SCIENTIFIC PROGRAMMING (WITH MATLAB)

A COURSE MATERIAL FOR CSC 231

S. O. AKINOLA
Associate Professor,
Computer Science Department,
University of Ibadan

General Introduction and Course Objective

Many of the scientific and engineering problems that were being solved manually in the days past are now being computerized. A computer receives, stores and processes data in a fast and accurate manner. This peculiarity of computer makes it a good choice of tool for solving real life problems today. Scientific and Engineering applications are characterized as those problems which predominantly manipulate numbers and arrays of such numbers using mathematical and statistical principles as basis for their algorithms. These algorithms encompass such problems as statistical significant tests, linear programming, regression analysis and numerical approximations for the solutions of differential and integral equations. Programmers must be well-versed in the mathematical principles underlying the algorithms in order to be able to develop programs for these problems.

In this book, we explore the basic principles of Scientific Programming using MATLAB as a language tool for solving scientific problems. The MATLAB programming language, descended from FORTRAN, has evolved to include many powerful and convenient graphical and analysis tools. It has become an important platform for engineering and science education, as well as research.

MATLAB is a tool for doing numerical computations with matrices and vectors. It is very powerful and easy to use. In fact, it integrates computation, visualization and programming all together in an easy-to-use environment and can be used on almost all the platforms: windows, Unix, and Apple Macintosh, etc.

The MATLAB programming language provides an excellent introductory language, with built-in graphical, mathematical, and user-interface capabilities. The goal is that the student learns to build computational models with graphical user interfaces (GUIs) that enable exploration of model behaviour.

MATLAB language is easier for beginners than many alternatives: it is interpreted rather than compiled; variable types and array sizes need not be declared in advance; it is not strongly typed; vector, matrix, multidimensional array, and complex numbers are basic data types; there is a sophisticated integrated development and debugging environment; and a rich set of mathematical and graphics functions is provided.

Course Curriculum Contents

Introduction to a relevant Scientific programming language. Implementations of basic programming constructs in the language. Procedure calls. Recursion. Extensive laboratory exercises in the programming language. Introduction to Scientific Computation and practical exercises in relevant scientific language like MATLAB or MAPLE.

4 Units, Required

Table of Contents

Study Session 4: MATLAB Operators

Study Session 5: Decision Making and Loop Controls

Study Session 6: MATLAB Vectors and Matrices Manipulations

Study Session 7: MATLAB Arrays

Study Session 8: Further Matrix Functions

Study Session 9: Manipulation of Linear Algebra

Study Session 10: Manipulation of Polynomials

Study Session 11: Introduction to Charts Plotting

Study Session 12: Introduction to Basic Image Processing

Study Session 1: ***Introduction to MATLAB***

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, you will learn about the meaning of MATLAB, the extent of its power in computations especially in Mathematic, Engineering and other scientific applications. The features and uses of MATLAB as well as its environment are discussed.

Learning Outcomes

When you have studied this session, you should be able to explain:

1.1 MATLAB's Power of Computational Mathematics
1.2 Features of MATLAB
1.3 Uses of MATLAB
1.4 MATLAB - Environment Setup
1.5 Understanding the MATLAB Environment
1.6 MATLAB Arithmetic Operators
1.7 MATLAB Logical Operators
1.8 Variable Name in MATLAB

1.1 MATLAB's Power of Computational Mathematics

MATLAB (MATrix LABoratory) is a programming language developed by MathWorks. It started out as a matrix programming language where linear algebra programming was simple. It can be run both under interactive sessions and as a batch job.

MATLAB is a fourth-generation high-level programming language and interactive environment for numerical computation, visualization and programming. 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 in mathematical calculations, generating plots and performing numerical methods. In this unit, you will be introduced to MATLAB. MATLAB is used in every facet of computational mathematics. Following are some commonly used mathematical calculations where it is used.

a. Dealing with Matrices and Arrays
b. 2-D and 3-D Plotting and graphics
c. Linear Algebra
d. Algebraic Equations
e. Non-linear Functions
f. Statistics
g. Data Analysis
h. Calculus and Differential Equations
i. Numerical Calculations
j. Integration
k. Transforms

l. Curve Fitting
m. Various other special functions

1.2 Features of MATLAB

The following are the basic features of MATLAB:

1. MATLAB is a high-level language for numerical computation, visualization and application development.
2. It also provides an interactive environment for iterative exploration, design and problem solving.
3. It provides vast library of mathematical functions for linear algebra, statistics, Fourier analysis, filtering, optimization, numerical integration and solving ordinary differential equations.
4. It provides built-in graphics for visualizing data and tools for creating custom plots.
5. MATLAB's programming interface gives development tools for improving code quality maintainability and maximizing performance.
6. It provides tools for building applications with custom graphical interfaces.
7. It provides functions for integrating MATLAB based algorithms with external applications and languages such as C, Java, .NET and Microsoft Excel.

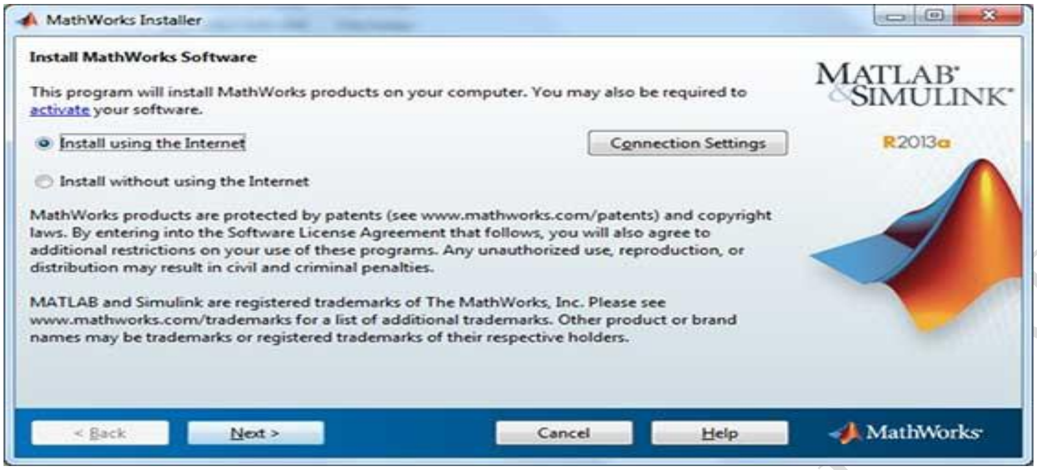
1.3 Uses of MATLAB

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

1. Signal Processing and Communications
2. Image and Video Processing
3. Control Systems
4. Test and Measurement
5. Computational Biology / Bioinformatics
6. Image Processing
7. Computer Vision
8. Signal Processing
9. Communications
10. Mathematics
11. Statistics
12. Optimization
13. Control Systems Design and Analysis
14. Computational Finance
15. Computer Graphics and Simulations, etc

1.4 MATLAB - Environment Setup

Setting up MATLAB environment is a matter of few clicks. The installer can be downloaded from https://www.mathworks.com/downloads/web_downloads. MathWorks provides the licensed product, a trial version and a student version as well. One needs to log into the site and wait a little for their approval. After downloading the installer the software can be installed through few clicks.



MathWorks Installer

Install MathWorks Software

This program will install MathWorks products on your computer. You may also be required to activate your software.

● Install using the Internet

○ Install without using the Internet

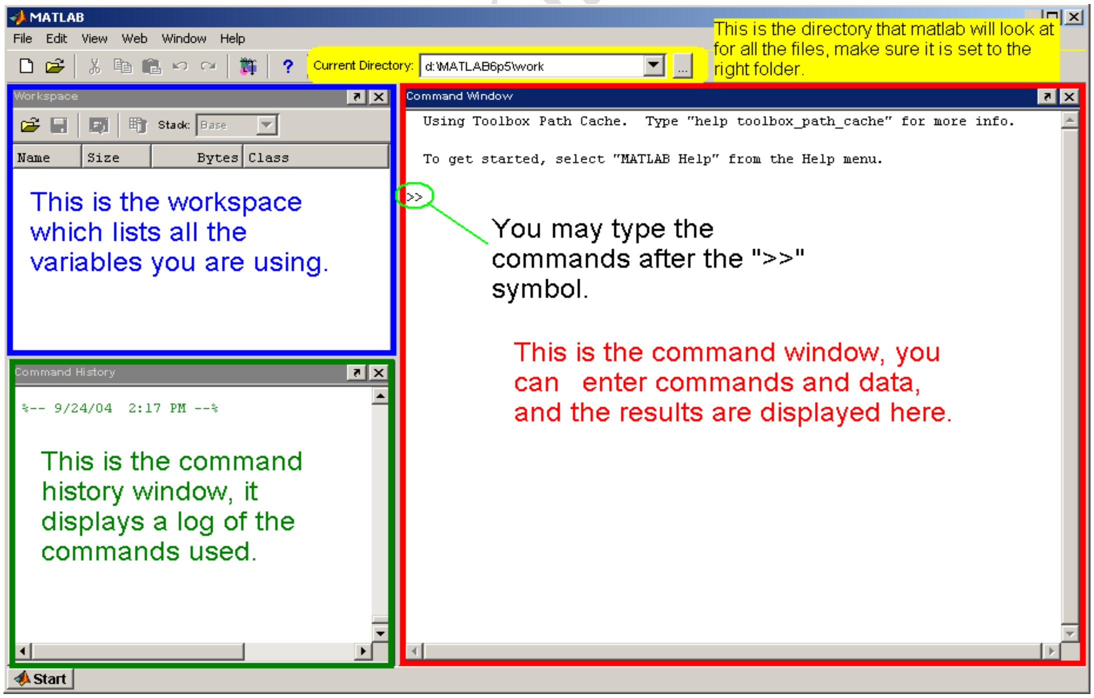
MathWorks products are protected by patents (see www.mathworks.com/patents) and copyright laws. By entering into the Software License Agreement that follows, you will also agree to additional restrictions on your use of these programs. Any unauthorized use, reproduction, or distribution may result in civil and criminal penalties.

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. Please see www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Figure 1.1: Downloading and Installing MATLAB

1.5 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 interface.



This is the directory that matlab will look at for all the files, make sure it is set to the right folder.

This is the workspace which lists all the variables you are using.

```
Using Toolbox Path Cache.  Type "help toolbox_path_cache" for more info.

To get started, select "MATLAB Help" from the Help menu.

>>
```

```
%-- 9/24/04  2:17 PM --%
```

You may type the commands after the ">>" symbol.

This is the command window, you can enter commands and data, and the results are displayed here.

This is the command history window, it displays a log of the commands used.

Figure 1.2: MATLAB Environment

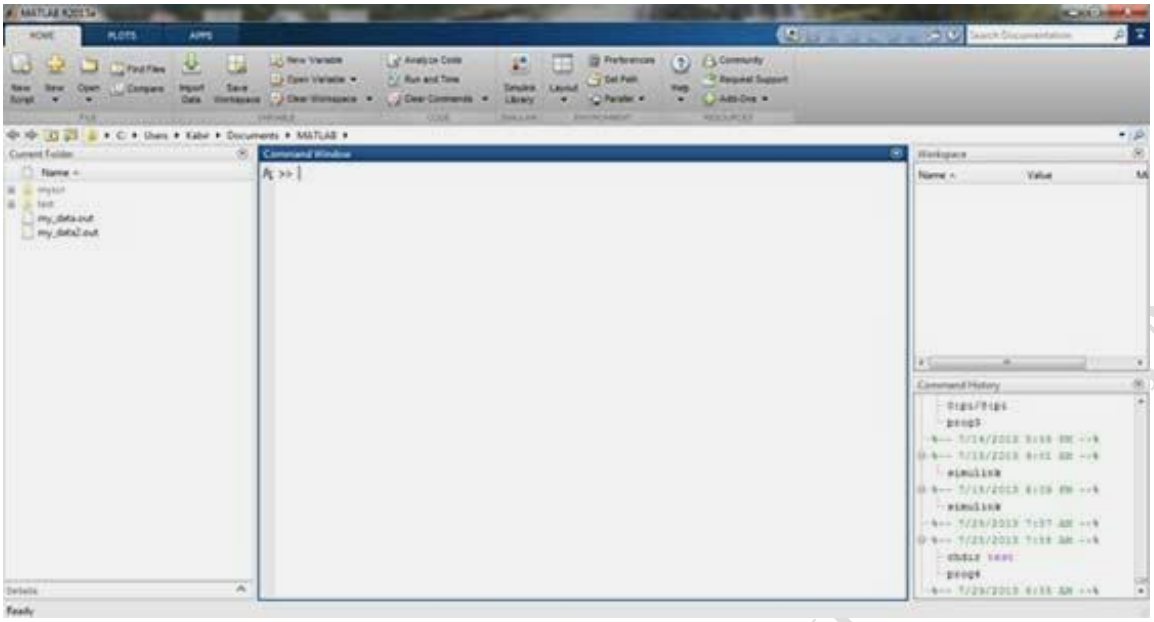


Figure 1.3: MATLAB Working Environment

The desktop has the following panels –

- **Current Folder** − This panel allows us to access the project folders and files.

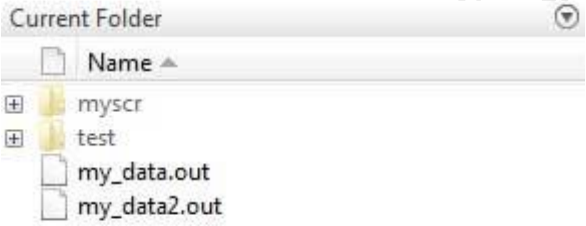
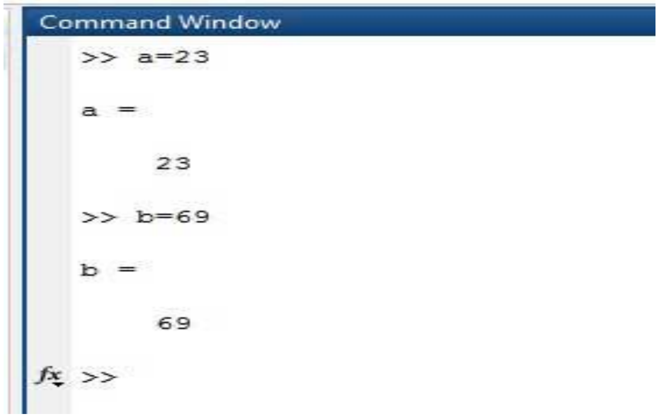


