ( a && b )
disp('Line 1 - Condition is true');
end
if ( a || b )
disp('Line 2 - Condition is true');
end
% lets change the value of a and b
```

```
a = 0;    b = 10;
if ( a && b )
disp('Line 3 - Condition is true');
else
disp('Line 3 - Condition is not true');
end
if (~(a && b))
disp('Line 4 - Condition is true');
end
```

When you run the file, it produces following result:

Line 1 - Condition is true

Line 2 - Condition is true

Line 3 - Condition is not true

Line 4 - Condition is true

Functions for Logical Operations

Apart from the above-mentioned logical operators, MATLAB provides the following commands or functions used for the same purpose:

Function	Description
and(A, B)	Finds logical AND of array or scalar inputs; performs a logical AND of all input arrays A, B, etc. and returns an array containing elements set to either logical 1 (true) or logical 0 (false). An
	element of the output array is set to 1 if all input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.
not(A)	Finds logical NOT of array or scalar input; performs a logical NOT of input array A and returns an array containing elements set to either logical 1 (true) or logical 0 (false). An element of the output array is set to 1 if the input array contains a zero value element at that same array location. Otherwise, that element is set to 0.
or(A, B)	Finds logical OR of array or scalar inputs; performs a logical OR of all input arrays A, B, etc. and returns an array containing elements set to either logical 1 (true) or logical 0 (false). An

	element of the output array is set to 1 if any input arrays contain a nonzero element at that same array location. Otherwise, that element is set to 0.
xor(A, B)	Logical exclusive-OR; performs an exclusive OR operation on the corresponding elements of arrays A and B. The resulting element C(i,j,...) is logical true (1) if A(i,j,...) or B(i,j,...), but not both, is nonzero.
all(A)	Determine if all array elements of array A are nonzero or true. If A is a vector, all(A) returns logical 1 (true) if all the elements are nonzero and returns logical 0 (false) if one or more elements are zero. If A is a nonempty matrix, all(A) treats the columns of A as vectors, returning a row vector of logical 1's and 0's. If A is an empty 0-by-0 matrix, all(A) returns logical 1 (true). If A is a multidimensional array, all(A) acts along the first non-singleton dimension and returns an array of logical values. The size of this dimension
	reduces to 1 while the sizes of all other dimensions remain the same.
all(A, dim)	Tests along the dimension of A specified by scalar *dim*.

any(A)	Determine if any array elements are nonzero; tests whether any of the elements along various dimensions of an array is a nonzero number or is logical 1 (true). The *any* function ignores entries that are NaN (Not a Number). If A is a vector, any(A) returns logical 1 (true) if any of the elements of A is a nonzero number or is logical 1 (true), and returns logical 0 (false) if all the elements are zero. If A is a nonempty matrix, any(A) treats the columns of A as vectors, returning a row vector of logical 1's and 0's. If A is an empty 0-by-0 matrix, any(A) returns logical 0 (false). If A is a multidimensional array, any(A) acts along the first non-singleton dimension and returns an array of logical values. The size of this dimension reduces to 1 while the sizes of all other dimensions remain the same.
any(A,dim)	Tests along the dimension of A specified by scalar *dim*.
False	Logical 0 (false)
false(n)	is an n-by-n matrix of logical zeros
false(m, n)	is an m-by-n matrix of logical zeros.
false(m, n, p, ...)	is an m-by-n-by-p-by-... array of logical zeros.
false(size(A))	is an array of logical zeros that is the same size as array A.
false(...,'like',p)	is an array of logical zeros of the same data type and sparsity as the logical array p.
ind = find(X)	Find indices and values of nonzero elements; locates all nonzero elements of array X, and returns the linear indices of those elements in a vector. If X is a row vector, then the returned vector is a row vector; otherwise, it returns a column vector. If X contains no nonzero elements or is an empty array, then an empty array is returned.

ind = find(X, k) ind = find(X, k, 'first')	Returns at most the first k indices corresponding to the nonzero entries of X. k must be a positive integer, but it can be of any numeric data type.
ind = find(X, k, 'last')	returns at most the last k indices corresponding to the nonzero entries of X.
[row,col] = find(X, ...)	Returns the row and column indices of the nonzero entries in the matrix X. This syntax is especially useful when working with sparse matrices. If X is an N-dimensional array with N > 2, col contains linear indices for the columns.
[row,col,v] = find(X, ...)	Returns a column or row vector v of the nonzero entries in X, as well as row and column indices. If X is a logical expression, then v is a logical array. Output v contains the non-zero elements of the logical array obtained by evaluating the expression X.
islogical(A)	Determine if input is logical array; returns true if A is a logical array and false otherwise. It also returns true if A is an instance of a class that is derived from the logical class.
logical(A)	Convert numeric values to logical; returns an array that can be used for logical indexing or logical tests.
True	Logical 1 (true)
true(n)	is an n-by-n matrix of logical ones.
true(m, n)	is an m-by-n matrix of logical ones.
true(m, n, p, ...)	is an m-by-n-by-p-by-... array of logical ones.
true(size(A))	is an array of logical ones that is the same size as array A.
true(...,'like', p)	is an array of logical ones of the same data type and sparsity as the logical array p.

Example

Create a script file and type the following code:

a = 60; % 60 = 0011 1100

b = 13; % 13 = 0000 1101

```
c = bitand(a, b)      % 12 = 0000 1100
c = bitor(a, b)       % 61 = 0011 1101
c = bitxor(a, b)      % 49 = 0011 0001
c = bitshift(a, 2)    % 240 = 1111 0000 */
c = bitshift(a,-2)    % 15 = 0000 1111 */
```

When you run the file, it displays the following result:

```
c =     12
c =
   61
c =     49
c =    240
 c =
   15
```

Bitwise Operations

Bitwise operators work on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

```
   A      = 0011 1100
   B      = 0000 1101
----------------- A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A  = 1100 0011
```

MATLAB provides various functions for bit-wise operations like 'bitwise and', 'bitwise or' and 'bitwise not' operations, shift operation, etc.

The following table shows the commonly used bitwise operations:

Function	Purpose
bitand(a, b)	Bit-wise AND of integers *a* and *b*
bitcmp(a)	Bit-wise complement of *a*
bitget(a,pos)	Get bit at specified position *pos*, in the integer array *a*
bitor(a, b)	Bit-wise OR of integers *a* and *b*
bitset(a, pos)	Set bit at specific location *pos* of *a*
bitshift(a, k)	Returns *a* shifted to the left by *k* bits, equivalent to multiplying by 2^k. Negative values of k correspond to shifting bits right or dividing by $2^{\|k\|}$ and rounding to the nearest integer towards negative infinite. Any overflow bits are truncated.
bitxor(a, b)	Bit-wise XOR of integers *a* and *b*
swapbytes	Swap byte ordering

Example

Create a script file and type the following code:

```
a = 60; % 60 = 0011 1100
b = 13; % 13 = 0000 1101
c = bitand(a, b)      % 12 = 0000 1100
c = bitor(a, b)       % 61 = 0011 1101
c = bitxor(a, b)      % 49 = 0011 0001
c = bitshift(a, 2)    % 240 = 1111 0000 */
c = bitshift(a,-2)    % 15 = 0000 1111 */
```

When you run the file, it displays the following result:

```
c =     12
c =     61
c =     49
c =    240
c =
   15
```

Set Operations

MATLAB provides various functions for set operations, like union, intersection and testing for set membership, etc.

The following table shows some commonly used set operations:

Function	Description
intersect(A,B)	Set intersection of two arrays; returns the values common to both A and B. The values returned are in sorted order.
intersect(A,B,'rows')	Treats each row of A and each row of B as single entities and returns the rows common to both A and B. The rows of the returned matrix are in sorted order.
ismember(A,B)	Returns an array the same size as A, containing 1 (true) where the elements of A are found in B. Elsewhere, it returns 0 (false).
ismember(A,B,'rows')	Treats each row of A and each row of B as single entities and returns a vector containing 1 (true) where the rows of matrix A are also rows of B. Elsewhere, it returns 0 (false).
issorted(A)	Returns logical 1 (true) if the elements of A are in sorted order and logical 0 (false) otherwise. Input A can be a vector or an N-by-1 or 1-by-N cell array of strings. A is considered to be sorted if A and the output of sort(A) are equal.
issorted(A, 'rows')	Returns logical 1 (true) if the rows of two-dimensional matrix A are in sorted order, and logical 0 (false) otherwise. Matrix A is considered to be sorted if A and the output of sortrows(A) are equal.
setdiff(A,B)	Sets difference of two arrays; returns the values in A that are not in B. The values in the returned array are in sorted order.
setdiff(A,B,'rows')	Treats each row of A and each row of B as single entities and returns the rows from A that are not in B. The rows of the returned matrix are in sorted order. The 'rows' option does not support cell arrays.
setxor	Sets exclusive OR of two arrays
union	Sets union of two arrays

unique	Unique values in array

Example

Create a script file and type the following code:

a = [7 23 14 15 9 12 8 24 35] b = [2 5 7 8 14 16 25 35 27] u = union(a, b)

i = intersect(a, b) s = setdiff(a, b)

When you run the file, it produces the following result:

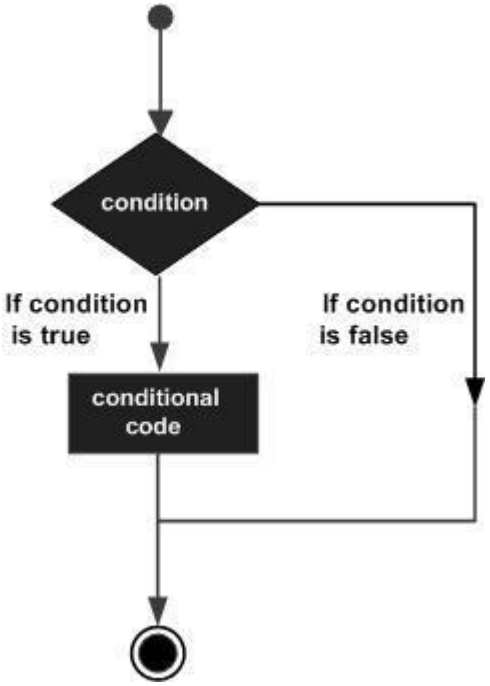
```
a =
    7    23    14    15     9    12     8    24    35
b =
    2     5     7     8    14    16    25    35    27
u =
  Columns 1 through 11
    2     5     7     8     9    12    14    15    16    23    24
  Columns 12 through 14
 25    27    35
i =
    7     8    14    35
s =
    9    12    15    23    24
```

CHAPTER 8

DECISION MAKING

Decision making structures require that the programmer should specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



MATLAB provides following types of decision making statements. Click the following links to check their detail:

Statement	Description
if ... end statement	An if ... end statement consists of a boolean expression followed by one or more statements.
if...else...end statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
If... elseif...elseif...else...end statements	An if statement can be followed by one (or more) optional elseif... and an else statement, which is very useful to test various conditions.

nested if statements	You can use one if or elseif statement inside another if or elseif statement(s).
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one switch statement inside another switch statement(s).

if... end Statement

An **if ... end** statement consists of an **if** statement and a boolean expression followed by one or more statements. It is delimited by the **end** statement.

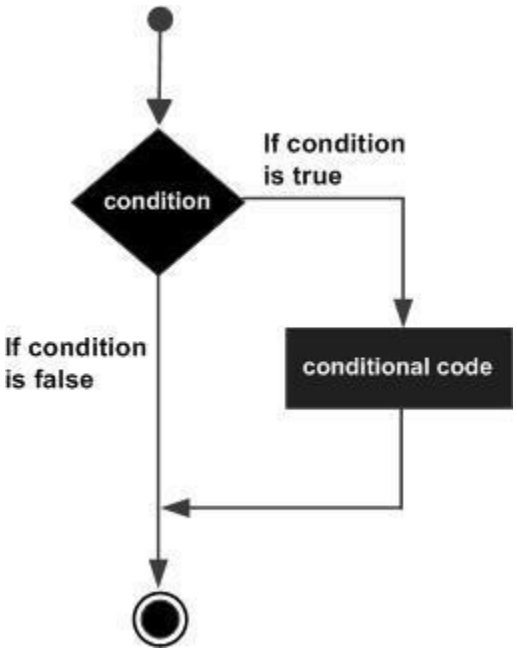
Syntax

The syntax of an if statement in MATLAB is:

```
if <expression>
% statement(s) will execute if the boolean expression is true
<statements> end
```

If the expression evaluates to true, then the block of code inside the if statement will be executed. If the expression evaluates to false, then the first set of code after the end statement will be executed.

Flow Diagram



Example

Create a script file and type the following code:

```
a = 10;
% check the condition using if statement
if a < 20
  % if condition is true then print the following
fprintf('a is less than 20\n' );
end
fprintf('value of a is : %d\n', a);
```

When you run the file, it displays the following result:

a is less than 20 value of a is : 10

if...else...end Statement

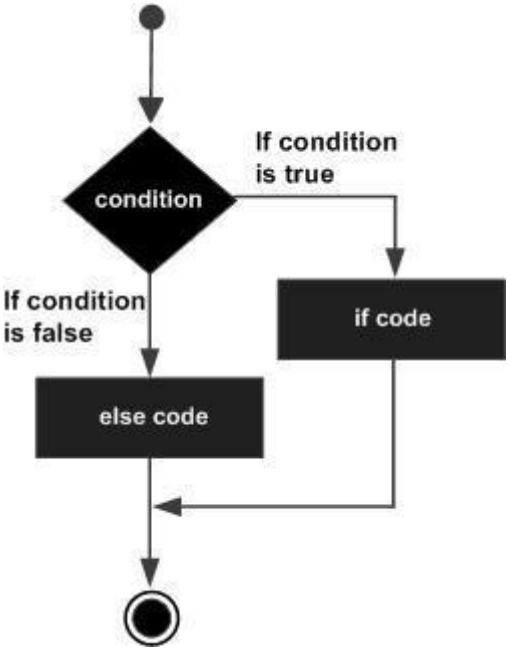
An if statement can be followed by an optional else statement, which executes when the expression is false.

Syntax

The syntax of an if...else statement in MATLAB is:

```
if <expression>
% statement(s) will execute if the boolean expression is true
<statement(s)> else
<statement(s)>
% statement(s) will execute if the boolean expression is false  end
```

If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed. Flow Diagram



Example

Create a script file and type the following code:

```
a = 100;
% check the boolean condition     if a < 20
      % if condition is true then print the following          fprintf('a is
less than 20\n' );    else
    % if condition is false then print the following         fprintf('a is not less
than 20\n' );    end    fprintf('value of a is : %d\n', a);
```

When the above code is compiled and executed, it produces the following result:

a is not less than 20 value of a is : 100

if...elseif...elseif...else...end Statements

An **if** statement can be followed by one (or more) optional **elseif...** and an **else** statement, which is very useful to test various conditions.

When using if... elseif...else statements, there are few points to keep in mind:

An if can have zero or one else's and it must come after any elseif's.

An if can have zero to many elseif's and they must come before the else.

Once an else if succeeds, none of the remaining elseif's or else's will be tested.

Syntax

```
if <expression 1>
% Executes when the expression 1 is true
<statement(s)>
elseif <expression 2>
% Executes when the boolean expression 2 is true
<statement(s)>
Elseif <expression 3>
% Executes when the boolean expression 3 is true
<statement(s)>
else
%  executes when the none of the above condition is true
<statement(s)>
end
```

Example

Create a script file and type the following code in it:

```
a = 100;
%check the boolean condition
if a == 10
      % if condition is true then print the following
fprintf('Value of a is 10\n' );
elseif( a == 20 )
     % if else if condition is true
fprintf('Value of a is 20\n' );
elseif a == 30
      % if else if condition is true
fprintf('Value of a is 30\n' );
else
      % if none of the conditions is true '
fprintf('None of the values are matching\n');
fprintf('Exact value of a is: %d\n', a );
end
```

When the above code is compiled and executed, it produces the following result:

None of the values are matching

Exact value of a is: 100

The Nested if Statements

It is always legal in MATLAB to nest if-else statements which means you can use one if or elseif statement inside another if or elseif statement(s).

Syntax

The syntax for a nested if statement is as follows:

if <expression 1>

% Executes when the boolean expression 1 is true if <expression 2>

  % Executes when the boolean expression 2 is true end end

You can nest elseif...else in the similar way as you have nested if statement. **Example**

Create a script file and type the following code in it:

```
a = 100; b = 200;
   % check the boolean condition
if( a == 100 )
        % if condition is true then check the following
 if( b == 200 )
        % if condition is true then print the following
fprintf('Value of a is 100 and b is 200\n' );
end
end
fprintf('Exact value of a is : %d\n', a );
fprintf('Exact value of b is : %d\n', b );
```

When you run the file, it displays:

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

The switch Statement

A switch block conditionally executes one set of statements from several choices. Each choice is covered by a case statement.

An evaluated switch_expression is a scalar or string.

An evaluated case_expression is a scalar, a string or a cell array of scalars or strings.

The switch block tests each case until one of the cases is true. A case is true when:

For numbers, eq(case_expression,switch_expression).

For strings, strcmp(case_expression,switch_expression).

For objects that support the eq function,eq(case_expression,switch_expression).

For a cell array case_expression, at least one of the elements of the cell array matches switch_expression, as defined above for numbers, strings and objects.

When a case is true, MATLAB executes the corresponding statements and then exits the switch block.

The **otherwise** block is optional and executes only when no case is true.

Syntax

The syntax of switch statement in MATLAB is:

```
switch <switch_expression>   case <case_expression>
   <statements>   case <case_expression>
   <statements>    ...    ...
  otherwise
    <statements> end
```

Example

Create a script file and type the following code in it:

```
grade = 'B';
switch(grade)
case 'A'
fprintf('Excellent!\n');
case 'B'
fprintf('Well done\n' );
case 'C'
fprintf('Well done\n' );
case 'D'
fprintf('You passed\n' );
case 'F'
fprintf('Better try again\n' );
otherwise
fprintf('Invalid grade\n' );
end
```

When you run the file, it displays:

Well done

Your grade is B

The Nested Switch Statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a nested switch statement is as follows:

switch(ch1) case 'A' fprintf('This A is part of outer switch'); switch(ch2) case 'A' fprintf('This A is part of inner switch'); case 'B' fprintf('This B is part of inner switch'); end case 'B' fprintf('This B is part of outer switch'); end

Example

Create a script file and type the following code in it:

```
a = 100; b = 200;
switch(a)
case 100
fprintf('This is part of outer switch %d\n', a );
switch(b)
case 200
fprintf('This is part of inner switch %d\n', a );
end
end
fprintf('Exact value of a is : %d\n', a );
fprintf('Exact value of b is : %d\n', b );
```

When you run the file, it displays:

```
This is part of outer switch 100
This is part of inner switch 100
Exact value of a is : 100
Exact value of b is : 200
```

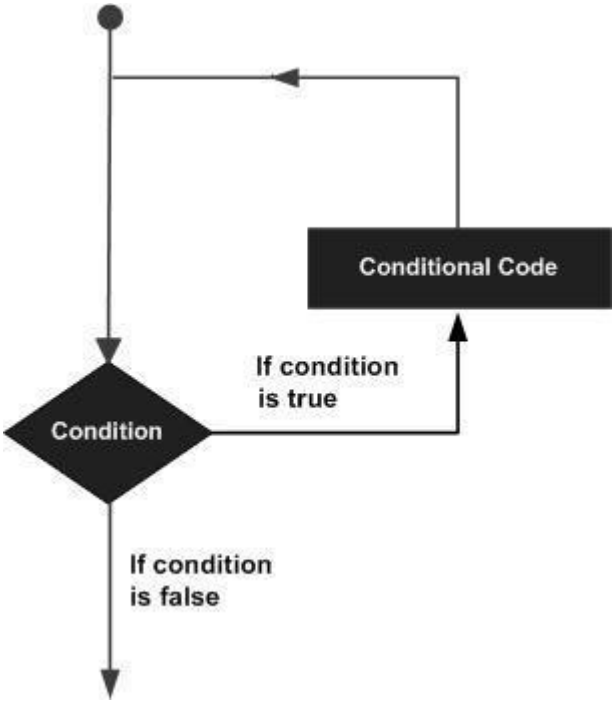
CHAPTER 9

LOOPS

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



MATLAB provides following types of loops to handle looping requirements. Click the following links to check their detail:

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loops inside any another loop.

The while Loop

The while loop repeatedly executes statements while condition is true.

Syntax

The syntax of a while loop in MATLAB is:

while <expression> <statements> end

The while loop repeatedly executes program statement(s) as long as the expression remains true.

An expression is true when the result is nonempty and contains all nonzero elements (logical or real numeric). Otherwise, the expression is false.

Example

Create a script file and type the following code:

```
a = 10;
% while loop execution  while( a < 20 )
fprintf('value of a: %d\n', a);
a = a + 1;
end
```

When you run the file, it displays the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The for Loop

A **for loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for loop** in MATLAB is:

for index = values <program statements> ...

end

values has one of the following forms:

Format	Description

initval:endval	increments the index variable from *initval* to *endval* by 1, and repeats execution of *program statements* until *index* is greater than *endval*.
initval:step:endval	increments *index* by the value step on each iteration, or decrements when step is negative.
valArray	creates a column vector *index* from subsequent columns of array*valArray* on each iteration. For example, on the first iteration, index = valArray(:,1). The loop executes for a maximum of n times, where n is the number of columns of *valArray*, given by numel(valArray, 1, :). The input *valArray* can be of any MATLAB data type, including a string, cell array, or struct.

Example 1

Create a script file and type the following code:

```
for a = 10:20
fprintf('value of a: %d\n', a);
end
```

When you run the file, it displays the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

Example 2

Create a script file and type the following code:

```
for a = 1.0: -0.1: 0.0
disp(a)
end
```

When you run the file, it displays the following result:

```
1
   0.9000
   0.8000
   0.7000
   0.6000
   0.5000
   0.4000
   0.3000
   0.2000
   0.1000
    0
```

Example 3

Create a script file and type the following code:

```
for a = [24,18,17,23,28]
disp(a)
end
```

When you run the file, it displays the following result:

```
   24
    18
    17
    23
    28
```

The Nested Loops

MATLAB allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax

The syntax for a nested for loop statement in MATLAB is as follows:

```
for m = 1:j
for n = 1:k
<statements>;
end
end
```

The syntax for a nested while loop statement in MATLAB is as follows:

while <expression1> while <expression2> <statements> end end

Example

Let us use a nested for loop to display all the prime numbers from 1 to 100. Create a script file and type the following code:

```
for i=2:100
for j=2:100
if(~mod(i,j))
break; % if factor found, not prime
end
end
if(j > (i/j))
fprintf('%d is prime\n', i);
end
end
```

When you run the file, it displays the following result:

```
 2    is prime
 3    is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
```

79 is prime

83 is prime

89 is prime

97 is prime

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

MATLAB supports the following control statements. Click the following links to check their detail.

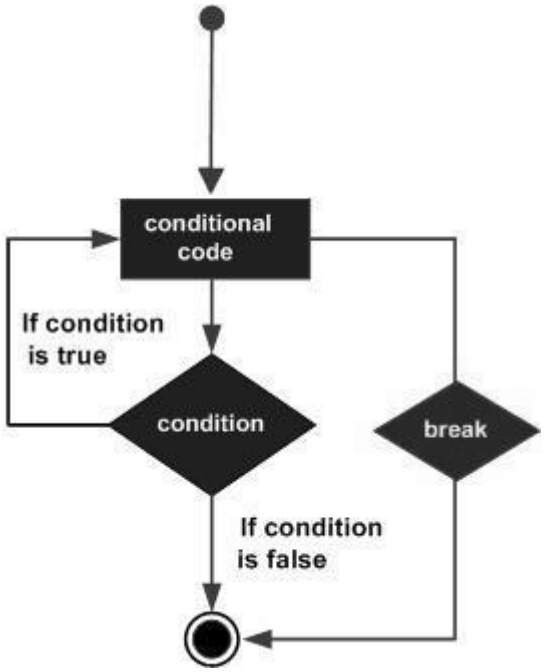
Control Statement	Description
break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

The break Statement

The break statement terminates execution of **for** or **while** loop. Statements in the loop that appear after the break statement are not executed.

In nested loops, break exits only from the loop in which it occurs. Control passes to the statement following the end of that loop.

Flow Diagram



Example

Create a script file and type the following code:

```
a = 10;
% while loop execution
while (a < 20 )
fprintf('value of a: %d\n', a);
a = a+1;
if( a > 15)
      % terminate the loop using break statement
break;
end
end
```

When you run the file, it displays the following result:

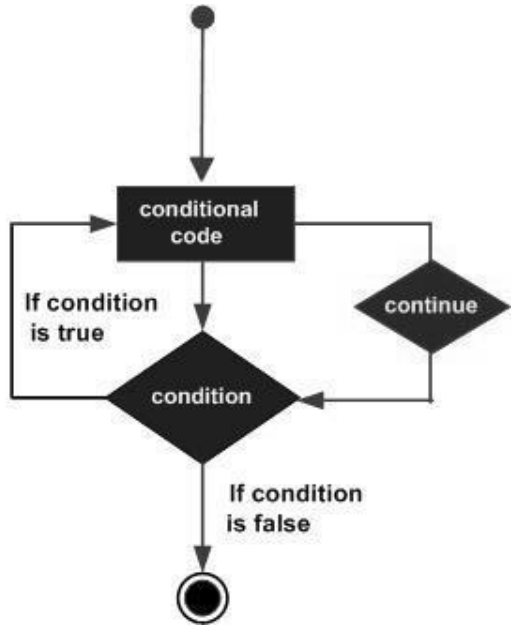
value of a: 10 value of a: 11 value of a: 12 value of a: 13 value of a: 14 value of a: 15

The continue Statement

The continue statement is used for passing control to next iteration of for or while loop.

The continue statement in MATLAB works somewhat like the break statement. Instead of forcing termination, however, 'continue' forces the next iteration of the loop to take place, skipping any code in between.

Flow Diagram



Example

Create a script file and type the following code:

```
a = 10;
%while loop execution
while a < 20
if a == 15
    % skip the iteration
a = a + 1;
continue;
end
fprintf('value of a: %d\n', a);
a = a + 1;
end
```

When you run the file, it displays the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

CHAPTER 10

ARRAYS

All variables of all data types in MATLAB are multidimensional arrays. A vector is a one-dimensional array and a matrix is a two-dimensional array.

We have already discussed vectors and matrices. In this chapter, we will discuss multidimensional arrays. However, before that, let us discuss some special types of arrays.

Special Arrays in MATLAB

In this section, we will discuss some functions that create some special arrays. For all these functions, a single argument creates a square array, double arguments create rectangular array.

The **zeros()** function creates an array of all zeros:

For example:

```
zeros(5)
```

MATLAB will execute the above statement and return the following result:

```
ans =
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
```

The **ones()** function creates an array of all ones:

For example:

```
ones(4,3)
```

MATLAB will execute the above statement and return the following result:

```
ans =
    1    1    1
    1    1    1
    1    1    1
    1    1    1
```

The **eye()** function creates an identity matrix.

For example:

```
eye(4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
    1    0    0    0
    0    1    0    0
```

0 0 1 0

0 0 0 1

The **rand()** function creates an array of uniformly distributed random numbers on (0,1):

For example:

rand(3, 5)

MATLAB will execute the above statement and return the following result:

ans =

0.8147 0.9134 0.2785 0.9649 0.9572

0.9058 0.6324 0.5469 0.1576 0.4854

0.1270 0.0975 0.9575 0.9706 0.8003

A Magic Square

A **magic square** is a square that produces the same sum, when its elements are added row-wise, column-wise or diagonally.

The **magic()** function creates a magic square array. It takes a singular argument that gives the size of the square. The argument must be a scalar greater than or equal to 3.

magic(4)

MATLAB will execute the above statement and return the following result:

ans =

16 2 3 13

5 11 10 8

9 7 6 12

4 14 15 1

Multidimensional Arrays

An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal twodimensional matrix.

Generally to generate a multidimensional array, we first create a two-dimensional array and extend it.

For example, let's create a two-dimensional array a.

a = [7 9 5; 6 1 9; 4 3 2]

MATLAB will execute the above statement and return the following result:

a =

7 9 5

6 1 9

4 3 2

The array *a* is a 3-by-3 array; we can add a third dimension to *a*, by providing the values like:

```
a(:, :, 2)= [ 1 2 3; 4 5 6; 7 8 9]
```

MATLAB will execute the above statement and return the following result:

```
a(:,:,1) =
    7    9    5
    6    1    9    4    3    2
a(:,:,2) =
    1    2    3
    4    5    6
    7    8    9
```

We can also create multidimensional arrays using the ones(), zeros() or the rand() functions.

For example,

```
b     = rand(4,3,2)
```

MATLAB will execute the above statement and return the following result:

```
b(:,:,1) =
   0.0344   0.7952   0.6463
   0.4387   0.1869   0.7094
   0.3816   0.4898   0.7547
   0.7655   0.4456   0.2760
b(:,:,2) =
   0.6797   0.4984   0.2238
   0.6551   0.9597   0.7513
   0.1626   0.3404   0.2551
   0.1190   0.5853   0.5060
```

We can also use the **cat()** function to build multidimensional arrays. It concatenates a list of arrays along a specified dimension:

Syntax for the cat() function is:

```
B = cat(dim, A1, A2...)
```

Where,

B is the new array created

A1, *A2*, ... are the arrays to be concatenated

dim is the dimension along which to concatenate the arrays

Example

Create a script file and type the following code into it:

```
a = [9 8 7; 6 5 4; 3 2 1]; b = [1 2 3; 4 5 6; 7 8 9]; c = cat(3, a, b, [ 2 3 1; 4 7 8; 3 9 0])
```

When you run the file, it displays:

```
c(:,:,1) =
    9    8    7
    6    5    4    3    2    1
c(:,:,2) =
         1         2    3
    4    5    6    7    8    9
c(:,:,3) =
         2         3    1
    4    7    8
         3         9    0
```

Array Functions

MATLAB provides the following functions to sort, rotate, permute, reshape, or shift array contents.

Function	Purpose
length	Length of vector or largest array dimension
ndims	Number of array dimensions
numel	Number of array elements
size	Array dimensions
iscolumn	Determines whether input is column vector
isempty	Determines whether array is empty
ismatrix	Determines whether input is matrix
isrow	Determines whether input is row vector
isscalar	Determines whether input is scalar
isvector	Determines whether input is vector
blkdiag	Constructs block diagonal matrix from input arguments

circshift	Shifts array circularly
ctranspose	Complex conjugate transpose
diag	Diagonal matrices and diagonals of matrix
flipdim	Flips array along specified dimension
fliplr	Flips matrix from left to right
flipud	Flips matrix up to down
ipermute	Inverses permute dimensions of N-D array
permute	Rearranges dimensions of N-D array
repmat	Replicates and tile array
reshape	Reshapes array
rot90	Rotates matrix 90 degrees
shiftdim	Shifts dimensions
issorted	Determines whether set elements are in sorted order
sort	Sorts array elements in ascending or descending order
sortrows	Sorts rows in ascending order
squeeze	Removes singleton dimensions
transpose	Transpose
vectorize	Vectorizes expression

Examples

The following examples illustrate some of the functions mentioned above.

Length, Dimension and Number of elements:

Create a script file and type the following code into it:

```
x = [7.1, 3.4, 7.2, 28/4, 3.6, 17, 9.4, 8.9];
length(x)  % length of x vector
y = rand(3, 4, 5, 2);
```

```
ndims(y)    % no of dimensions in array y
s = ['Zara', 'Nuha', 'Shamim', 'Riz', 'Shadab'];
numel(s)   % no of elements in s
```

When you run the file, it displays the following result:

```
ans =      8
ans =      4
ans =
   23
```

Circular Shifting of the Array Elements:

Create a script file and type the following code into it:

```
a = [1 2 3; 4 5 6; 7 8 9]  % the original array a
b = circshift(a,1)  %  circular shift first dimension values down by 1.
c = circshift(a,[1 -1]) % circular shift first dimension values % down by 1
                    % and second dimension values to the left % by 1.
```

When you run the file, it displays the following result:

```
a =
    1    2    3
    4    5    6    7    8    9
b =
        78    9
    1    2    3    4    5    6
 c =
        89    7
    2    3    1
    5    6    4
```

Sorting Arrays

Create a script file and type the following code into it:

```
v = [ 23 45 12 9 5 0 19 17]  % horizonal vector sort(v)    %sorting v m = [2 6 4; 5 3 9; 2 0 1]  %  two
dimensional array sort(m, 1)   % sorting m along the row sort(m, 2)   % sorting m along the column
```

When you run the file, it displays the following result:

```
v =
   23    45    12     9     5     0    19    17 ans =
    0     5     9    12    17    19    23    45 m =
```

```
    2    6    4
    5    3    9
    2    0    1
112
ans =
    2    0    1
    2    3    4    5    6    9 ans =
         2         4    6
         3         5    9
    0    1    2
```

Cell Array

Cell arrays are arrays of indexed cells where each cell can store an array of a different dimensions and data types.

The **cell** function is used for creating a cell array. Syntax for the cell function is:

C = cell(dim)

```
C     = cell(dim1,...,dimN)
D     = cell(obj)
```

Where,

C is the cell array;

dim is a scalar integer or vector of integers that specifies the dimensions of cell array C;

dim1, ... , dimN are scalar integers that specify the dimensions of C; *obj* is One of the following:

Java array or object

.NET array of type System.String or System.Object

Example

Create a script file and type the following code into it:

c = cell(2, 5); c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5} When you run the file, it displays the following result:

```
c =
  'Red'   'Blue'   'Green'   'Yellow'   'White'
  [ 1]    [ 2]    [  3]    [   4]    [   5]
```

Accessing Data in Cell Arrays

There are two ways to refer to the elements of a cell array:

Enclosing the indices in first bracket (), to refer to sets of cells

Enclosing the indices in braces {}, to refer to the data within individual cells When you enclose the indices in first bracket, it refers to the set of cells.

Cell array indices in smooth parentheses refer to sets of cells.

For example:

```
c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5}; c(1:2,1:2)
```

MATLAB will execute the above statement and return the following result:

```
ans =
   'Red'    'Blue'
   [  1]    [   2]
```

You can also access the contents of cells by indexing with curly braces.

For example:

```
c = {'Red', 'Blue', 'Green', 'Yellow', 'White'; 1 2 3 4 5}; c{1, 2:4}
```

MATLAB will execute the above statement and return the following result:

```
ans =
  Blue
ans =    Green
ans =
  Yellow
```

CHAPTER 11

FUNCTIONS

A function is a group of statements that together perform a task. In MATLAB, functions are defined in separate files. The name of the file and of the function should be the same.

Functions operate on variables within their own workspace, which is also called the **local workspace**, separate from the workspace you access at the MATLAB command prompt which is called the **base workspace**.

Functions can accept more than one input arguments and may return more than one output arguments

Syntax of a function statement is:

function [out1,out2, ..., outN] = myfun(in1,in2,in3, ..., inN)

Example

The following function named *mymax* should be written in a file named *mymax.m*. It takes five numbers as argument and returns the maximum of the numbers.

Create a function file, named mymax.m and type the following code in it:

```
function max = mymax(n1, n2, n3, n4, n5)
%This function calculates the maximum of the
% five numbers given as input
max =  n1;
if(n2 > max)
max = n2;
end
if(n3 > max)
max = n3;
end
if(n4 > max)
max = n4;
end
if(n5 > max)
 max = n5;
end
```

The first line of a function starts with the keyword **function**. It gives the name of the function and order of arguments. In our example, the *mymax* function has five input arguments and one output argument.

The comment lines that come right after the function statement provide the help text. These lines are printed when you type:

help mymax

MATLAB will execute the above statement and return the following result:

This function calculates the maximum of the five numbers given as input

You can call the function as:

mymax(34, 78, 89, 23, 11)

MATLAB will execute the above statement and return the following result:

ans =

89

Anonymous Functions

An anonymous function is like an inline function in traditional programming languages, defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments.

You can define an anonymous function right at the MATLAB command line or within a function or script.

This way you can create simple functions without having to create a file for them.

The syntax for creating an anonymous function from an expression is

f = @(arglist)expression

Example

In this example, we will write an anonymous function named power, which will take two numbers as input and return first number raised to the power of the second number.

Create a script file and type the following code in it:

power = @(x, n) x.^n; result1 = power(7, 3) result2 = power(49, 0.5) result3 = power(10, -10) result4 = power (4.5, 1.5)

When you run the file, it displays:

```
result1 =    343
result2 =      7
result3 =
   1.0000e-10
result4 =
   9.5459
```

Primary and Sub-Functions

Any function other than an anonymous function must be defined within a file. Each function file contains a required primary function that appears first and any number of optional sub-functions that comes after the primary function and used by it.

Primary functions can be called from outside of the file that defines them, either from command line or from other functions, but sub-functions cannot be called from command line or other functions, outside the function file.

Sub-functions are visible only to the primary function and other sub-functions within the function file that defines them.

Example

Let us write a function named quadratic that would calculate the roots of a quadratic equation. The function would take three inputs, the quadratic coefficient, the linear co-efficient and the constant term. It would return the roots.

The function file quadratic.m will contain the primary function *quadratic* and the sub-function *disc*, which calculates the discriminant.

Create a function file *quadratic.m* and type the following code in it:

```
function [x1,x2] = quadratic(a,b,c) %this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficients of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end
% end of quadratic
 function dis = disc(a,b,c)
%function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end
% end of sub-function
```

You can call the above function from command prompt as:

quadratic(2,4,-4)

MATLAB will execute the above statement and will give the following result:

```
ans =
   0.7321
```

Nested Functions

You can define functions within the body of another function. These are called nested functions. A nested function contains any or all of the components of any other function.

Nested functions are defined within the scope of another function and they share access to the containing function's workspace.

A nested function follows the below syntax:

```
function x = A(p1, p2)
...
B(p2)    function y = B(p3)    ...    end ... end
```

Example

Let us rewrite the function *quadratic*, from previous example, however, this time the disc function will be a nested function.

Create a function file *quadratic2.m* and type the following code in it:

```
function [x1,x2] = quadratic2(a,b,c)
function disc  % nested function
d = sqrt(b^2 - 4*a*c);
end % end of function disc
disc;
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end % end of function quadratic2
```

You can call the above function from command prompt as:

```
quadratic2(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
   0.7321
```

Private Functions

A private function is a primary function that is visible only to a limited group of other functions. If you do not want to expose the implementation of a function(s), you can create them as private functions.

Private functions reside in **subfolders** with the special name **private**.

They are visible only to functions in the parent folder.

Example

Let us rewrite the *quadratic* function. This time, however, the *disc* function calculating the discriminant, will be a private function.

Create a subfolder named private in working directory. Store the following function file *disc.m* in it:

```
function dis = disc(a,b,c)  %function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end % end of sub-function
```

Create a function quadratic3.m in your working directory and type the following code in it:

```
function [x1,x2] = quadratic3(a,b,c)
%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficients of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end % end of quadratic3
```

You can call the above function from command prompt as:

```
quadratic3(2,4,-4)
```

MATLAB will execute the above statement and return the following result:

```
ans =
   0.7321
```

Global Variables

Global variables can be shared by more than one function. For this, you need to declare the variable as global in all the functions.

If you want to access that variable from the base workspace, then declare the variable at the command line.

The global declaration must occur before the variable is actually used in a function. It is a good practice to use capital letters for the names of global variables to distinguish them from other variables.

To plot the graph of a function, you need to take the following steps:

Define **x**, by specifying the **range of values** for the variable **x**, for which the function is to be plotted

Define the function, **y = f(x)**

CHAPTER 12

PLOTTING

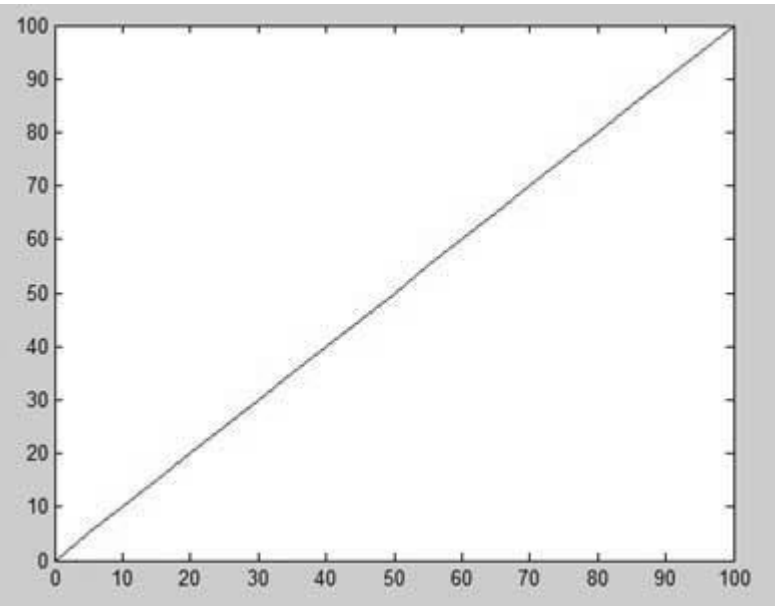
Call the **plot** command, as **plot(x, y)**

Following example would demonstrate the concept. Let us plot the simple function **y = x** for the range of values for x from 0 to 100, with an increment of 5.

Create a script file and type the following code:

```
x = [0:5:100]; y = x; plot(x, y)
```

When you run the file, MATLAB displays the following plot:

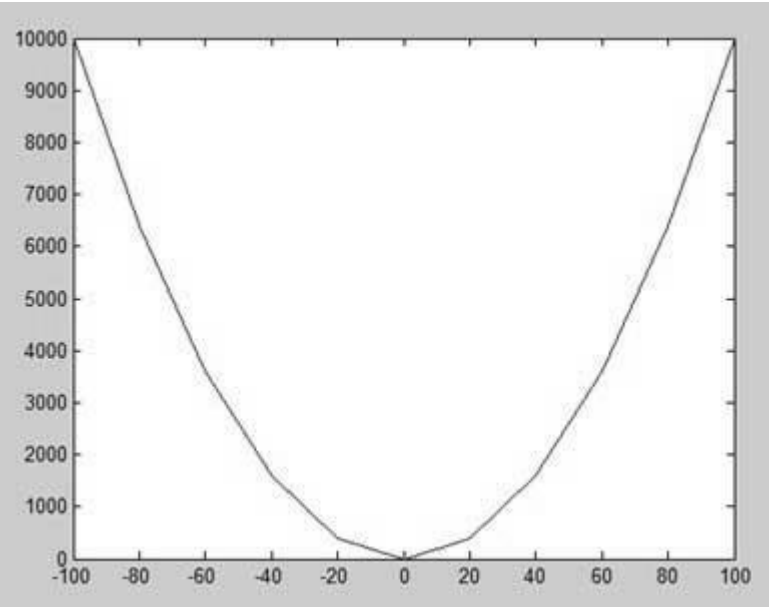


Let us take one more example to plot the function $y = x^2$. In this example, we will draw two graphs with the same function, but in second time, we will reduce the value of increment. Please note that as we decrease the increment, the graph becomes smoother.

Create a script file and type the following code:

```
x = [1 2 3 4 5 6 7 8 9 10]; x = [-100:20:100]; y = x.^2; plot(x, y)
```

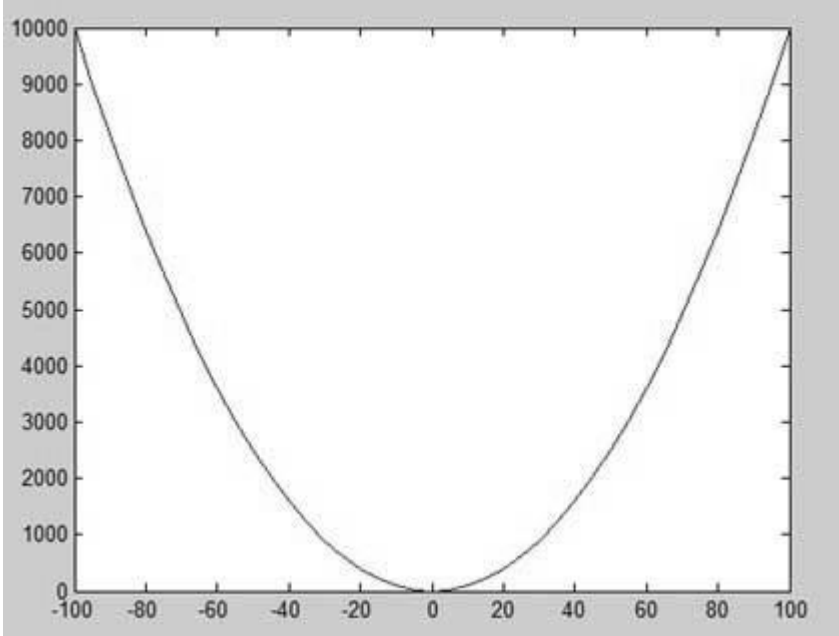
When you run the file, MATLAB displays the following plot:



Change the code file a little, reduce the increment to 5:

```
x = [-100:5:100]; y = x.^2; plot(x, y)
```

MATLAB draws a smoother graph:



Adding Title, Labels, Grid Lines, and Scaling on the Graph

MATLAB allows you to add title, labels along the x-axis and y-axis, grid lines and also to adjust the axes to spruce up the graph.

The **xlabel** and **ylabel** commands generate labels along x-axis and y-axis.

The **title** command allows you to put a title on the graph.

The **grid on** command allows you to put the grid lines on the graph.

The **axis equal** command allows generating the plot with the same scale factors and the spaces on both axes.

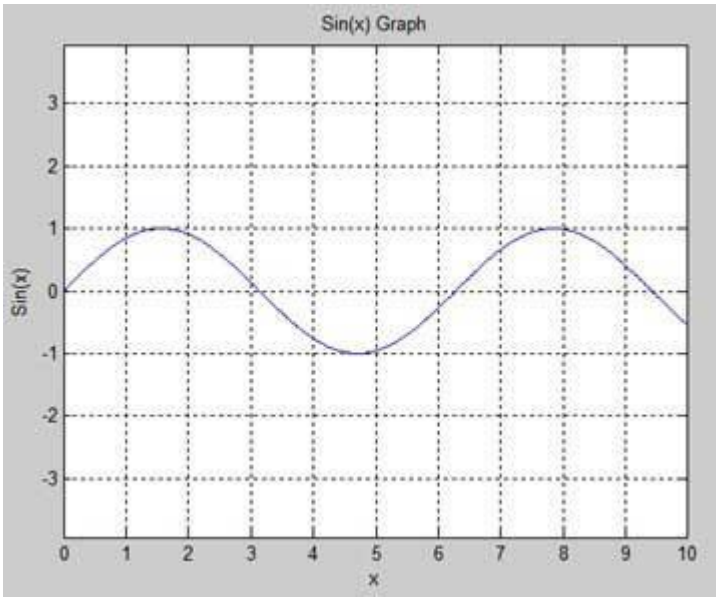
The **axis square** command generates a square plot.

Example

Create a script file and type the following code:

```
x = [0:0.01:10];
y = sin(x);
plot(x, y), xlabel('x'), ylabel('Sin(x)'), title('Sin(x) Graph'), grid on, axis equal
```

MATLAB generates the following graph:



Drawing Multiple Functions on the Same Graph

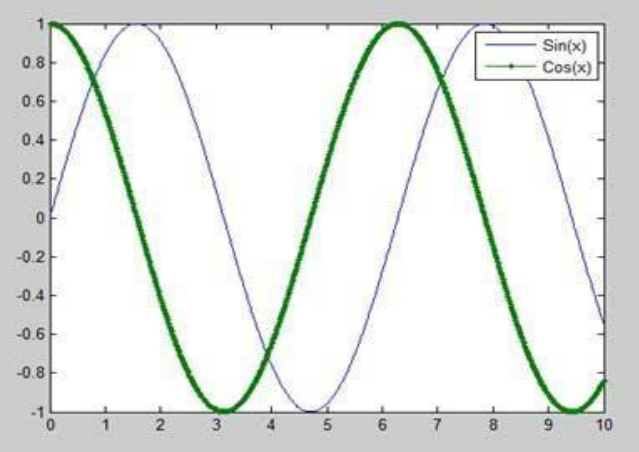
You can draw multiple graphs on the same plot. The following example demonstrates the concept:

Example

Create a script file and type the following code:

```
x = [0 : 0.01: 10];
y = sin(x);
g = cos(x);
plot(x, y, x, g, '.-'), legend('Sin(x)', 'Cos(x)')
```

MATLAB generates the following graph:



Setting Colors on Graph

MATLAB provides eight basic color options for drawing graphs. The following table shows the colors and their codes:

Code	Color
w	White
k	Black
b	Blue
r	Red
c	Cyan
g	Green
m	Magenta
y	Yellow

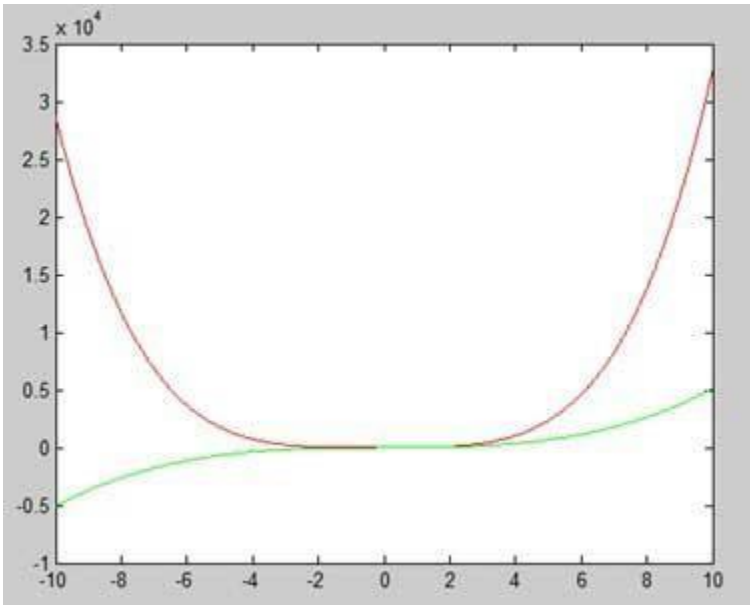
Example

Let us draw the graph of two polynomials $f(x) = 3x^4 + 2x^3 + 7x^2 + 2x + 9$ and $g(x) = 5x^3 + 9x + 2$

Create a script file and type the following code:

```
x = [-10 : 0.01: 10];
y = 3*x.^4 + 2 * x.^3 + 7 * x.^2 + 2 * x + 9;
g = 5 * x.^3 + 9 * x + 2;
plot(x, y, 'r', x, g, 'g')
```

When you run the file, MATLAB generates the following graph:



Setting Axis Scales

The **axis** command allows you to set the axis scales. You can provide minimum and maximum values for x and y axes using the axis command in the following way:

axis ([xmin xmax ymin ymax])

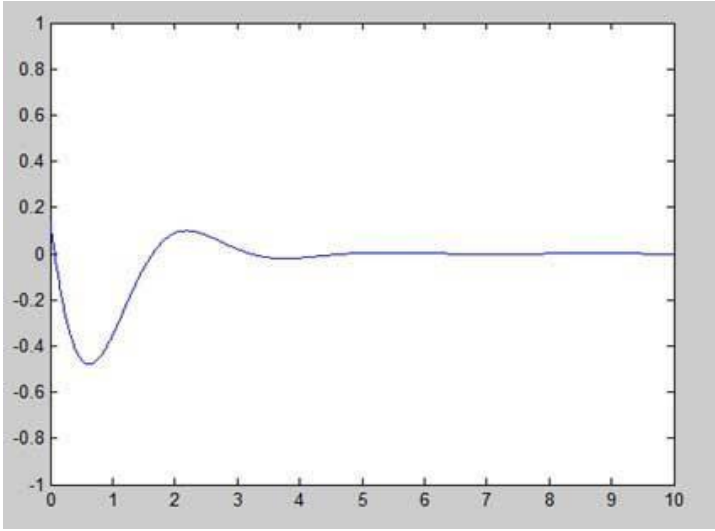
The following example shows this:

Example

Create a script file and type the following code:

```
x = [0 : 0.01: 10];
y = exp(-x).* sin(2*x + 3);
plot(x, y), axis([0 10 -1 1])
```

When you run the file, MATLAB generates the following graph:



Generating Sub-Plots

When you create an array of plots in the same figure, each of these plots is called a subplot. The **subplot** command is used for creating subplots.

Syntax for the command is:

subplot(m, n, p)

where, *m* and *n* are the number of rows and columns of the plot array and *p* specifies where to put a particular plot.

Each plot created with the subplot command can have its own characteristics. Following example demonstrates the concept:

Example

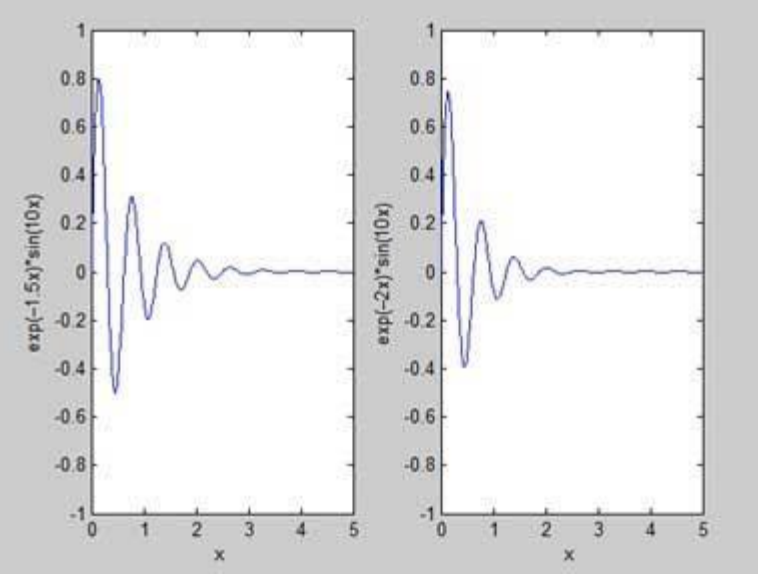
Let us generate two plots:

$y = e^{-1.5x}\sin(10x)$ $y = e^{-2x}\sin(10x)$

Create a script file and type the following code:

```
x = [0:0.01:5];
y = exp(-1.5*x).*sin(10*x);
subplot(1,2,1) plot(x,y), xlabel('x'),ylabel('exp(–1.5x)*sin(10x)'),axis([0 5 -1 1])
y = exp(-2*x).*sin(10*x);
subplot(1,2,2) plot(x,y),xlabel('x'),ylabel('exp(–2x)*sin(10x)'),axis([0 5 -1 1])
```

When you run the file, MATLAB generates the following graph:



CHAPTER 13

GRAPHICS

This chapter will continue exploring the plotting and graphics capabilities of MATLAB. We will discuss:

- Drawing bar charts
- Drawing contours
- Three dimensional plots

Drawing Bar Charts

The **bar** command draws a two dimensional bar chart. Let us take up an example to demonstrate the idea.

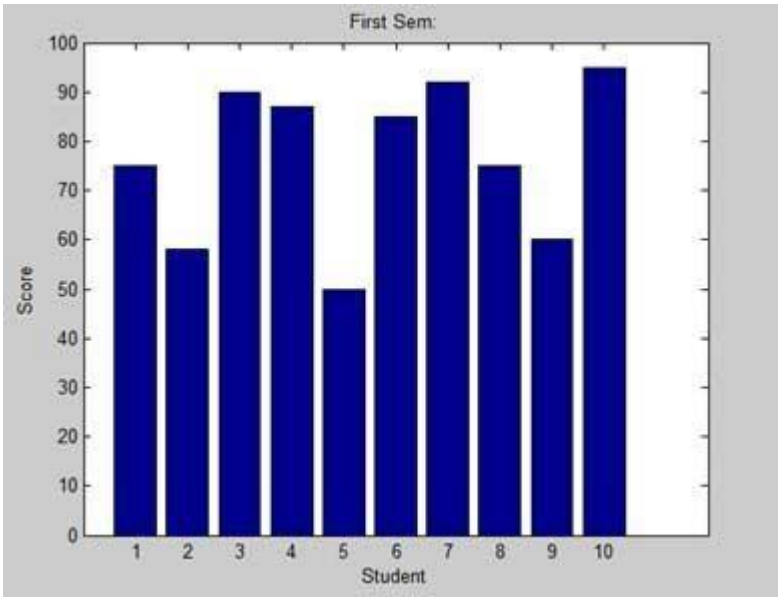
Example

Let us have an imaginary classroom with 10 students. We know the percent of marks obtained by these students are 75, 58, 90, 87, 50, 85, 92, 75, 60 and 95. We will draw the bar chart for this data.

Create a script file and type the following code:

```
x = [1:10];
y = [75, 58, 90, 87, 50, 85, 92, 75, 60, 95];
bar(x,y), xlabel('Student'),ylabel('Score'), title('First Sem:') print -deps graph.eps
```

When you run the file, MATLAB displays the following bar chart:



Drawing Contours

A contour line of a function of two variables is a curve along which the function has a constant value. Contour lines are used for creating contour maps by joining points of equal elevation above a given level, such as mean sea level.

MATLAB provides a **contour** function for drawing contour maps.

Example

Let us generate a contour map that shows the contour lines for a given function g = f(x, y). This function has two variables. So, we will have to generate two independent variables, i.e., two data sets x and y. This is done by calling the **meshgrid** command.

The **meshgrid** command is used for generating a matrix of elements that give the range over x and y along with the specification of increment in each case.

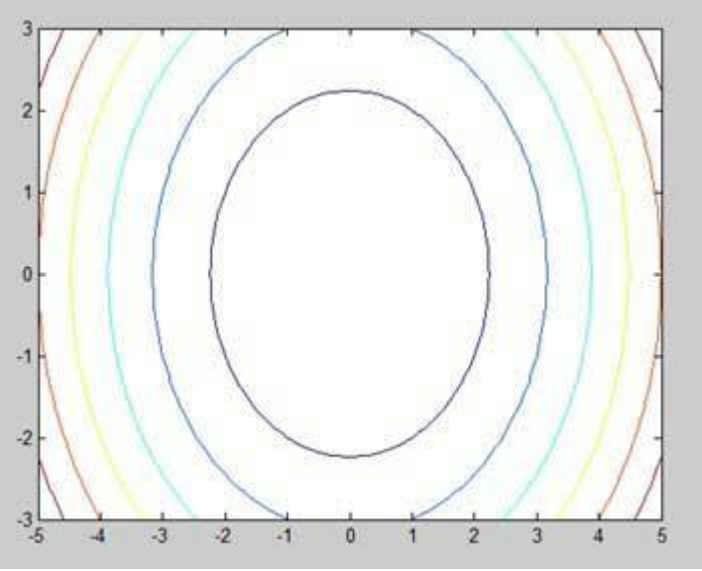
Let us plot our function g = f(x, y), where $-5 \leq x \leq 5$, $-3 \leq y \leq 3$. Let us take an increment of 0.1 for both the values. The variables are set as:

[x,y] = meshgrid(–5:0.1:5, –3:0.1:3);

Lastly, we need to assign the function. Let our function be: $x^2 + y^2$ Create a script file and type the following code:

```
[x,y] = meshgrid(-5:0.1:5,-3:0.1:3); %independent variables
g = x.^2 + y.^2;                     % our function
contour(x,y,g)                       % call the contour function
print -deps graph.eps
```

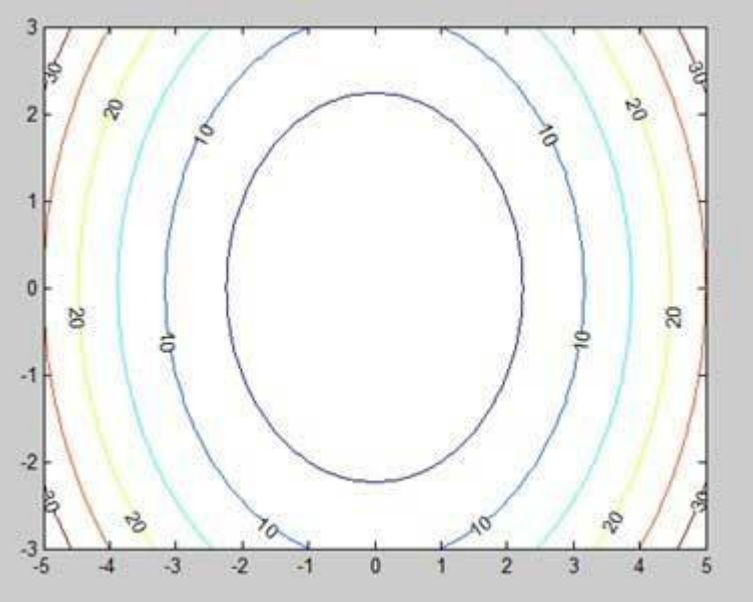
When you run the file, MATLAB displays the following contour map:



Let us modify the code a little to spruce up the map:

```
[x,y] = meshgrid(-5:0.1:5,-3:0.1:3); %independent variables
g = x.^2 + y.^2;                    % our function
[C, h] = contour(x,y,g); % call the contour function
set(h,'ShowText','on','TextStep',get(h,'LevelStep')*2) print -deps graph.eps
```

When you run the file, MATLAB displays the following contour map:



Three-Dimensional Plots

Three-dimensional plots basically display a surface defined by a function in two variables, g = f (x,y). As before, to define g, we first create a set of (x,y) points over the domain of the function using the **meshgrid** command. Next, we assign the function itself. Finally, we use the **surf** command to create a surface plot.

The following example demonstrates the concept:

Example

Let us create a 3D surface map for the function $g = xe^{-(x2 + y2)}$ Create a script file and type the following code:

```
[x,y] = meshgrid(-2:.2:2);
g = x .* exp(-x.^2 - y.^2);
surf(x, y, g) print -deps graph.eps
```