Figure 1.4: MATLAB Current Folder

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

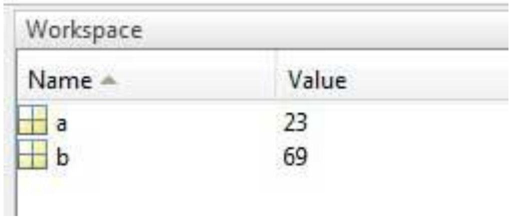


Command Window

```
>> a=23

a =

    23

>> b=69

b =

    69

fx >>
```

Figure 1.5: MATLAB Command Window

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

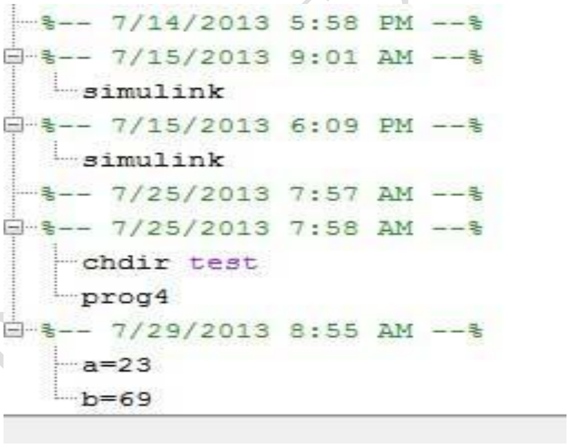


Workspace

Name ▲	Value
a	23
b	69

Figure 1.6: MATLAB Workspace

- **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
```

Figure 1.7: MATLAB Command History

1.6 MATLAB Arithmetic Operators

The following arithmetic operators are allowed in MATLAB

(i) Addition:	+	E.g. 2 + 3
(ii) Subtraction:	-	E.g. 2 - 3
(iii) Multiplication:	*	E.g. 2 * 3
(iv) Division:	/	E.g. 2 / 3
(v) Exponentiation:	^	E.g. 2 ^ 3 (2^3)

1.7 MATLAB Logical Operators

Logical operators are used to compare two variables or arithmetic expressions together. Following are the logical operators supported by MATLAB

(i) Equal:	= =
(ii) Less than:	<
(iii) Greater than:	>
(iv) Not equal:	~ =
(v) Not:	~

1.8 Variable Name in MATLAB

Variable naming rules in MATLAB are

(i) must be unique in the first 63 characters
(ii) must begin with a letter
(iii) may not contain blank spaces or other types of punctuation
(iv) may contain any combination of letters, digits, and underscores
(v) are case-sensitive
(vi) should not use MATLAB keyword

1.9 Pre-defined variable names

- **pi** for 22/7

SUMMARY

In this unit, you have been introduced to MATLAB with respect to its

(i) Power of Computational Mathematics;
(ii) Features;
(iii) Uses;
(iv) Environment Setup and
(v) Environment

Self Assessment Questions (SAQs)

(i) Explain the features and uses of MATLAB in Scientific Computing.
(ii) State four rules of variable naming in MATLAB

Study Session 2: MATLAB - Basic Syntax

Expected Duration: 1 week or 2 contact hours

Introduction

Syntax has to do with grammatical rules of a language; i.e. how statements are constructed and terminated in a language. In this Session, we shall be introduced to how commands or statements are written and executed in a MATLAB environment.

Learning Outcomes

When you have studied this session, you should be able to explain:

2.1 MATLAB’s Command Prompt
2.2 Use of Semicolon (;) in MATLAB
2.3 Adding Comments
2.4 Commonly used Operators and Special Characters
2.5 Special Variables and Constants
2.6 Naming Variables
2.7 Saving Your Work
2.8 MATLAB - Variables
2.9 Multiple Assignments
2.10 Long Assignments
2.11 The format Command
2.12 Creating Vectors
2.13 Creating Matrices
2.14 MATLAB’s Commands

2.1 MATLAB’s Command Prompt

MATLAB environment behaves like a super-complex calculator. We can enter commands at the >> command prompt. MATLAB is an interpreted environment. In other words, we give a command and MATLAB executes it right away.

Practice 1

Type any valid expression, such as:

```
6 + 5
```

Then press ENTER

On clicking the Execute button, or type Ctrl + E, MATLAB executes it immediately and the result returned is −

```
ans = 11
```

Consider more examples:

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

On clicking the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result returned is −

```
ans = 9
```

Note that the % symbol written against a statement signifies the statement is a comment.

Another example,

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

On clicking 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
```

On clicking 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
```

On clicking 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'.

2.2 Use of Semicolon (;) in MATLAB

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

For example,

```
x = 3;

y = x + 5
```

Result:

```
y =  8
```

2.3 Adding Comments

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

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

We can also write a block of comments using the block comment operators % { and % }. For example,

```
%{ This section computes ….
     …………………..
…………………………….   %}
```

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

2.4 Commonly used Operators and Special Characters

MATLAB supports the following commonly used operators and special characters –

Operator	Purpose
+	Plus; addition operator. E.g. 2 + 3
-	Minus; subtraction operator. E.g. 2 - 3
*	Scalar and matrix multiplication operator. E.g. 2 * 3
.*	Array multiplication operator.
^	Scalar and matrix exponentiation operator. E.g. 2 ^ 3
.^	Array exponentiation operator.
\	Left-division operator. E.g. 2 \ 3
/	Right-division operator. E.g. 2 / 3
.\	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.

2.5 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

2.6 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**.

2.7 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

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

```
load myfile
```

2.8 MATLAB - Variables

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

Note that:

- Once a variable is entered into the system, we 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
```

We can use this variable **ans** –

```
sqrt(78);
9876/ans
```

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

```
ans =  1118.2
```

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 =  441.84
```

2.9 Multiple Assignments

We 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
```

What of if you have forgotten the Variables!

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

```
who
```

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

```
your variables are:
a    ans  b    c
```

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:–

```
Attr Name          Size       Bytes     Class
==== ====        ====        ====       =====
   a                   1x1          8       double
   ans                 1x70        757        cell
   b                   1x1          8       double
   c                   1x1          8       double

Total is 73 elements using 781 bytes
```

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
```

2.10 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
```

2.11 The format Command

By default, MATLAB displays numbers with four decimal place values. This is known as **short format**. However, if we want more precision, we 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 statement and return the following result−

```
x = 17.2319816406394
```

Another example,

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

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

```
x = 17.232
```

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
4.678 * 4.9
```

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

```
ans = 34177/1491
```

2.12 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 =
   7    8    9   10   11
```

Another example, corresponding elements are added together in this case

```
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 =
        9         11         13         15         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
```

2.13 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
```

2.14 MATLAB - Commands

MATLAB is an interactive program for numerical computation and data visualization. We 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.

2.14.1 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).

2.14.2 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.

2.14.3 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.

2.14.4 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.

2.14.5 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 x-y 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 subwindows.
text	Places string in figure.
bar	Creates bar chart.
loglog	Creates log-log plot.
polar	Creates polar plot.
semilogx	Creates semilog plot. (logarithmic abscissa).
semilogy	Creates semilog plot. (logarithmic ordinate).
stairs	Creates stairs plot.
stem	Creates stem plot.

Summary

In this Session, you have been introduced to the following MATLAB's basic syntaxes:

(i) MATLAB's Command Prompt
(ii) Use of Semicolon (;) in MATLAB
(iii) Adding Comments
(iv) Commonly used Operators and Special Characters
(v) Special Variables and Constants
(vi) Naming Variables
(vii) Saving Your Work
(viii) MATLAB – Variables
(ix) Multiple Assignments
(x) Long Assignments
(xi) The format Command
(xii) Creating Vectors
(xiii) Creating Matrices
(xiv) MATLAB Commands

Self Assessment Questions (SAQs)

Execute the following statements in a MATLAB's environment.

1. $Y = A^m – (B \text{ x } C)^t$ Assign arbitrary values to the variables
2. Write MATLAB statement to create any arbitrary 4 x 4 square matrix

Study Session 3: MATLAB M-Files and Data Types

Expected Duration: 1 week or 2 contact hours

Introduction

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

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.

In this Session, we shall study the creation and use of M-Files as well as the data types that exist in MATLAB.

Learning Outcomes

When you have studied this session, you should be able to explain:

3.1 The M Files
3.2 Creating and Running Script File
3.3 MATLAB - Data Types
3.4 MATLAB – Operators
3.5 MATLAB - Decision Making
3.6 MATLAB - Loop Types
3.7 Loop Control Statements

3.1 The M Files

MATLAB allows writing two kinds of program files –

- **Scripts** − script files are program files with **.m extension**. In these files, we write series of commands, which we 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.

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

3.2 Creating and Running Script File

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

- Using the command prompt
- Using the IDE

If we are using the command prompt, type **edit** in the command prompt. This will open the editor. We 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 we want to store all program files in a specific folder, then we 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 we are creating the file for first time, MATLAB prompts us to confirm it. Click Yes
.

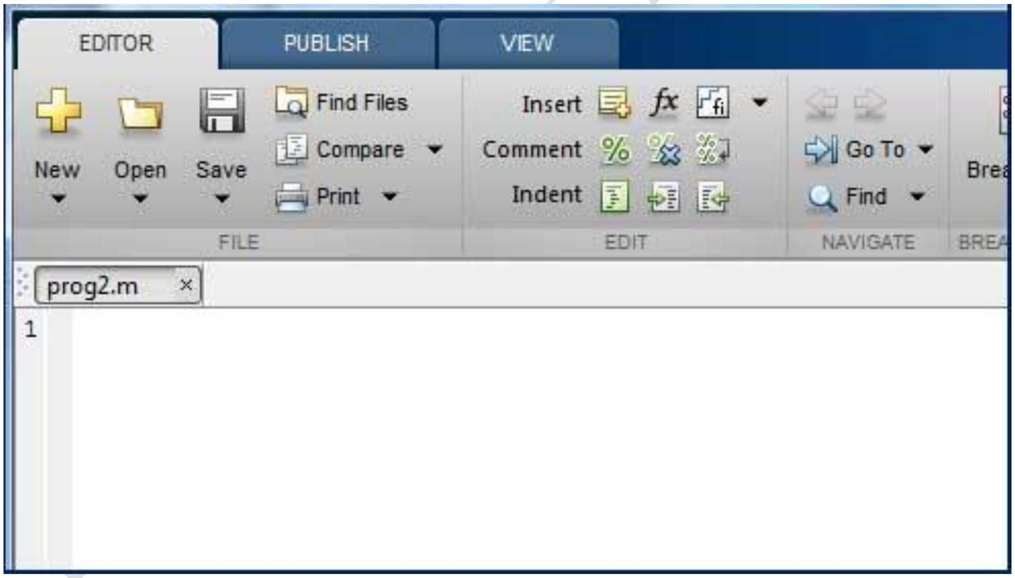


Figure 3.1 Editor Environment

Alternatively, if we are using the IDE, choose NEW -> Script. This also opens the editor and creates a file named Untitled. We 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, we 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 = 67120/5303
e = 23099/365
f = 29/9104528
```

3.3 MATLAB - Data Types

3.3.1 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 = 5678.9
c =  5679
```

3.3.2 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

3.3.3 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 we run the file, it produces the following result –

```
x = 3
ans = 0
ans = 1
ans = 1
ans = 1
ans = 1
x = 1177/50
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
```

Summary

In this Unit, you have been introduced to the following concepts:

(i) How to create and use the M Files
(ii) Creating and Running Script File
(iii) MATLAB - Data Types

Self-Assessment Questions (SAQs)

(i) Create an m-file to compute the Interest on Principal, given the rate and Time (I = PRT/100)
(ii) Create an m-file to compute the roots of any quadratic equation using the formula method

Study Session 4: *MATLAB Operators*

Expected Duration: 1 week or 2 contact hours

Introduction

Operators are symbols used in programming to perform some mathematical or logical operations. In this Session, you shall be introduced to the different mathematical and logical operators the MATLAB supports.

Learning Outcomes

When you have studied this session, you should be able to explain and use the following operators:

4.1 MATLAB – Operators
4.2 Arithmetic Operators
4.2 Relational Operators
4.3 Logical Operators
4.4 Bitwise Operations
4.5 Set Operations

4.1 MATLAB – 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 non-scalar data. MATLAB allows the following types of elementary operations –

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

4.2 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.

4.2 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

4.3 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.

4.4 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

4.5 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 is 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

Summary

In this Session, you have been introduced to the different operators that can be used for computations in MATLAB, such as

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators and
- Set Operations

Self-Assessment Questions (SAQs)

Open your MATLAB environment to practice all the operators as explained in this Session using your own supplied data

Study Session 5: Decision Making and Loop Controls

Expected Duration: 1 week or 2 contact hours

Introduction

Sometimes you may need to make some decisions in your program, like if the value of A is less than B, then C should be computed. In another way, you want to repeat a section of your code based on certain conditions or for some number of times. The means to achieve these your goals is explained in this Session

Learning Outcomes

When you have studied this session, you should be use the following controls in MATLAB:

5.1 MATLAB - Decision Making
5.2 MATLAB - Loop Types
5.3 Loop Control Statements

5.1 MATLAB - 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 –

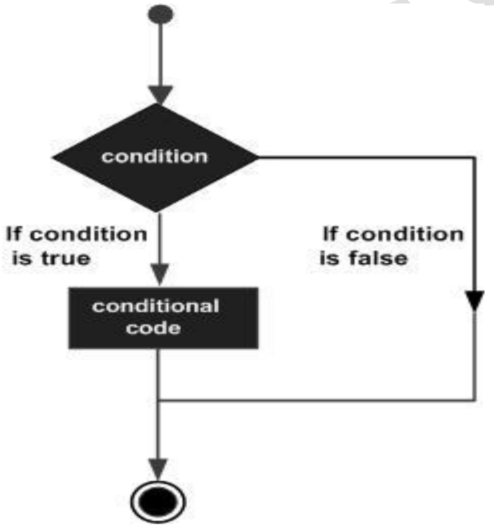


Figure 5.1: Decision-making Structure

MATLAB provides following types of decision making statements.

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 … else…. end statement	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 statement	We 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 statement	We can use one **switch** statement inside another **switch** statement(s).

5.1.1 Explanations and Illustrations

(i) Simple decisions

If statements:

The general form of the IF statement is

```
IF expression
        statements
ELSEIF expression
        statements
ELSE
        statements
END
```

IF … END (as in Example 1 below)

Example 1:

```
n = input('Enter the upper limit: ');
if n < 1
        disp ('Your answer is meaningless!')
end
x = 1:n;
```

```
term = sqrt(x);
y = sum(term)
```

Complex Decisions

Use IF … ELSEIF … ELSE ... END

Example 2:

```
if (x == 3)
    disp('The value of x is 3.');
  elseif (x == 5)
    disp('The value of x is 5.');
  else
    disp('The value of x is not 3 or 5.');
  end;
```

Example 3:

Roots of $ax^2+bx+c=0$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$\Delta = b^2 - 4ac$$

The roots are set by the discriminant

If $\Delta < 0$ (no real roots)
If $\Delta = 0$ (one real root)
If $\Delta > 0$ (two real roots)

MATLAB needs to make decisions (based on Δ).

A possible M-file (explained in Session 3) will look like:

```
Read in values of a, b, c
Calculate $\Delta$
IF $\Delta < 0$
   Print message ' No real roots'→ Go END
ELSEIF $\Delta = 0$
     Print message 'One real root'→ Go END
ELSE
     Print message 'Two real roots'
END
```

OurMFile Code:

```
%============================================================
%   Demonstration of an m-file
%   Calculate the real roots of a quadratic equation
%============================================================
clear all;      % clear all variables
clc;            % clear screen
coeffts = input('Enter values for a,b,c (as a vector):  ');    % Read in equation coefficients
a = coeffts(1);
b = coeffts(2);
c = coeffts(3);
delta = b^2 - 4*a*c;    % Calculate discriminant
% Calculate number (and value) of real roots
if delta < 0
   fprintf('\nEquation has no real roots:\n\n')
   disp(['discriminant = ', num2str(delta)])
elseif delta == 0
   fprintf('\nEquation has one real root:\n')
   xone = -b/(2*a)
else
   fprintf('\nEquation has two real roots:\n')
   x(1) = (-b + sqrt(delta))/(2*a);
   x(2) = (-b – sqrt(delta))/(2*a);
   fprintf('\n First root = %10.2e\n\t Second root = %10.2f', x(1),x(2))
end
```

Switch statement

SWITCH – Switch among several cases based on expression. The general form of SWITCH statement is:

```
SWITCH switch_expr
         CASE case_expr,
                          statement, …, statement
         CASE {case_expr1, case_expr2, case_expr3, …}
                          statement, …, statement
         …
         OTHERWISE
                          statement, …, statement
END
```

Note:

(i) Only the statements between the matching CASE and the next CASE, OTHERWISE, or END are executed

(ii) Unlike C, the SWITCH statement does not fall through (so BREAKs are unnecessary)

Example:

```
switch face
    case {1}
      disp('Rolled a 1');
    case {2}
      disp('Rolled a 2');
    otherwise
      disp('Rolled a number >= 3');
    end
```

5.2 MATLAB - Loop Types

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:

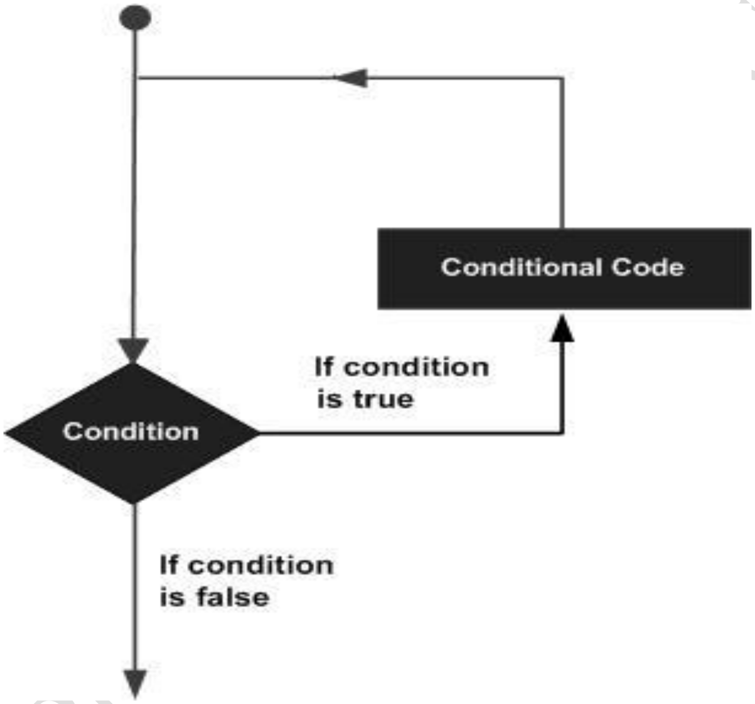


Figure 5.2: Loop Control

MATLAB provides following types of loops to handle looping requirements.

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	We can use one or more loops inside any another loop.

5.2.1 Examples and illustrations

For loops

FOR repeats statements a specific number of times. The general form of a FOR statement is:

```
FOR variable=expr
          statements
END
```

```
>> x = 0;
   for i=1:2:5       % start at 1, increment by 2
     x = x+i;         % end with 5.
   end
```

This computes x = 0+1+3+5=9.

While loops

WHILE repeats statements an indefinite number of times. The general form of a WHILE statement is:

```
WHILE expression
    statements
END
```

```
x=7;
   while (x > = 0)
     x = x-2;
   end;
```

This computes x = 7-2-2-2-2 = -1

5.3 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.

Control Statement	Description
break statement	Terminates the **loop** statement and transfers execution to the statement immediately following the loop. break terminates execution of for and while loops. For nested loops, it exits the innermost loop only
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Example:

```
x=7;
  while (x > = 0)
    x = x-2;
    if  x == 5 continue;
  end;
```

Guess what the solution will be.

```
x=7;
  while (x > = 0)
    x = x-2;
    if  x == 5 break;
  end;
```

Again guess what the solution will be.

Summary

In this Session, you have been introduced to the different control structures that MATLAB supports. These are Decision, Loops and Loop Control structures

Self-Assessment Questions (SAQ)

(1) Computing the area of a triangle using Heron’s formula

Problem Statement

Given a triangle with side lengths **a**, **b** and **c**, its area can be computed using the Heron's formula:

$$\text{Area} = \sqrt{s*(s-a)*(s-b)*s-c)}$$

where **s** is the half of the perimeter length:

$$s = \frac{1}{2}(a+b+c)$$

Write a program to read in the coefficients **a**, **b** and **c**, and compute the area of the triangle. However, not any three numbers can make a triangle. There are two conditions. First, all side lengths must be positive:

$$a > 0,\ b > 0 \text{ and } c > 0$$

and second the sum of any two side lengths must be greater than the third side length:

$$a+b > c,\ a+c > b \text{ and } b+c > a$$

In the program, these two conditions must be checked before computing the triangle area; otherwise, square root computation will be in trouble.

(2) A seller agrees to give the following discounts to his customers based on the number of items bought by them.

Quantity	Discount
1 – 5	2%
6 – 10	5%
11 – 20	10%
>=21	15%

Assuming that the seller deals with only one product at a unit price P, write a program that will compute the initial pay before discount, the discount amount and the final payable amount after discount for the seller.

Study Session 6: *MATLAB Vectors and Matrices Manipulations*

Expected Duration: 1 week or 2 contact hours

Introduction

A vector is a one-dimensional array of numbers. MATLAB allows creating two types of vectors − Row vectors and Column vectors. A matrix is a two-dimensional array of numbers. In MATLAB, We create a matrix by entering elements in each row as comma or space delimited numbers and using semicolons to mark the end of each row. We shall look at these two concepts in this Session.

Learning Outcomes

When you have studied this session, you should be able to explain:

6.1 Types of MATLAB - Vectors
- 6.1.1 Row Vectors
- 6.1.2 Column Vectors
- 6.1.3 Referencing the Elements of a Vector
- 6.1.4 Vector Operations

6.2 MATLAB - Matrix
- 6.2.1 Referencing the Elements of a Matrix
- 6.2.2 Deleting a Row or a Column in a Matrix
- 6.2.3 Matrix Operations

6.1 Types of MATLAB - Vectors

6.1.1 Row Vectors

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

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

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

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

6.1.2 Column Vectors

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
```

6.1.3 Referencing the Elements of a Vector

We can reference one or more of the elements of a vector in several ways. The ith component of a vector v is referred as v(i). For example –

```
v = [ 1; 2; 3; 4; 5; 6];            % creating a column vector of 6 elements
v(3)
```

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

```
ans =  3
```

When we reference a vector with a colon, such as v(:), all the components of the vector are listed.

```
v = [ 1; 2; 3; 4; 5; 6];            % creating a column vector of 6 elements
v(:)
```

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

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

MATLAB allows us to select a range of elements from a vector. For example, let us create a row vector *rv* of 9 elements, then we will reference the elements 3 to 7 by writing ***rv(3:7)*** and create a new vector named *sub_rv*.

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

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

```
sub_rv =

   3   4   5   6   7
```

6.1.4 Vector Operations

6.1.4.1 MATLAB - Addition & Subtraction of Vectors

We can add or subtract two vectors. Both the operand vectors must be of same type and have same number of elements.

Example:
Create a script file with the following code:

```
A = [7, 11, 15, 23, 9];
B = [2, 5, 13, 16, 20];
C = A + B;
D = A - B;
disp(C);
disp(D);
```

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

```
9    16    28    39    29
5     6     2     7   -11
```

6.1.4.2 MATLAB - Scalar Multiplication of Vectors

When we multiply a vector by a number, this is called the **scalar multiplication**. Scalar multiplication produces a new vector of same type with each element of the original vector multiplied by the number. **Example:**

Create a script file with the following code –

```
v = [ 12 34 10 8];
m = 5 * v
```

When we run the file, it displays the following result –

```
m =
   60   170    50    40
```

Please note that we can perform all scalar operations on vectors. For example, we can add, subtract and divide a vector with a scalar quantity.

6.1.4.3 MATLAB - Transpose of a Vector

The transpose operation changes a column vector into a row vector and vice versa. The transpose operation is represented by a single quote (**'**). **Example:**

Create a script file with the following code –

```
r = [ 1 2 3 4 ];
tr = r';
v = [1;2;3;4];
tv = v';
disp(tr); disp(tv);
```

When we run the file, it displays the following result –

```
    1
    2
    3
    4

    1     2     3     4
```

6.1.4.4 MATLAB - Appending Vectors

MATLAB allows us to append vectors together to create new vectors. If we have two row vectors r1 and r2 with n and m number of elements, to create a row vector r of n plus m elements, by appending these vectors, we write –

```
r = [r1, r2]
```

We can also create a matrix r by appending these two vectors, the vector r2, will be the second row of the matrix –

```
r = [r1; r2]
```

However, to do this, both the vectors should have same number of elements.

Similarly, we can append two column vectors c1 and c2 with n and m number of elements. To create a column vector c of n plus m elements, by appending these vectors, we write –

```
c = [c1; c2]
```

We can also create a matrix c by appending these two vectors; the vector c2 will be the second column of the matrix –

```
c = [c1, c2]
```

However, to do this, both the vectors should have same number of elements.

Example:

Create a script file with the following code:

```
r1 = [ 1 2 3 4 ];
r2 = [5 6 7 8 ];
r = [r1,r2]
rMat = [r1;r2]

c1 = [ 1; 2; 3; 4 ];
c2 = [5; 6; 7; 8 ];
c = [c1; c2]
cMat = [c1,c2]
```

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

```
r =
 Columns 1 through 7:

     1     2     3     4     5     6     7

 Column 8:

     8

rMat =

     1     2     3     4
     5     6     7     8

c =
     1
     2
     3
     4
     5
     6
     7
     8

cMat =

     1     5
     2     6
     3     7
     4     8
```

6.1.4.5 MATLAB - Magnitude of a Vector

Magnitude of a vector v with elements v1, v2, v3, …, vn, is given by the equation:
$|v| = \surd(v1^2 + v2^2 + v3^2 + \ldots + vn^2)$

We need to take the following steps to calculate the magnitude of a vector:

- Take the product of the vector with itself, using **array multiplication** (.*). This produces a vector sv, whose elements are squares of the elements of vector v.
  sv = v**.***v;

- Use the sum function to get the **sum** of squares of elements of vector v. This is also called the dot product of vector v.
  dp= sum(sv);

- Use the **sqrt** function to get the square root of the sum which is also the magnitude of the vector v.
  mag = sqrt(s);

Example:

Create a script file with the following code –

```
v = [1: 2: 20];
sv = v.* v;             %the vector with elements
                        % as square of v's elements
dp = sum(sv);            % sum of squares -- the dot product
mag = sqrt(dp);         % magnitude
disp('Magnitude:'); disp(mag);
```

When we run the file, it displays the following result –

```
Magnitude:
76877/2108
```

6.1.4.6 MATLAB - Vector Dot Product

Dot product of two vectors a = (a1, a2, …, an) and b = (b1, b2, …, bn) is given by:
$a.b = \sum(ai.bi)$

Dot product of two vectors a and b is calculated using the **dot** function.

```
dot(a, b);
```

Example

Create a script file with the following code:

```
v1 = [2 3 4];
v2 = [1 2 3];
dp = dot(v1, v2);
disp('Dot Product:'); disp(dp);
```

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

```
Dot Product:
   20
```

6.1.4.7 Vectors with Uniformly Spaced Elements

MATLAB allows us to create a vector with uniformly spaced elements. To create a vector v with the first element f, last element l, and the difference between elements is any real number n, we write:

```
v = [f : n : l]
```

Example: Create a script file with the following code –

```
v = [1: 2: 20];
sqv = v.^2;
disp(v);disp(sqv);
```

When we run the file, it displays the following result –

```
Columns 1 through 7:

        1         3         5         7         9        11        13

 Columns 8 through 10:

       15        17        19
 Columns 1 through 7:

        1         9        25        49        81       121       169

 Columns 8 through 10:

      225       289       361
```

6.2 MATLAB - Matrix

A matrix is a two-dimensional array of numbers. In MATLAB, We create a matrix by entering elements in each row as comma or space delimited numbers and using semicolons to mark the end of each row. For example, let us create a 4-by-5 matrix *a*:

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

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

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

6.2.1 Referencing the Elements of a Matrix

To reference an element in the mth row and nth column, of a matrix *mx*, we write:

```
mx(m, n);
```

For example, to refer to the element in the 2nd row and 5th column, of the matrix *a*, as created in the last section, we type –

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

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

```
ans =  6
```

To reference all the elements in the mth column we type A(:,m). Let us create a column vector v, from the elements of the 4th row of the matrix a:

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

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

```
v =
    4
    5
    6
    7
```

We can also select the elements in the m^{th} through n^{th} columns, for this we write:

```
a(:,m:n)
```

Let us create a smaller matrix taking the elements from the second and third columns:

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

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

```
ans =
    2    3
    3    4
    4    5
    5    6
```

In the same way, we can create a sub-matrix taking a sub-part of a matrix.

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

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

```
ans =
    2    3
    3    4
    4    5
    5    6
```

In the same way, we can create a sub-matrix taking a sub-part of a matrix. For example, let us create a sub-matrix *sa* taking the inner subpart of a:

```
3    4    5
4    5    6
```

To do this, write:

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

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

```
sa =
    3    4    5
    4    5    6
```

6.2.2 Deleting a Row or a Column in a Matrix

We can delete an entire row or column of a matrix by assigning an empty set of square braces [ ] to that row or column. Basically, [ ] denotes an empty array.

For example, let us delete the fourth row of a:

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

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

```
a =
    1    2    3    4    5
    2    3    4    5    6
    3    4    5    6    7
```

Next, let us delete the fifth column of a:

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

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

```
a =
    1    2    3    4
    2    3    4    5
    3    4    5    6
    4    5    6    7
```

In this example, let us create a 3-by-3 matrix m, then we will copy the second and third rows of this matrix twice to create a 4-by-3 matrix.

Create a script file with the following code:

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

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

```
new_mat =
    4    5    6
    7    8    9
    4    5    6
    7    8    9
```

6.2.3 Matrix Operations

6.2.3.1 MATLAB - Addition & Subtraction of Matrices

We can add or subtract matrices. Both the operand matrices must have the same number of rows and columns. Create a script file with the following code –

```
a = [ 1 2 3 ; 4 5 6; 7 8 9];
b = [ 7 5 6 ; 2 0 8; 5 7 1];
c = a + b
d = a - b
```

When we run the file, it displays the following result –

```
c =
    8     7     9
    6     5    14
   12    15    10
d =
   -6    -3    -3
    2     5    -2
    2     1     8
```

6.2.3.2 MATLAB - Matrix Multiplication and Division

Consider two matrices A and B. If A is an m x n matrix and B is an n x p matrix, they could be multiplied together to produce an m x n matrix C. Matrix multiplication is possible only if the number of columns n in A is equal to the number of rows n in B.

In matrix multiplication, the elements of the rows in the first matrix are multiplied with corresponding columns in the second matrix. Each element in the (i, j)th position, in the resulting matrix C, is the summation of the products of elements in ith row of first matrix with the corresponding element in the jth column of the second matrix. Matrix multiplication in MATLAB is performed by using the * operator.

Example:
Create a script file with the following code:

```
a = [ 1 2 3; 2 3 4; 1 2 5]
b = [ 2 1 3 ; 5 0 -2; 2 3 -1]
prod = a * b
```

When we run the file, it displays the following result –

```
a =
    1    2    3
    2    3    4
    1    2    5
b =
    2    1    3
    5    0   -2
    2    3   -1
prod =
   18   10   -4
   27   14   -4
   22   16   -6
```

We can divide two matrices using left (\) or right (/) division operators. Both the operand matrices must have the same number of rows and columns. **Example:**

Create a script file with the following code –

```
a = [ 1 2 3 ; 4 5 6; 7 8 9];
b = [ 7 5 6 ; 2 0 8; 5 7 1];
c = a / b
d = a \ b
```

When we run the file, it displays the following result –

```
c =
  -0.52542   0.68644   0.66102
  -0.42373   0.94068   1.01695
  -0.32203   1.19492   1.37288

d =

  -3.27778  -1.05556  -4.86111
  -0.11111   0.11111  -0.27778
   3.05556   1.27778   4.30556
```

6.2.3.3 MATLAB - Transpose of a Matrix

The transpose operation switches the rows and columns in a matrix. It is represented by a single quote ('). **Example:**

Create a script file with the following code –

```
a = [ 10 12 23 ; 14 8 6; 27 8 9]
b = a'
```

When we run the file, it displays the following result –

```
a =
    10    12    23
    14     8     6
    27     8     9

b =
    10    14    27
    12     8     8
    23     6     9
```

6.2.3.4 MATLAB - Concatenating Matrices

We can concatenate two matrices to create a larger matrix. The pair of square brackets '[ ]' is the concatenation operator. MATLAB allows two types of concatenations −

- Horizontal concatenation
- Vertical concatenation

When we concatenate two matrices by separating those using commas, they are just appended horizontally. It is called horizontal concatenation. Alternatively, if We concatenate two matrices by separating those using semicolons, they are appended vertically. It is called vertical concatenation. **Example:**

Create a script file with the following code –

```
a = [ 10 12 23 ; 14 8 6; 27 8 9]
b = [ 12 31 45 ; 8 0 -9; 45 2 11]
c = [a, b]
d = [a; b]
```

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

```
a =
    10    12    23
    14     8     6
    27     8     9
b =
    12    31    45
     8     0    -9
    45     2    11

c =
    10    12    23    12    31    45
    14     8     6     8     0    -9
    27     8     9    45     2    11
```

```
d =
   10    12    23
   14     8     6
   27     8     9
   12    31    45
    8     0    -9
   45     2    11
```

6.2.3.4 MATLAB - Determinant of a Matrix

Determinant of a matrix is calculated using the **det** function of MATLAB. Determinant of a matrix A is given by det(A). **Example:**

Create a script file with the following code:

```
a = [ 1 2 3; 2 3 4; 1 2 5]
det(a)
```

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

```
a =
    1     2     3
    2     3     4
    1     2     5

ans = -2
```

6.2.3.5 MATLAB - Inverse of a Matrix

The inverse of a matrix A is denoted by A^{-1} such that the following relationship holds –

$AA^{-1} = A^{-1}A = 1$

The inverse of a matrix does not always exist. If the determinant of the matrix is zero, then the inverse does not exist and the matrix is singular.
Inverse of a matrix in MATLAB is calculated using the **inv** function. Inverse of a matrix A is given by inv(A). **Example:**

Create a script file and type the following code –

```
a = [ 1 2 3; 2 3 4; 1 2 5]
inv(a)
```

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

```
a =
     1     2     3
     2     3     4
     1     2     5

ans =
   -3.5000    2.0000    0.5000
    3.0000   -1.0000   -1.0000
   -0.5000         0    0.5000
```

Summary

Self-Assessment Questions

Study Session 7: MATLAB Arrays

Expected Duration: 1 week or 2 contact hours

Introduction

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 section, we will discuss multidimensional arrays. However, before that, let us discuss some special types of arrays

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

7.1 Special Arrays in MATLAB
7.2 Multidimensional Arrays
7.3. Array Functions
7.4 Other Square Array/Matrix Functions

7.1 Special Arrays in MATLAB

Below are the special arrays that can be created and used in MATLAB:

Function	**Description**
accumarray	Distribute elements of an input matrix to specified locations in an output matrix, also allowing for accumulation.
diag	Create a diagonal matrix from a vector.
eye	Create a matrix with ones on the diagonal and zeros elsewhere.
magic	Create a square matrix with rows, columns, and diagonals that add up to the same number.
ones	Create a matrix or array of all ones.
rand	Create a matrix or array of uniformly distributed random numbers.
randn	Create a matrix or array of normally distributed random numbers and arrays.
randperm	Create a vector (1-by-n matrix) containing a random permutation of the specified integers.
zeros	Create a matrix or array of all zeros.
accumarray	Distribute elements of an input matrix to specified locations in an output matrix, also allowing for accumulation.
diag	Create a diagonal matrix from a vector.
eye	Create a matrix with ones on the diagonal and zeros elsewhere.
magic	Create a square matrix with rows, columns, and diagonals that add up to the same number.

ones	Create a matrix or array of all ones.
rand	Create a matrix or array of uniformly distributed random numbers.
randn	Create a matrix or array of normally distributed random numbers and arrays.
randperm	Create a vector (1-by-n matrix) containing a random permutation of the specified integers.
zeros	Create a matrix or array of all zeros.

Now, let us discuss some functions that create some special arrays. For all these functions, a single argument creates a square array, double arguments create rectangular array.

7.1.1 zeros( )

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
```

7.1.2 ones ( )

The **ones( )** function creates an array of all ones. For example,

```
ones(4,3)

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

7.1.3 Eye( )

The **eye( )** function creates an identity matrix. For example,

```
>> a = eye(4)     % This creates a 4 by 4 matrix with 1 in its leading/left diagonal
                  % and zero elsewhere
```

```
a =
           1         0        0         0
           0        1        0         0
           0        0        1         0
           0        0        0         1
          >> b = eye(3,4)      % Explain what this function will do

b =
          1         0         0         0
          0         1         0         0
          0         0         1         0
```

7.1.4 Rand( ):

The **rand( )** function creates an array of uniformly distributed random numbers on (0,1). Using this function many times, even with the same input, generates a different set of numbers

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
```

Exercises: Try the following functions and report your observation

(i) a = rand(4)

(ii) b = rand(4,3)

7.1.6 randn( ) and randperm( )

The randn function behaves exactly like the rand function, but distributes its random numbers normally. randperm gives a random permutation of integers.

To get full details about each function, type the function name on the command window followed by its opening brace and wait a few seconds, the help information about the function will be displayed and you can open it up to get more details.

```
>> a = randperm(3)

a =
   3    2    1
```

```
>> b = randperm(3, 2)
```

7.1.7 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
    4    14    15     1

>> a = magic(3)
                    a =
                              8     1     6
                              3     5     7
                              4     9     2
```

(i) In this special matrix. Every row, column and diagonal add up to the same value
(ii) The sum of row 1 is **15**, row 2 is **15** and row 3 is **15**
(iii) The sum of column 1 is **15**, column 2 is **15** and column 3 is **15**
(iv) The sum of the left diagonal is **15** and the right diagonal is **15** also

7.2 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 two-dimensional 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 we 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
```

7.3. 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

The following examples illustrate some of the functions mentioned above.

7.3.1 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 = ['Zakariyau', 'Nureni', 'Samuel', 'Rauf', 'Chimezie'];
numel(s)                            % no of elements in s
```

When we run the file, it displays the following result –

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

7.3.2 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 we run the file, it displays the following result –

```
a =
    1     2     3
    4     5     6
    7     8     9


b =
    7     8     9
    1     2     3
    4     5     6

c =
    8     9     7
    2     3     1
    5     6     4
```

7.3.3 Sorting Arrays

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

```
v = [ 23 45 12 9 5 0 19 17]              % horizontal 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 we 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

ans =
    2     0     1
    2     3     4
    5     6     9

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

7.3.4 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 we run the file, it displays the following result:

```
c =
{
  [1,1] = Red
  [2,1] =  1
  [1,2] = Blue
  [2,2] =  2
  [1,3] = Green
  [2,3] =  3
  [1,4] = Yellow
  [2,4] =  4
  [1,5] = White
  [2,5] =  5
}
```

7.3.5 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 we 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 =
{
  [1,1] = Red
  [2,1] =  1
  [1,2] = Blue
  [2,2] =  2
}
```

We 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
```

7.3.6 MATLAB - Colon Notation

The **colon(:)** is one of the most useful operator in MATLAB. It is used to create vectors, subscript arrays, and **specify for iterations**. The colon (:) operator is used to indicate range within a matrix or an array.

It could be in the format: A(first:last) **or** A (first:step:last)

The variable, step, in the second format, indicates the value by which the range should increment or decrement.

If we want to create a row vector, containing integers from 1 to 10, we write 1:10

MATLAB executes the statement and returns a row vector containing the integers from 1 to 10

```
ans =

   1    2    3    4    5    6    7    8    9   10
```

If we want to specify an increment value other than one, for example –

```
100: -5: 50
```

MATLAB executes the statement and returns the following result –

```
ans =
  100    95    90    85    80    75    70    65    60    55    50
```

To generate the odd numbers from 1 to 20, type the following:

```
        >> B = 1:2:20
        B =
                1        3        5        7        9        11       13       15       17       19
```

If you do not specify a step value as in (example 1), MATLAB uses a default step of 1.

Exercise:

How would you generate all the multiples of 4 between 1 and 100?

It's quite simple in MATLAB… Just type

```
>> mult_3 = 4:4:100
```

- ▶ Why did we start the series from 4?

Let us take another example –

```
0:pi/8:pi
```

MATLAB executes the statement and returns the following result –

```
ans =
  Columns 1 through 7
      0    0.3927    0.7854    1.1781    1.5708    1.9635    2.3562
  Columns 8 through 9
      2.7489    3.1416
```

We can use the colon operator to create a vector of indices to select rows, columns or elements of arrays.

The following table describes its use for this purpose (let us have a matrix A) –

Format	Purpose
A(:,j)	is the jth column of A.
A(i,:)	is the ith row of A.
A(:,:)	is the equivalent two-dimensional array. For matrices this is the same as A.
A(j:k)	is A(j), A(j+1),...,A(k).
A(:,j:k)	is A(:,j), A(:,j+1),...,A(:,k).
A(:,:,k)	is the k^{th} page of three-dimensional array A.
A(i,j,k,:)	is a vector in four-dimensional array A. The vector includes A(i,j,k,1), A(i,j,k,2), A(i,j,k,3), and so on.
A(:)	is all the elements of A, regarded as a single column. On the left side of an assignment statement, A(:) fills A, preserving its shape from before. In this case, the right side must contain the same number of elements as A.

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

```
A = [1 2 3 4; 4 5 6 7; 7 8 9 10]
A(:,2)      % second column of A
A(:,2:3)    % second and third column of A
A(2:3,2:3)  % second and third rows and second and third columns
```

When we run the file, it displays the following result –

```
A =
    1     2     3     4
    4     5     6     7
    7     8     9    10

ans =
    2
    5
    8

ans =
    2     3
    5     6
    8     9

ans =
    5     6
    8     9
```

7.4 Other Square Array/Matrix Functions

(i) To find the sums along the columns and rows respectively, type

Try

```
>> sum(A,1)
       and
>> sum(A,2).
```

(ii) To get the sum of the left (main) diagonal, type

Type trace(A) OR >> sum(diag(A))

(iii) To find the differences between the elements of a matrix, type

```
>> d = diff(A)                 OR                >> d = diff(A, 1, 1)
          d =
           -11    9    7   -5
             4   -4   -4    4
            -5    7    9  -11
```

(a) The above gives the differences between the rows i.e (row i+1) – (row i). To find the difference between columns, type >> d = diff(A, 1, 1)

(b) The second argument specifies that the difference should be carried out just once, while the third argument specifies the dimension along which to perform the differencing

(iv) To find the product between the columns and rows of a matrix respectively, type

```
>> p = prod(A)          OR          >> p = prod(A, 1)

        p =

                2880    2156    2700    1248

>> p = prod(A, 2)

        p =

        1248

        4400

        4536

        840
```

Summary

In this Session, you have been introduced to the following MATLAB array concepts:

1 Special Arrays in MATLAB
2 Multidimensional Arrays
3. Array Functions
4 Other Square Array/Matrix Functions

Self Assessment Questions (SAQs)

Create two 4 x 4 matrices and compute the sum, difference and product of the two matrices.

Study Session 8: *Further Matrix Functions*

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, further operations that can be performed on Matrices are discussed.

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

8.1 How to find the sum of each column of a matrix and store them in a vector
8.2 How to find the sum of each row of a matrix and store them in a vector
8.3 Matrix Indexing
8.4 The Find function
8.5 Adjusting Matrices
  8.5.1 Concatenation of Matrices
  8.5.2 Vertical concatenation:
  8.5.3 Horizontal concatenation:
8.6 Operations on Matrix Diagonals
8.7 Sparse Matrices and Multidimensional Arrays

8.1 To find the sum of each column of a matrix and store them in a vector

Try the following code:

```
>> col_sum = sum(magic(5))
        col_sum =
                                65      65      65      65      65
```

8.2 To find the sum of each row of a matrix and store them in a vector

Try the following code:

```
>> col_sum = sum(magic(3), 2)
        col_sum =
                                15
                                15
                                15
```

8.3 Matrix Indexing

As in conventional programming languages, MATLAB uses row and column indices to access matrix elements. Therefore in the matrix A, below

Example:

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

the element on the second row, first column (i.e. 4) can be accessed by writing A(2,1). Note that the row index is written first and then the column index.

It is important to note that MATLAB stores matrices in a column-wise manner, i.e. matrix A above is stored in memory as

```
          1        4        7        2        5        8        3        6        9
```

By virtue of MATLAB's linear mode of string its arrays, we can access multidimensional arrays as if they were linear. This is called ***Linear Indexing***

Therefore, we could also access A(2,1) as A(2)

You can sub2ind() to obtain the linear index equivalent of a subscripted index and ind2sub() to do vice versa. Still using the matrix A above, we illustrate the two functions as follows:

Example:

```
          >> linearindex = sub2ind(size(A), 3, 2)
                    linearindex =
                                        6
>> [row col] = ind2sub(size(A), 6)
                    row =
                                        3
                    col =
                                        2
```

The colon (:) operator (discussed in the previous section) can also be used to specify that MATLAB should read every element in a column or row of a matrix.

Example: Using the same matrix A, in example 4

```
>> row1 = A(1, :)
row1 =
          16    2    3   13

>> c3 = A(:, 3)
c3 =
    3
   10
    6
   15
```

MATLAB provides a keyword for the last row or column of a matrix.

Example: To display all the elements on the last row and last column of a matrix respectively, use the **end** keyword as follows

```
>> last_row = A(end, :)
```

```
last)_row =
                     4    14    15     1
>> last_col = A(:, end)
last_col =
                              13
                               8
                              12
                               1
```

MATLAB also provides logical indexing.

A logical array index designates the elements of an array, A based on their position in the indexing array, B, and not based on their value. In this type of operation, every true element in the indexing array is treated as a positional index into the array being accessed. In the following example, B is a matrix of logical ones and zeros. The position of these elements in B determines which elements of A are designated by the expression A(B) as shown below:

Example:

```
>> A = [1 2 3; 4 5 6; 7 8 9]
                    A =
                              1         2         3
                              4         5         6
                              7         8         9
>> B = logical([0 0 1; 1 0 1; 0 0 1])
                    B =
                              0         0         1
                              1         0         1
                              0         0         1

>> A(B)
ans =
          4
          3
          6
          9
```

Observe that the returned matrix, ans, is a column vector created using the linear indices of the elements in A with respect to the ones (1) in B.

8.4 The Find function

The find function is another powerful MATLAB function for array indexing. It is quite useful with logical arrays as it returns the linear indices of nonzero elements in B, and thus helps interpret A(B) as shown in the previous example

Example: The find function in action

```
>> find(B)
                    ans =
                              2
                              7
                              8
                              9
```

Observe that find gives the linear indices of non-zero elements in B

Another interesting manipulation that can be done on matrices is to convert them to a vector using the colon operator as follows (using the matrix, A from example 11 above above):

Example:

```
>> A(:)

ans =
    1
    4
    7
    2
    5
    8
    3
    6
    9
```

8.5 Adjusting Matrices

MATLAB matrices can also be adjusted by concatenating, reshaping, rotating, transposing etc.

8.5.1 Concatenation of Matrices

Matrices can be concatenated either vertically (row-wise) or horizontally (column-wise). Concatenation means merging two matrices

Example: Given two matrices A and B as follows:

```
          >>A = [1 2 3; 4 5 6; 7 8 9];
          >>B = [10 12 14; 20 24 28; 40 48 56];
```

8.5.2 Vertical concatenation:

```
          >>C = [A; B]
C =
          1         2         3
          4         5         6
          7         8         9
          10        12        14
          20        24        28
```

```
        40      48      56
```

8.5.3 Horizontal concatenation:

>>D = [A B]

```
D =
1        2      3      10      12      14
4        5      6      20      24      28
7        8      9      40      48      56
```

There are also specialized functions for concatenating matrices along different directions/dimensions.

(i) **cat(d, A, B):** Concatenate matrices A and B along the specified dimension, d
(ii) **horzcat(A, B):** Concatenate matrices A and B along the horizontal dimension. The same as [A, B]
(iii) **vertcat(A, B):** Concatenate matrices A and B along the vertical dimension. The same as [A; B]
(iv) **repmat(A, v, h):** Produces a new matrix which replicates matrix A, v times vertically and h times horizontally.
(v) **blkdiag(A, B, C,...):** This takes in a variable number of matrices as arguments and puts each on the right diagonal of the new matrix created with every other element of the matrix being zero.

Type the following on your MATLAB command window

```
>> M = magic(3); N = [-5 -6 -9; -4 -4 -2]; ...
E = eye(2) * 8; D = blkdiag(M, N, E)
```

Your output should look like this:

```
D =
   8    1    6    0    0    0    0    0
   3    5    7    0    0    0    0    0
   4    9    2    0    0    0    0    0
   0    0    0   -5   -6   -9    0    0
   0    0    0   -4   -4   -2    0    0
   0    0    0    0    0    0    8    0
   0    0    0    0    0    0    0    8
```

The ... (ellipses) symbol is used to indicate to MATLAB that there are still more lines of code on the next line (useful when code is too long to be written on a line).

8.6 Operations on Matrix Diagonals

MATLAB provides many powerful functions for performing operations on the diagonal of matrices.

(i) **blkdiag( )** constructs a block diagonal matrix.

(ii) **diag( )** returns the diagonals of a matrix and could also return a diagonal matrix depending on the usage as we will see.

(iii)**trace( )** function compute the sum of the main diagonal of a matrix

(iv)**tril( ) and triu( )** return the lower and upper triangular part of a matrix, zeroing out all other elements of the matrix that are not within the triangle

The diag( ) function gives the major diagonal of a matrix when used with a single argument as shown in Example below.

```
>> A = [1 2 3; 4 5 6; 7 8 9];
>> diag(A)
                    ans =
                                1
                                5
                                9
```

You can also construct a diagonal matrix using the diag() function as shown in example **17B.**

```
>> A = diag([8:8:32])
A=
             8      0      0      0
             0      16     0      0
             0      0      24     0
             0      0      0      32
```

Interprete what **triu( )** function is doing using the following result:

```
>> A = [1 2 3; 4 5 6; 7 8 9];
       >> triu(A)
                    ans =
                             1   2   3
                             0   5   6
                             0   0   9
```

Guess what **tril( )** function is doing using the following result:

```
>> tril(A)
ans =
                    1   0   0
                    4   5   0
                    7   8   9
```

8.7 Sparse Matrices and Multidimensional Arrays

Some matrices can contain a large number of zeros which are actually stored in memory – using up space in memory. You can convert such matrices into a sparse matrix. Such a matrix no longer has the shape of a matrix, but does save the appropriate index of each element. Sparse matrices can also be reconverted into full ones.

Example 1

```
>> A = [magic(2) zeros(2)];
>> sp_mat= sparse( A )
        sp_mat=
                  (1,1)     1
                  (2,1)     4
                  (1,2)     3
                  (2,2)     2
```

Example 2

```
>> full_mat = full( sp_mat )
                full_mat =
                                1       3       0       0
                                4       2       0       0
```

Summary

In this study session, you have learned that:

1. blkdiag constructs a block diagonal matrix.
2. diag returns the diagonals of a matrix.
3. trace computes the sum of the elements on the main diagonal.
4. tril returns the lower triangular part of a matrix.
5. triu returns the upper triangular part of a matrix

Self-Assessment Questions with Answers

1. How can you create a 3-by-3 matrix with 3 perfect squares on its diagonals?
   `>> A = diag([4, 9, 16])`
2. How can you get all the elements on the last row of a matrix?
   Using the ***end*** keyword
3. How can I create a 4-by-4 identity matrix?
   `>> I = blkdiag([1, 1, 1, 1])`   OR   `>> I = eye(4)`
4. How do you obtain the indexes of non-zero values from a matrix?
   Using the find function

5. Given that A = ones(4), what will be the output of sum(A(:))?

```
        >> sum(A(:))
                ans =
                        16
```

The colon operator makes the sum function add all elements of A

6. How do you sum the diagonals of a matrix, M?

```
    >> sum(diag(M));             or                >> trace(M)
```

7. How do you obtain the upper and lower triangles of a matrix?

    Using the ***triu*** and ***tril*** functions respectively

Study Session 9: Manipulation of Linear Algebra

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session, we will study how linear algebra is handled in MATLAB.

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

9.1 MATLAB’s matfun Directory
9.2 Adding and Subtracting Matrices
9.3 Vector Products and Transpose
9.4 Multiplying Matrices
9.5 Identity Matrix
9.6 Factorization – Cholesky Factorization
9.7 Factorization – Lower and Upper Factorization
 9.7.1 Lower and Upper Factorization
9.8 Power and Exponential
9.9 Eigenvalues – Eigenvalue Decomposition
 9.9.1 Eigenvalue Decomposition
 9.9.2 Multiple Eigenvalues
9.10 Schur Decomposition

9.1 MATLAB’s matfun Directory

MATLAB’s list of Linear Algebra functions is provided in its matfun directory and can be viewed by typing on the command window

```
>> help matfun
```

We will use some of MATLAB’s matrix generating functions to explain the Linear Algebra operations and functions.

pascal(n) generates symmetric matrices
magic(n) generates asymmetric matrices

Symmetric special matrix function

```
>> A = pascal(4)
A =
         1    1    1    1
         1    2    3    4
         1    3    6   10
         1    4   10   20
```

```
>> A’
          ans =
                     1     1     1     1
                     1     2     3     4
                     1     3     6    10
                     1     4    10    20
```

Asymmetric special matrix function

```
>> B = magic(4)

B =
   16     2     3    13
    5    11    10     8
    9     7     6    12
    4    14    15     1

>> B’
ans =
                    16     5     9     4
                     2    11     7    14
                     3    10     6    15
                    13     8    12     1
```

9.2 Adding and Subtracting Matrices

Addition and subtraction of matrices is defined just as it is for arrays, element by element. Adding A to B, and then subtracting A from the result recovers B:

```
A = pascal(3);
B = magic(3);
X = A + B

X =
          9          2          7
          4          7         10
          5         12          8

Y = X - A

Y =
          8          1          6
          3          5          7
          4          9          2
```

Note that addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results:

```
C = fix(10*rand(3,2))

X = A + C
Error using plus
Matrix dimensions must agree.

w = v + s
        w =
                9       7       6
```

9.3 Vector Products and Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, the *inner* product, or a matrix, the *outer product*:

```
u = [3; 1; 4];
v = [2  0 -1];

x = v*u
x =
        2

X = u * v
X =
        6       0       -3
        2       0       -1
        8       0       -4
```

For real matrices, the *transpose* operation interchanges *aij* and *aji*. MATLAB uses the apostrophe operator (') to perform a complex conjugate transpose, and uses the dot-apostrophe operator (.') to transpose without conjugation.

For matrices containing all real elements, the two operators return the same result. The example matrix A is *symmetric*, so A' is equal to A. But, B is not symmetric:

```
B = magic(3);
X = B'
X =
        8       3       4
        1       5       9
        6       7       2
```

Transposition turns a row vector into a column vector:

```
x = v'
x =
          2
          0
         -1
```

If x and y are both real column vectors, the product x*y is not defined, but the two products

```
x' * y   and
y' * x
```

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

For a complex vector or matrix, z, the quantity z' not only transposes the vector or matrix, but also converts each complex element to its complex conjugate. That is, the sign of the imaginary part of each complex element changes. So if

```
z = [1+2i 7-3i 3+4i; 6-2i 9i 4+7i]
z =
          1.0000 + 2.0000i        7.0000 - 3.0000i         3.0000 + 4.0000i
          6.0000 - 2.0000i               0 + 9.0000i        4.0000 + 7.0000i
```

then

```
z'
ans =
          1.0000 - 2.0000i        6.0000 + 2.0000i
          7.0000 + 3.0000i               0 - 9.0000i
          3.0000 - 4.0000i        4.0000 - 7.0000i
```

The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by z.':

```
z.'
ans =
1.0000 + 2.0000i        6.0000 - 2.0000i
7.0000 - 3.0000i               0 + 9.0000i
3.0000 + 4.0000i        4.0000 + 7.0000i
```

For complex vectors, the two scalar products x' * y and y' * x are complex conjugates of each other, and the scalar product x' * x of a complex vector with itself is real.

9.4 Multiplying Matrices

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of defined when the column dimension of *A* is equal to the row dimension of *B*, or when one of them is a scalar. If *A* is *m*-by-*p* and *B* is *p*-by-

n, their product *C* is *m*-by-*n*. The product can actually be defined using MATLAB for loops, colon notation, and vector dot products:

```
A = pascal(3);
B = magic(3);

m = 3; n = 3;

for i = 1:m
        for j = 1:n
                C(i,j) = A(i,:)*B(:,j);
        end
end
```

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; *AB* is usually not equal to *BA*:

```
X = A*B
X =
        15      15      15
        26      38      26
        41      70      39

Y = B*A
Y =
        15      28      47
        15      34      60
        15      28      43
```

A matrix can be multiplied on the right by a column vector and on the left by a row vector:

```
u = [3; 1; 4];
x = A * u
x =
        8
        17
        30

v = [2   0  -1];
y = v * B
y =
        12  -7   10
```

Rectangular matrix multiplications must satisfy the dimension compatibility conditions:

```
C = fix(10*rand(3,2));
```

```
X = A*C
X =
        17      19
        31      41
        51      70

Y = C*A
Error using mtimes
Inner matrix dimensions must agree.
```

Anything can be multiplied by a scalar:

```
s = 7;
w = s * v
w =
        14      0       -7
```

9.5 Identity Matrix

Generally accepted mathematical notation uses the capital letter *I* to denote identity matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that *AI* = *A* and *IA* = *A* whenever the dimensions are compatible. The original version of MATLAB could not use *I* for this purpose because it did not distinguish between uppercase and lowercase letters and *i* already served as a subscript and as the complex unit. So an English language pun was introduced.

The function

```
                eye(m,n)
```

returns an *m*-by-*n* rectangular identity matrix and eye(n) returns an *n*-by-*n* square identity matrix.

9.6 Factorization – Cholesky Factorization

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose. Below is the default

A = R′ * R   Where R is an upper triangular matrix.

It should be noted however that, not all symmetric matrices can be factored in this way. Consequently, all matrices that have such a factorization are termed to be ***positive definite***.

Example 1

```
>> A = pascal(6)

A =
     1     1     1     1     1     1
     1     2     3     4     5     6
     1     3     6    10    15    21
     1     4    10    20    35    56
     1     5    15    35    70   126
     1     6    21    56   126   252
```

Example 2

```
>> R = chol(A)

R =
     1     1     1     1     1     1
     0     1     2     3     4     5
     0     0     1     3     6    10
     0     0     0     1     4    10
     0     0     0     0     1     5
     0     0     0     0     0     1
```

Observe that the elements of both matrices in examples are binomial coefficients. The fact is that R‘ * R = A demonstrates an identity involving sums of products of binomial coefficients.

1. The Cholesky factorization also applies to complex matrices. Any complex matrix that has a Cholesky factorization satisfies A′ = A and is said to be ***Hermitian positive definite***.
2. The Cholesky factorization allows the linear system A * x = b to be replaced by R′ * R * x = b.
3. This is because the backslash operator recognizes triangular systems, and this has a solution in the MATLAB environment via x = R\(R'\b)

9.7 Factorization – Lower and Upper Factorization

LU factorization, or Gaussian elimination, expresses any square matrix A as the product of a permutation of a lower triangular matrix and an upper triangular matrix

A = L * U
where L is a permutation of a lower triangular matrix with ones on its diagonal and U is an upper triangular matrix.

Observe that the matrix

$$\begin{matrix} 0 & 1 \\ 1 & 0 \end{matrix}$$

cannot be expressed as the product of triangular matrices without interchanging its two rows.

However, the matrix

$$\begin{matrix} \varepsilon & 1 \\ 1 & 0 \end{matrix}$$

can be expressed as the product of triangular matrices, when ε is small, the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable.

Partial pivoting ensures that the elements of L are bounded by one in magnitude and that the elements of U are not much larger than those of A.

Example:

```
[L,U]  =  lu(B)
                    L  =
                              1.0000          0               0
                              0.3750          0.5441          1.0000
                              0.5000          1.0000          0

                    U  =
                              8.0000          1.0000          6.0000
                              0               8.5000          -1.0000
                              0               0               5.2941
```

The LU factorization of A allows the linear system A * x = b to be solved quickly with x = U\(L\b)

9.7.1 Lower and Upper Factorization

Determinants and inverses are computed from the LU factorization using

det(A) = det(L) * det(U)
and
inv(A) = inv(U)*inv(L)

You can also compute the determinants using det(A) = prod(diag(U)), though the signs of the determinants might be reversed.

9.8 Power and Exponential

If A is a square matrix and p is a positive integer, A^p effectively multiplies A by itself p-1 times.

Example:

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

>> X = A^2
X =
     3     6    10
     6    14    25
    10    25    46
```

If A is square and nonsingular, A^(-p) effectively multiplies inv(A) by itself p-1 times:

Example:

```
 >> Y = A^(-3)

Y =
  145.0000      -207.0000       81.0000
 -207.0000      298.0000       -117.0000
   81.0000      -117.0000       46.0000
```

Fractional powers, like A^(2/3), are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

The **.^** operator produces element-by-element powers. So that each element of the matrix is raise to the specified power independent of every other element

The MATLAB function given as sqrtm(A) computes A ^ (1/2) more accurately than sqrt(A), which computes A **.^** (1/2), (i.e. element by element exponentiation)

Example:

```
X = [ 3 6 10; 6 14 25; 10 25 46]

X =
    3     6    10
    6    14    25
   10    25    46
```

```
>> sqrt(X)

 ans =

 1.7321    2.4495    3.1623
 2.4495    3.7417    5.0000
 3.1623    5.0000    6.7823

sqrtm(X)
ans =
1.0000    1.0000    1.0000
1.0000    2.0000    3.0000
1.0000    3.0000    6.0000
```

9.9 Eigenvalues – Eigenvalue Decomposition

We define the eigenvalue and eigenvector of a square matrix A respectively as, a scalar λ and a nonzero vector υ satisfying the notation below
$A\upsilon = \lambda\upsilon$.

Suppose we represent the eigenvalues on the diagonal of a diagonal matrix as Λ and the corresponding eigenvectors that constitutes the columns of a matrix V, then we have the following
$AV = V\Lambda$.

If V is nonsingular, then the resulting expression becomes the eigenvalue decomposition
$A = V\Lambda V–1$.

Example:

```
>> A = [0 -6 -1; 6 2 -16; -5 20 -10]

A =
     0    -6    -1
     6     2   -16
    -5    20   -10

>> EigenValueB = eig(A)

EigenValueB =

               -3.0710 + 0.0000i
               -2.4645 +17.6008i
               -2.4645 -17.6008i
```

The example above produces a column vector containing the eigenvalues. In this case, the matrix produces eigenvalues that are complex. The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as t increases. The nonzero imaginary part of two of the eigenvalues, ±ω, contributes the oscillatory component, sin(ωt), to the solution of the differential equation.

9.9.1 Eigenvalue Decomposition

With two output arguments, *eig* computes the eigenvectors and stores the eigenvalues in a diagonal matrix:

Example:

```
>> [V, D] = eig(A)
V =
  -0.8326 + 0.0000i     0.2003 - 0.1394i          0.2003 + 0.1394i
  -0.3553 + 0.0000i      -0.2110 - 0.6447i       -0.2110 + 0.6447i
  -0.4248 + 0.0000i     -0.6930 + 0.0000i       -0.6930 + 0.0000i

D =
  -3.0710 + 0.0000i     0.0000 + 0.0000i        0.0000 + 0.0000i
   0.0000 + 0.0000i     -2.4645 +17.6008i       0.0000 + 0.0000i
   0.0000 + 0.0000i     0.0000 + 0.0000i        -2.4645 -17.6008i
```

Observe that the first eigenvector is real while the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, norm(v,2), equal to one.

9.9.2 Multiple Eigenvalues

Some matrices do not have an eigenvector decomposition. These matrices are not diagonalizable.
Example

```
>> A  =  [ 6     12        19; -9  -20      -33;     4         9         15   ]

A =
     6    12    19
    -9   -20   -33
     4     9    15

>> [V, D] = eig(A)

V =
   -0.4741   -0.4082    0.4082
    0.8127    0.8165   -0.8165
   -0.3386   -0.4082    0.4082
```

```
D =
  -1.0000          0       0
       0      1.0000       0
       0           0  1.0000
```

There is a double eigenvalue at λ = 1. The second and third columns of V are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

9.10 Schur Decomposition

Advanced matrix computations in MATLAB do not require eigenvalue decompositions. Instead, they are based on the *Schur decomposition*

A = USU′.

U is an orthogonal matrix, **S** is a block upper triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal.

The eigenvalues are revealed by the diagonal elements and blocks of S, while the columns of U provide a basis with much better numerical properties than a set of eigenvectors.

What has been described above is a defective example and the Schur decomposition of such is given by

```
>> [U, S] = schur(A)
U =
  -0.4741    0.6648    0.5774
   0.8127    0.0782    0.5774
  -0.3386   -0.7430    0.5774

S =
  -1.0000    20.7846    -44.6948
       0      1.0000     -0.6096
       0      0           1.0000
```

The double eigenvalue is contained in the lower 2-by-2 block of S.

Study Session 10: *Manipulation of Polynomials*

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session you will be introduced to how polynomials such as ax^3 + bx^2 + cx + d are handled with MATLAB

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

10.1 The meaning of Polynomials
10.2 MATLAB Polynomial Representation
10.3 Polynomial Evaluation
10.4 Roots of Polynomial
10.5 Addition and Subtraction of Polynomials
10.6 Multiplication of Polynomials
10.7 Division of Polynomials
10.8 Deriving Polynomial Equations
10.9 Equation of a Matrix
10.10 Polynomial Differentiation
10.11 Polynomial Integration
10.12 Plotting Polynomials
10.13 Polynomial Curve Fitting
10.14 Polynomial Evaluation with Matrix Arguments
10.15 Roots of Scalar Functions
    10.15.1 Solving a Nonlinear Equation in One Variable
    10.15.2 Setting Options for fzero
    10.15.3 Using a Starting Interval
    10.15.4 Using a Starting Point
10.16 Partial Fraction Expansions

10.1 What are Polynomials?

A polynomial is an expression consisting of a sum of finite number of terms, each being the product of a constant coefficient and one or more variables raised to a non-negative integer power.

$$a_n + x^n + a_{n-1}x^{n-1} + \ldots + a_0x^0$$

is a polynomial of order *n*, *n* being the highest power of the variable, *x*. Observe that a_n is the constant term of the polynomial – it has no variable term.

Example:

1. x^3 – 3x + 8 is a polynomial of order 3
2. x^6 + x – 6 is a polynomial of order 6
3. x is a polynomial of order 1

Operations permissible on polynomial were those mentioned earlier but also include evaluation of roots, differentiation, and integration amongst others. These operations are made possible via the inbuilt function ***polyfun*** located in the MATLAB directory.

10.2 MATLAB Polynomial Representation

Within the MATLAB, a polynomial is usually handled by a row vector. Using only the coefficients of the polynomial, the resultant row vector terms gives a term more than the total number of terms.

Given the following polynomial p(s) = s^4 + $3s^3$ - $15s^2$ – 2s + 9

Generating a row vector from the above, we have:

p = [1 3 -15 -2 9] or p = [1, 3, -15, -2, 9]

Note that the number of terms is one more than the original polynomial terms. This follows from the fact that MATLAB can interpret vector length of n + 1 as an nth order of the polynomial.

Suppose we have y = s^4 + 1, the resultant vector is

y = [1 0 0 0 1] or y = [1, 0, 0, 0, 1]

But where comes the stream of zeros?

We have to insert 'zeros' at the powers between s^4 and s^1 to make up for the vector.

10.3 Polynomial Evaluation

Given a value for a variable in s, a polynomial can be evaluated using the MATLAB function polyval. The syntax is polyval (c, s)

c is a vector having its elements as the coefficients of polynomial in the descending order of power, s represents the value at which the polynomial is to be evaluated.

Example:
Evaluate the value of the polynomial y = $4s^2$ + 3s + 3 at s = 1, - 3.

Solution: we shall execute for the two variables of s = 1 and -3, thus

```
      >> y = [4    3     3];
      >> s = 1;
      >> value = polyval(y, 1)
                  value =
                              10
      >> s = -3;
      >> polyval(y, -3)
                  ans =
                           30
```

10.4 Roots of Polynomial

MATLAB also provides function for determining the root of polynomials. The syntax is *roots(p).*

In the function, p stands for a row vector representing the coefficients of a polynomial. P will return a column vector r whose elements are the roots of polynomial.

Example:
Find the root of the polynomial s3 + 3s2 + 2s + 2 = 0. The solution of the polynomial is

```
>> p = [1    3     2    2]
>> p = [1    3     2    2];
>> r = roots(p)
            r =
-2.5214 + 0.0000i
-0.2393 + 0.8579i
-0.2393 - 0.8579i
```

10.5 Addition and Subtraction of Polynomials

Polynomials can be added or subtracted using the conventional symbols for these operations
Example:
Add the two polynomials y = (s^3 + 2s + 7) and x = (s^2 + 3s + 2)

```
>> y = [1       0        2         7];
>> x = [0       1        3         2];
>> z = y + x
            z =
                     1         1         5         9
```

Intuitively, polynomial x has been padded to make up for the degree of y. Thus, the resultant polynomial is z = s^3 + s^2 + 5s + 9

Example:
Subtract the polynomial x = ($4s^4$ + 2s + 2) from y = (s^2 + 6).

Hint: don’t forget to add the zeros to make up for the power of s.

Solution

```
        >> x = [4 0 0 2 2];
        >> y = [0 0 1 0 6];
        >> z = y – x
                z = [-4 0 1 -2 4]
```

We see that the resultant polynomial is $z = -4s^4 + s^2 – 2s + 4$

Once again, observe how the vectors x and y, was padded with zeros to make up for the missing terms.

10.6 Multiplication of Polynomials

Polynomial multiplication is achieved with the *conv* (convolution) function.

Example:
Evaluate the product of two polynomials (s + 3) and ($s^3 + 6s + 7$)

```
>> x = [1       3];
>> y = [1       6       7];
>> z = conv(x, y)
                z =
                 1    9    25    21
```

The resultant products of the two polynomial $z = s^3 + 9s^2 + 25s + 21$

Quiz 1:Why is padding with zeros not seen here?

Quiz 2:What happens if there are more than two polynomials to be multiplied, say z = w * x * y?

Quite simple! Use the function to multiply the first two and the result to multiply the other one…

```
conv(w, conv(x, y));
```

This first multiplies x and y and uses the result to multiply w.

10.7 Division of Polynomials

Polynomial division can be viewed as the inverse of multiplication. The function *deconv* (deconvolution) is used for polynomial division.

```
[x, r] = deconv(u, v)
```

Where u is the dividend vector, v is the divisor vector, x is the vector quotients and r is the vector of the resulting remainder

Example:
Divide the polynomial u by v given that u = s3 + 8s2 + 12s + 16 and v = s2 + 2s + 8

```
>> u = [1        8          12        16];
>> v = [1,       2          8];

>> [x, r]          = deconv(u, v)

                   x =
                             1     6

                   r =
                             0     0    -8   -32
```

The quotient from the above is x = [1   6],
i.e. the polynomial (s + 6), and the remainder is r = [0   0   -8   -32]

10.8 Deriving Polynomial Equations

We sometimes need to derive a polynomial equation from its roots. The MATLAB function *poly( )* does this.

```
x = poly(r)
```

Where r represents the column vector bearing the roots of the polynomial and x is a row vector of the coefficients of the polynomial.

Example:
Derive the polynomial whose roots are – 1, - 3

```
>> r = [-1;       - 3];
>> y = poly(r)
                   y =
                             1     4     3
```

From the above, the resultant polynomial equation is given as $s^2 + 4s + 3 = 0$.
Note that the roots of the equation could also be complex.

10.9 Equation of a Matrix

Every matrix has an equation (often referred to as the characteristic polynomial of a matrix) from which it has been derived.

This equation can be derived by the MATLAB function   p = poly(A)

Where A is the matrix whose equation is to be derived and p is the row vector containing the coefficients of the matrix equation in descending order of power of the variable terms.

Example:

Given the matrix A = $\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 2 \\ 7 & 6 & 5 \end{bmatrix}$

This can be represented in MATLAB as A = [1 3 5; 2 4 2; 7 6 5]

```
>> p = poly(A)
                    p =
                       1.0000  -10.0000  -24.0000   60.0000
```

Thus the vector p = [1 -10 -24   60] represents the constituents of the equation of matrix A which is given as s3 – 10s2 – 24s + 60 = 0.

10.10 Polynomial Differentiation

MATLAB provides for polynomial differentiation through the function polyder as follows

dydx = polyder(y)

Where y is the vector of the coefficients of the polynomial to be differentiated and dydx is the vector of the coefficient of the derived polynomial

Example :

Evaluate the derivatives of the polynomial y = $s^4 + 6s^3 + 8s^2 + 12$

```
>> y = [1      6     8     0      12];
          >> dydx = polyder(y);

                    dydx =
                                        4        18       16       0
```

i.e. $\frac{dy}{dx} = 4s^3 + 18s^2 + 16s$

10.11 Polynomial Integration

MATLAB also provides for polynomial integration through the function *polyint*

x = polyint(y, k)

Where y is the vector of the coefficients of the polynomial to integrate and k is an optional scalar constant of integration, while x contains the result.

Example: Integrate the polynomial $y = 4s^3 + 8s^2 – 12s + 3$, taking $k = 3$.

Generate a row vector from the expression to have: $y = [4 \quad 8 \quad -12 \quad 3]$

```
>> y = [4 8 -12 3];
>>  u = polyint(y, 3)
                    u =
                         1.0000    2.6667   -6.0000    3.0000    3.0000
```

From the above, u is the coefficient of the polynomial which represents the integral of polynomial y.

Thus we can have $u = \int y = [1 \quad 2.7 \quad -6 \quad 3 \quad 3]$

$u = s^4 + 2.7s^3 – 6s^2 + 3s + 3$

10.12 Plotting Polynomials

Example:

```
>> s = linspace (-5, 5, 100);
>> coeff = [ 1 3 3 1];
>> A = polyval (coeff, s);
>> plot (s, A),
>> xlabel ('s')
>> ylabel ('A(s)')
```

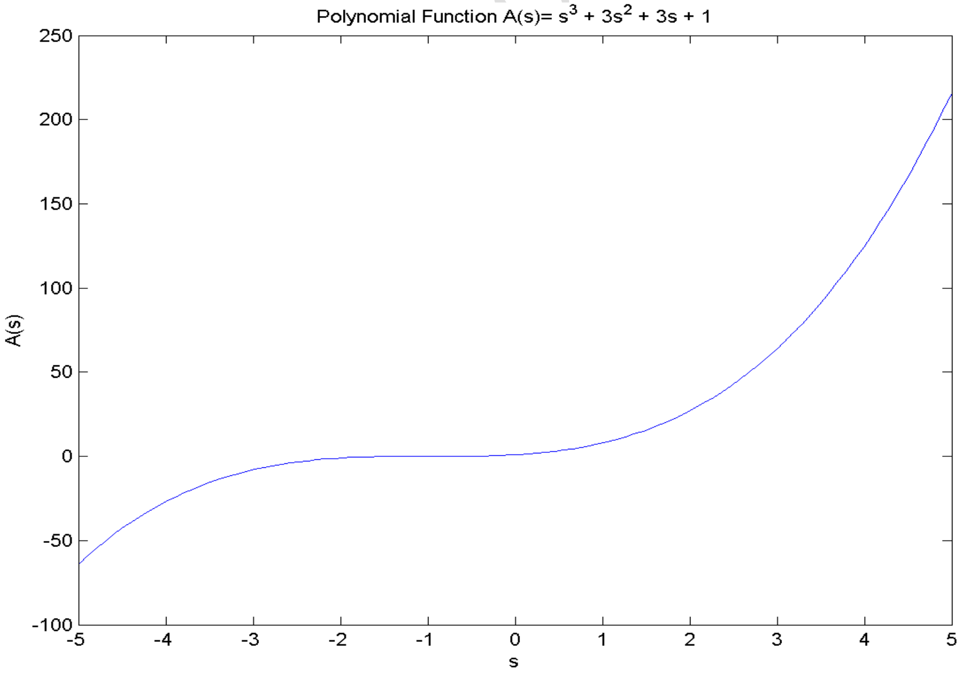


Figure 10.1: Plotting polynomial

10.13 Polynomial Curve Fitting

There are occasions in which there is need to know/derive other values to understand the interaction between different participating values. For example, the interaction between demand, supply and resultant price. When such multiple values are given say, x and y vectors, we can always derive a polynomial that fits the vectors or points with the MATLAB function *polyfit* as follows:

```
v = polyfit(x, y, k)
```

As earlier mentioned, x and y are the vectors of the points for the curve fitting, k is the order of the desired polynomial and v is evaluated to be the coefficients of k^{th} order polynomial which fits the data in descending power of x, in a least-square sense.
polyfit finds the coefficients of a polynomial that fits a set of data in a least-squares sense:

```
p = polyfit(x, y, n)
```

x and y are vectors containing the *x* and *y* data to be fitted, and n is the degree of the polynomial to return. For example, consider the $x - y$ test data

```
x = [1    2     3    4     5]; y = [5.5     43.1     128      290.7     498.4];
```

A third degree polynomial that approximately fits the data is:

```
p = polyfit(x,  y,   3)

p =
     -0.1917      31.5821       -60.3262      35.3400
```

Compute the values of the polyfit estimate over a finer range, and plot the estimate over the real data values for comparison:

```
x2 = 1:   .1  :  5;
y2 = polyval(p,   x2);

plot(x, y, 'o',  x2,  y2)
grid on
```

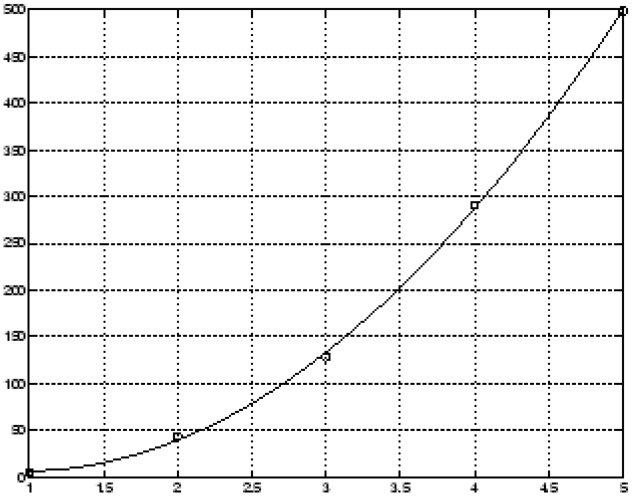


Figure 10.2: Polynomial Curve Fitting

Example: An engineer, in an experiment, had the following result values as given in the table below. Use the values to find the polynomial of degree 2 that fits the values.

Current	**13**	**15**	**21**	**25**	**31**
Voltage	120	145	170	204	245

We can generate polynomials for current and voltage from the values given above as follows

```
>> cur = [13   15      21      25      31];

>> vol = [143  165     231     275     341];
```

From the simple knowledge of physics, we know if we have the voltage and the current, then we can actually derive the resistance. Therefore, we shall assume that the resultant polynomial will be stored in the resistance value. The polynomial of degree 2 is what we want to derive, thus:

```
>> res =  polyfit(cur, vol, 1);
```

The result for res is

```
res =
          11.0000    0.0000
```

The result shows that the value of the voltage is about ten times that of current

10.14 Polynomial Evaluation with Matrix Arguments

All our earlier operations on polynomials have been limited to column or row vectors. In this section, we shall look at how we can also evaluate polynomials given a matrix argument.

Suppose we have a square matrix S, the polynomial $p(s) = s^3 + 5s + 9$ will become $p(S) = S^3 + 5S + 9I$, where I is the identity matrix and S is the square matrix.

The MATLAB function for evaluating this is

```
        v = polyvalm(a, S)
```

Where a is the resulting row vector of the polynomial to be evaluated (i.e. the coefficients)

Example:
Given a matrix polynomial $4s^2 + 3s + 5$ and the square matrix s as [1 -5   2;   7   2   5;   3   8 2]
As usual, we derive the vectors as follows

```
        >> a = [4     3      5];
        >> s = [1    -5     2;   7     2     5;   3      8      2];

        >> v = polyvalm(A, s)
```

The result is a matrix

```
v =
  -104   -11   -70
   165    47   151
   269    92   211
```

10.15   Roots of Scalar Functions

10.15.1   Solving a Nonlinear Equation in One Variable

The *fzero* function attempts to find a root of one equation with one variable. You can call this function with either a one-element starting point or a two-element vector that designates a starting interval. If you give fzero a starting point x0, fzero first searches for an interval around this point where the function changes sign. If the interval is found, fzero returns a value near where the function changes sign. If no such interval is found, fzero returns NaN. Alternatively, if you know two points where the function value differs in sign, you can specify this starting interval using a two-element vector; fzero is guaranteed to narrow down the interval and return a value near a sign change.

The following sections contain two examples that illustrate how to find a zero of a function using a starting interval and a starting point. The examples use the function *humps.m*, which is provided with MATLAB. The following figure shows the graph of humps.

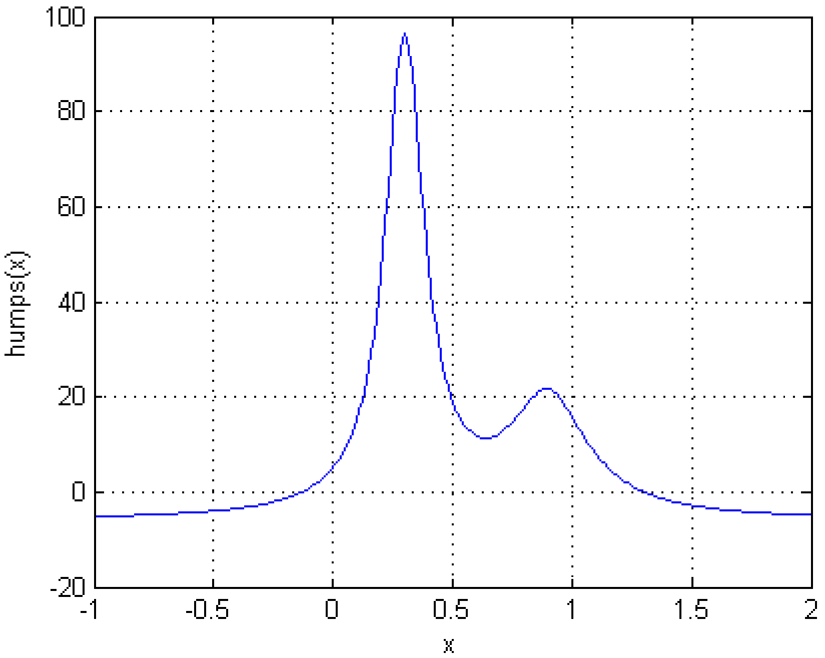


Figure 10.3: Nonlinear equation curve

10.15.2 Setting Options for fzero

You can control several aspects of the fzero function by setting options. You set options using optimset. Options include:

- Choosing the amount of display fzero generates
- Choosing various tolerances that control how fzero determines it is at a root
- Choosing a plot function for observing the progress of fzero towards a root
- Using a custom-programmed output function for observing the progress of fzero towards a root

10.15.3 Using a Starting Interval

The graph of humps indicates that the function is negative at x = -1 and positive at x = 1. You can confirm this by calculating humps at these two points.

```
>> humps(1)
ans =
        16

humps(-1)
ans =
        -5.1378
```

Consequently, you can use [-1   1] as a starting interval for fzero. The iterative algorithm for fzero finds smaller and smaller subintervals of [-1 1]. For each subinterval, the sign of humps differs at the two endpoints. As the endpoints of the subintervals get closer and closer, they converge to zero for humps.

To show the progress of fzero at each iteration, set the Display option to iter using the optimset function.

```
>>options = optimset('Display','iter');
```

Then call fzero as follows:

```
>>a = fzero(@humps,[-1 1],options)
```

This returns the following iterative output:

```
a = fzero(@humps,[-1 1],options)

Func-count        x            f(x)           Procedure
2              -1            -5.13779        initial
3               -0.513876    -4.02235        interpolation
4              -0.513876     -4.02235        bisection
5              -0.473635     -3.83767        interpolation
6              -0.115287     0.414441        bisection
7              -0.115287     0.414441        interpolation
8              -0.132562     -0.0226907      interpolation
9              -0.131666     -0.0011492       interpolation
10             -0.131618     1.88371e-007  interpolation
11             -0.131618     -2.7935e-011   interpolation
12             -0.131618     8.88178e-016  interpolation
13             -0.131618     8.88178e-016  interpolation

Zero found in the interval [-1, 1]
a =
          -0.1316
```

Each value x represents the best endpoint so far. The Procedure column tells you whether each step of the algorithm uses bisection or interpolation. You can verify that the function value at a is close to zero by entering:

```
humps(a)
ans =
          8.8818e-016
```

10.15.4 Using a Starting Point

Suppose you do not know two points at which the function values of humps differ in sign. In that case, you can choose a scalar x0 as the starting point for fzero. fzero first searches for an interval around this point on which the function changes sign. If fzero finds such an interval, it proceeds with the algorithm described in the previous section. If no such interval is found, fzero returns NaN.

For example, set the starting point to -0.2, the Display option to Iter, and call fzero:

```
options = optimset('Display','iter');
a = fzero(@humps,-0.2,options)
```

fzero returns the following output:

```
Search for an interval around -0.2 containing a sign change:

Func-count   a          f(a)        b           f(b)       Procedure
1            -0.2       -1.35385    -0.2        -1.35385   initial interval
3            -0.194343  -1.26077    -0.205657   -1.44411   search
5            -0.192     -1.22137    -0.208      -1.4807    search
7            -0.188686  -1.16477    -0.211314   -1.53167   search
9            -0.184     -1.08293    -0.216      -1.60224   search
11           -0.177373  -0.963455   -0.222627   -1.69911   search
13           -0.168     -0.786636   -0.232      -1.83055   search
15           -0.154745  -0.51962     -0.245255  -2.00602   search
17           -0.136     -0.104165   -0.264      -2.23521   search
18           -0.10949   0.572246    -0.264       -2.23521  search

Search for a zero in the interval [-0.10949, -0.264]:

Func-count     x        f(x)        Procedure
18           -0.10949   0.572246    initial
19           -0.140984  -0.219277   interpolation
20           -0.132259  -0.0154224  interpolation
21           -0.131617  3.40729e-005  interpolation
22           -0.131618  -6.79505e-008 interpolation
23           -0.131618  -2.98428e-013 interpolation
24           -0.131618  8.88178e-016 interpolation
25           -0.131618  8.88178e-016 interpolation

Zero found in the interval [-0.10949, -0.264]
a =
-0.1316
```

The endpoints of the current subinterval at each iteration are listed under the headings a and b, while the corresponding values of humps at the endpoints are listed under f(a) and f(b), respectively.

Note: The endpoints a and b are not listed in any specific order: a can be greater than b or less than b.

For the first nine steps, the sign of humps is negative at both endpoints of the current subinterval, which is shown in the output. At the tenth step, the sign of humps is positive at the endpoint, -0.10949, but negative at the endpoint, -0.264. From this point on, the algorithm continues to narrow down the interval [-0.10949 -0.264], as described in the previous section, until it reaches the value -0.1316.

10.16 Partial Fraction Expansions

residue finds the partial fraction expansion of the ratio of two polynomials. This is particularly useful for applications that represent systems in transfer function form. For polynomials b and a, if there are no multiple roots,

$$\frac{bs}{as} = \frac{\mathrm{r1}}{s-p1} + \frac{\mathrm{r2}}{s-p2} \ldots \frac{\mathrm{rn}}{s-pn} + ks$$

where r is a column vector of residues, p is a column vector of pole locations, and k is a row vector of direct terms. Consider the transfer function

$$\frac{-4x+8}{\mathrm{x}^2+6x+8}$$

```
b = [-4    8];
a = [1      6      8];

[r,p,k] = residue(b,a)

r =
          -12
            8

p =
          -4
          -2

          k =
                    [ ]
```

Given three input arguments (r, p, and k), residue converts back to polynomial form:

```
[b2,   a2] = residue(r,p,k)

b2 =
          -4    8

a2 =
          1          6          8
```

Summary

The following table lists the MATLAB polynomial functions.

S/N	Function	Description
1	conv	Multiply polynomials
2	deconv	Divide polynomials
3	fzero	Find root of continuous function of one variable
4	poly	Polynomial with specified roots
5	polyder	Polynomial derivative
6	polyfit	Polynomial curve fitting
7	polyval	Polynomial evaluation
8	polyvalm	Matrix polynomial evaluation
9	residue	Partial-fraction expansion (residues)
10	roots	Find polynomial roots

Self-Assessment Questions

Formulate some quadratic equations, use MATLAB to find the roots of those equations.

Study Session 11: Introduction to Charts Plotting

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session you will be introduced to the different types of charts that can be plotted in MATLAB and how to do the plotting.

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

11.1 How Charts are plotted
11.2 What kind of graphics is possible in MATLAB?

11.1 Plotting of Charts

The plot command produces a two-dimensional plot and is the workhorse of MATLAB graphics. It creates a plot that represents graphically the relationship between two vectors (either 1-by-N or N-by-1 arrays).

The plot command has the general form:

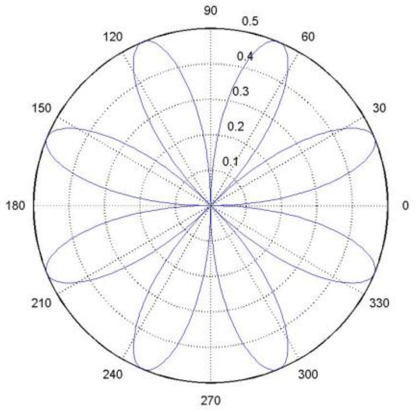
```
plot(<vector of x-values>,<vector of y-values>,<style-option string>)
```

The vectors must be of the same length and are interpreted as forming a series of (*x, y*) pairs to be plotted. By this we simply mean that the values in the first vector will be plotted on the horizontal axis and the values in the second vector will be plotted on the vertical axis. This is very important in understanding how the plot command works. The plot command simply draws markers at (*x*, *y*) points and/or connects the points with straight line segments. The command itself is in no way concerned with functional relationships; it is just connecting points.

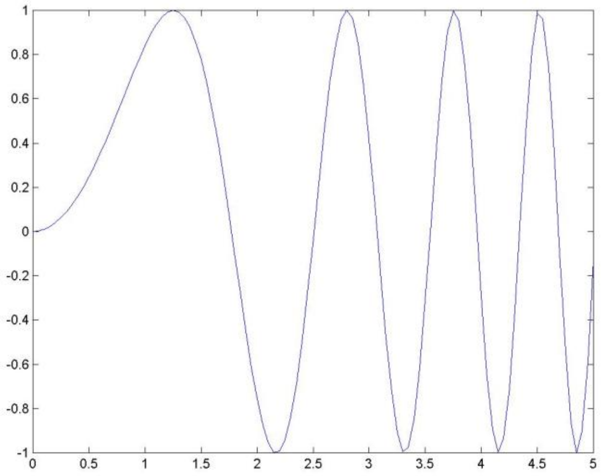
11.2 What kind of graphics is possible in Matlab?

1. **Polar plot:**

```
t=0:.01:2*pi;
polar(t,abs(sin(2*t).*cos(2*t)));
```

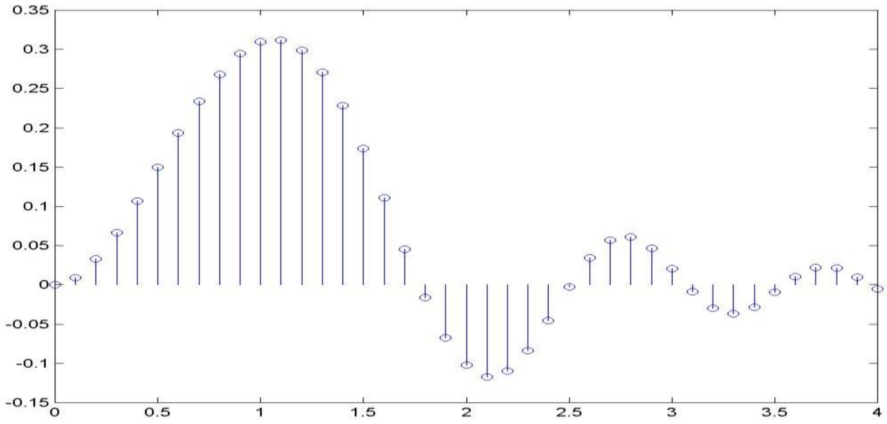


2. **Line plot:**
   x=0:0.05:5;,y=sin(x.^2);,plot(x,y);



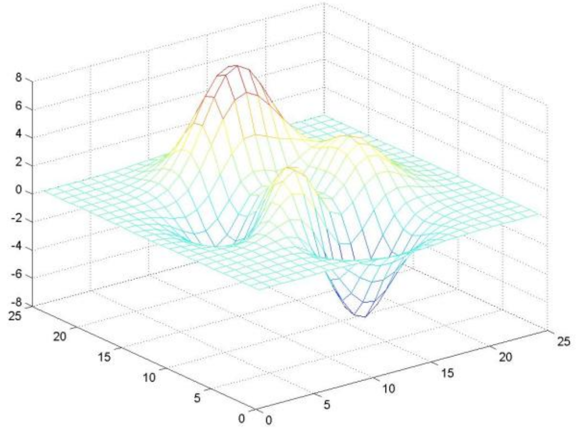
3. **Stem plot:**

x = 0:0.1:4;, y = sin(x.^2).*exp(-x); stem(x,y)



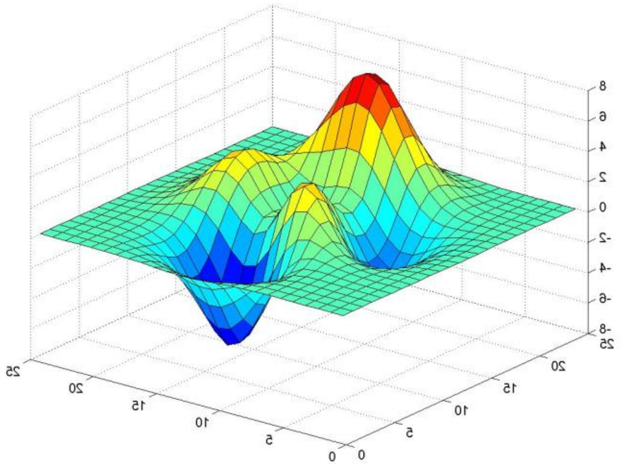
4. Mesh plot:

z=peaks(25);, mesh(z);

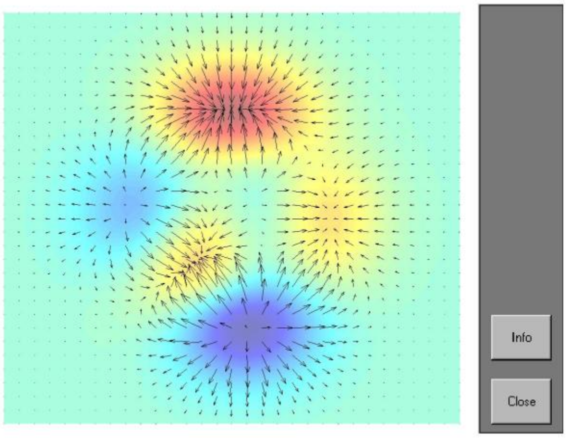


5. Surface plot:

z=peaks(25);, surf(z);, colormap(jet);

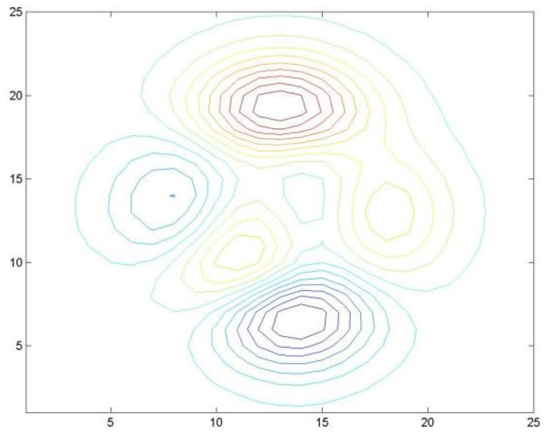


6. Quiver plot:



7. Contour plot:

z=peaks(25);,contour(z,16);



Summary

There are seven types of charts that can be plotted in MATLAB. They are:

1. Polar plot
2. Line plot
3. Stem plot
4. Mesh plot
5. Surface plot
6. Quiver plot
7. Contour plot

Self Assessment Questions

Try the following plots and write down your observations

(1) Visualization of vector data is available

```
>> x=-pi:0.1:pi; y=sin(x);
>> plot(x,y)
>> plot(x,y,’s-’)
>> xlabel(’x’); ylabel(’y=sin(x)’);
```

(2) We can change plot properties in Figure menu, or via ”handle”

```
>> h=plot(x,y); set(h, ’LineWidth’, 4);
```

(3) Many other plot functions available

```
Try: >> v=1:4; pie(v)
```

(4) Three-dimensional graphics

```
>> A = zeros(32);
>> A(14:16,14:16) = ones(3);
>> F=abs(fft2(A));
>> mesh(F)
>> rotate3d on
```

```
        >> surfl(F)
```

(5) We can change lightning and material properties

```
        >> cameramenu
        >> material metal
```

(6) Bitmap images can also be visualized

```
    >> load mandrill
    >> image(X); colormap(map)
    >> axis image off
```

Try also the following:
(7)

```
        v1=[1, 3, 4, 6, 5, 2];
        v2=[1, 2, 2, 3, 4, 2];
        plot(v1,v2,'-o');
        axis([0 7 0 5]);
```

(8) If the style-option string is 'ko', then a black circle is plotted at each of the points.

```
>> v1=[1, 3, 4, 6, 5, 2];
>> v2=[1, 2, 2, 3, 4, 2];
>> plot(v1, v2, 'ko') % circles at (1,1) (3,2) (4,2)
% (6,3) (5,4) and (2,2)
>> axis([0 7 0 5])
```

Alternately, we could plot the v2 values on the horizontal axis and the v1 values on the vertical axis

```
>> v1=[1, 3, 4, 6, 5, 2];
>> v2=[1, 2, 2, 3, 4, 2];
>> plot(v2, v1, 'ko') % circles at (1,1) (2,3) (2,4)
% (3,6) (5,4) and (2,2)
>> axis([0 5 0 7])
```

A style-option string of '-o' produces lines connecting circles,

```
v1=[1, 3, 4, 6, 5, 2];
v2=[1, 2, 2, 3, 4, 2];
plot(v1,v2,'-o');
axis([0 7 0 5]);
```

If the style-option is omitted, the default is '-', which produces a solid line connecting points.

Study Session 12: Introduction to Basic Image Processing

Expected Duration: 1 week or 2 contact hours

Introduction

In this Session you will be briefly introduced to how images are processed in MATLAB.

Learning Outcomes

When you have studied this session, you should be able to understand and explain:

12.1 The Image Processing Toolbox
12.2 Images in MATLAB
12.3 Data types in MATLAB
12.4 The following types of images are supported by MATLAB
12.5 Image Import and Export
    12.5.1 Reading and writing images in MATLAB
12.6 Images and Matrices
    12.6.1 How to build a matrix (or image)?
12.7 Image Display

12.1 What is the Image Processing Toolbox?

The Image Processing Toolbox is a collection of functions that extend the capabilities of the MATLAB's numeric computing environment. The toolbox supports a wide range of image processing operations, including:

- Geometric operations
- Neighborhood and block operations
- Linear filtering and filter design
- Transforms
- Image analysis and enhancement
- Binary image operations
- Region of interest operations

12.2 Images in MATLAB

MATLAB can import/export several image formats:

- BMP (Microsoft Windows Bitmap)
- GIF (Graphics Interchange Files)
- HDF (Hierarchical Data Format)
- JPEG (Joint Photographic Experts Group)
- PCX (Paintbrush)
- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)
- raw-data and other types of image data

Typically switch images to double to perform any processing and convert back to unsigned integer

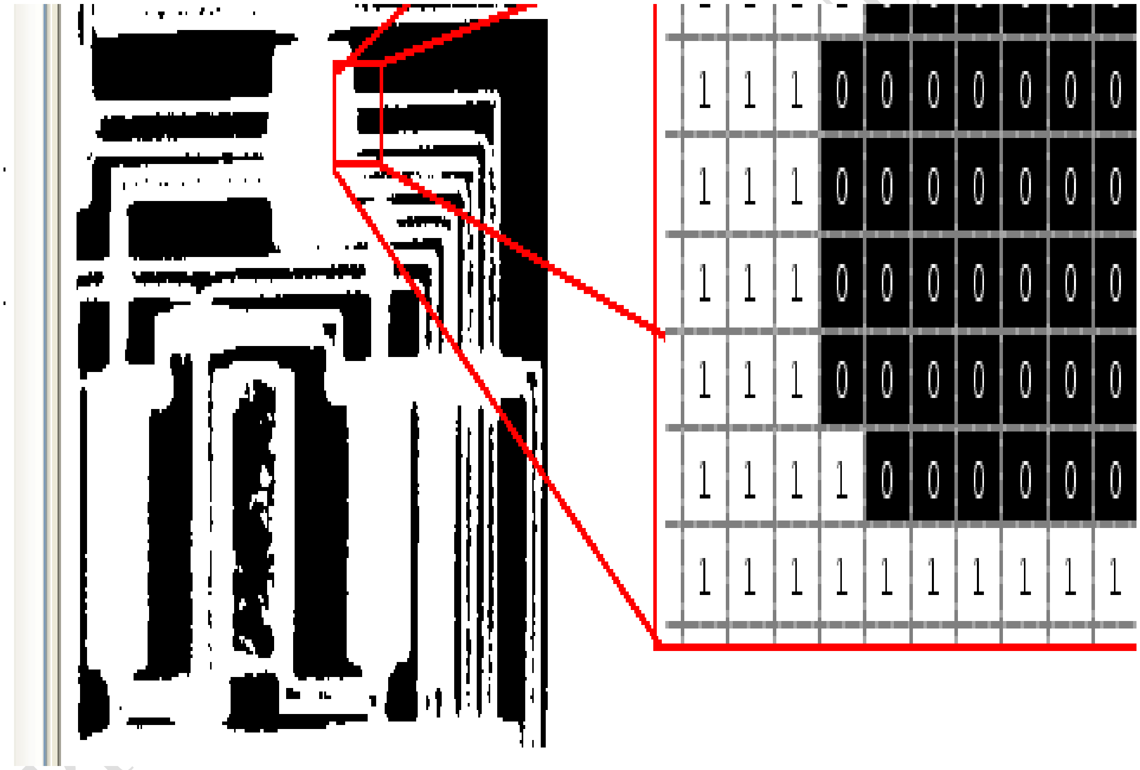
12.3 Data types in MATLAB

- Double (64-bit double-precision floating point)
- Single (32-bit single-precision floating point)
- Int32 (32-bit signed integer)
- Int16 (16-bit signed integer)
- Int8 (8-bit signed integer)
- Uint32 (32-bit unsigned integer)
- Uint16 (16-bit unsigned integer)
- Uint8 (8-bit unsigned integer)

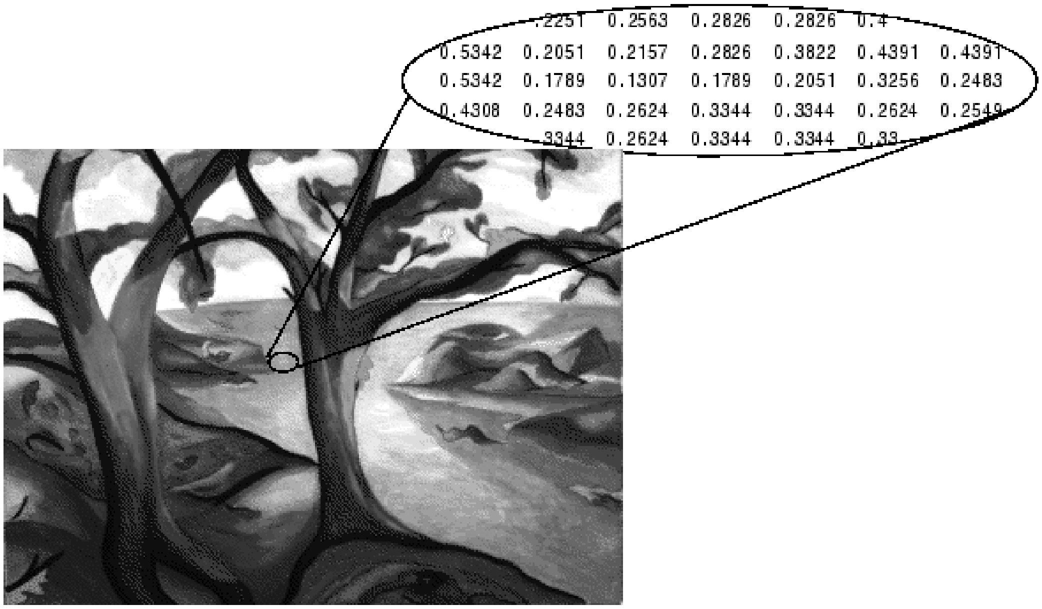
12.4 Types of Images Supported by MATLAB

The following types of images are supported by MATLAB

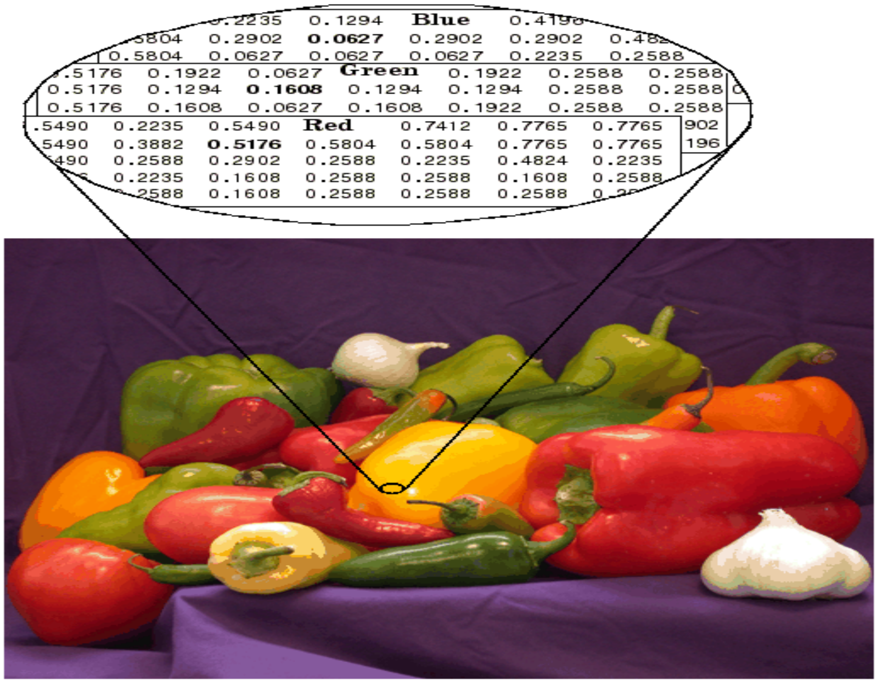
(i) Binary images : {0,1}
(ii) Intensity images : [0,1] or uint8, double etc.
(iii)RGB images : m × n × 3
(iv)Multidimensional images: m × n × p (p is the number of layers)



1	1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1



```
 .2251  0.2563  0.2826  0.2826  0.4
0.5342  0.2051  0.2157  0.2826  0.3822  0.4391  0.4391
0.5342  0.1789  0.1307  0.1789  0.2051  0.3256  0.2483
0.4308  0.2483  0.2624  0.3344  0.3344  0.2624  0.2549
  3344  0.2624  0.3344  0.3344  0.33
```



```
        .2235   0.1294  Blue    0.4198
  5804  0.2902  0.0627  0.2902  0.2902  0.482
0.5804  0.0627  0.0627  0.0627  0.2235  0.2588
.5176   0.1922  0.0627  Green   0.1922  0.2588  0.2588
0.5176  0.1294  0.1608  0.1294  0.1294  0.2588  0.2588
0.5176  0.1608  0.0627  0.1608  0.1922  0.2588  0.2588
.5490   0.2235  0.5490  Red     0.7412  0.7765  0.7765  902
5490    0.3882  0.5176  0.5804  0.5804  0.7765  0.7765  196
490     0.2588  0.2902  0.2588  0.2235  0.4824  0.2235
        0.2235  0.1608  0.2588  0.2588  0.1608  0.2588
  2588  0.1608  0.2588  0.2588  0.2588  0.
```

12.5 Image Import and Export

12.5.1 Reading and writing images in MATLAB

```
img = imread('apple.jpg');
dim = size(img);
figure;
imshow(img);
imwrite(img, 'output.bmp', 'bmp');
```

Alternatives to imshow

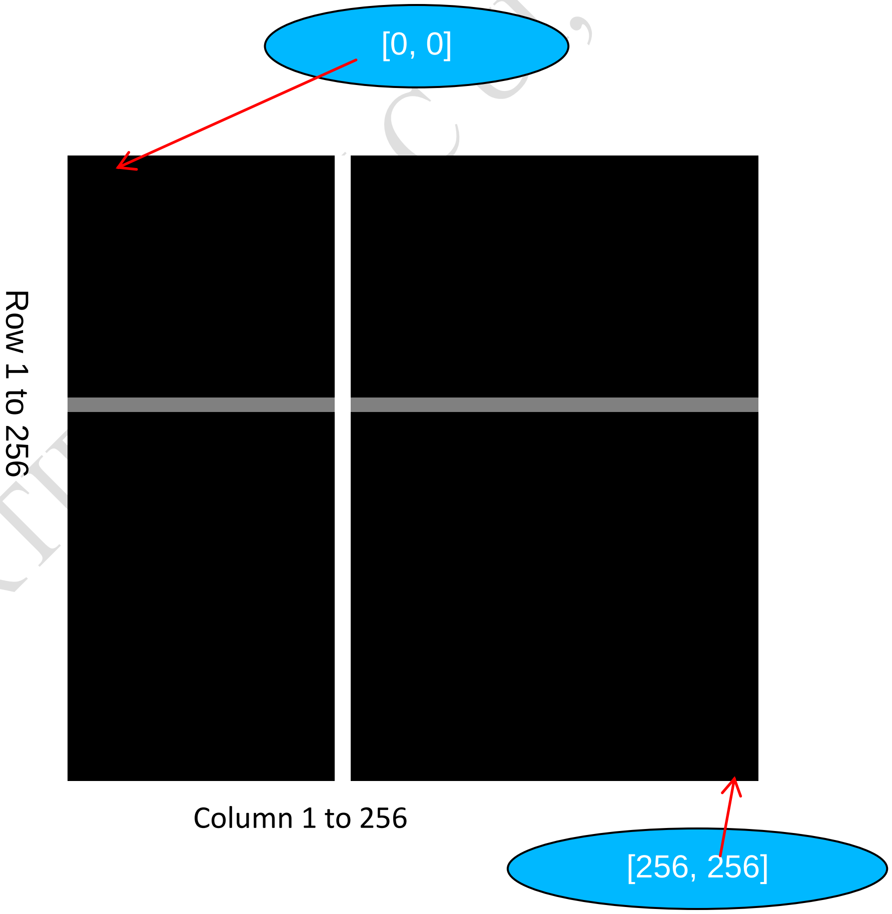
```
              imagesc(I)
             imtool(I)
             image(I)
```

12.6 Images and Matrices

12.6.1 How to build a matrix (or image)?

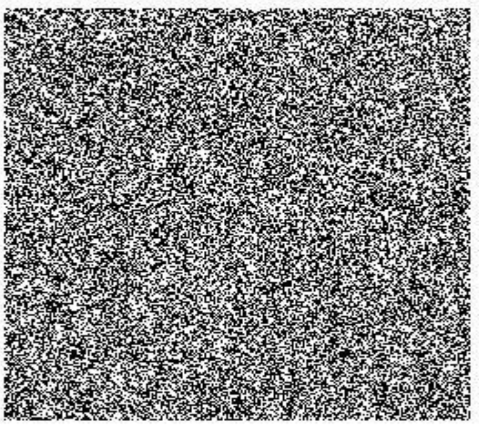
(i) Intensity Image:

```
row = 256;
col = 256;
img = zeros(row, col);
img(100:105, :) = 0.5;
img(:, 100:105) = 1;
figure;
imshow(img);
```



(ii) Binary Image:

```
row = 256;
col = 256;
img = rand(row, col);
img = round(img);
figure;
imshow(img);
size(im)
```



12.7 Image Display

Use the following functions

- image - create and display image object
- imagesc - scale and display as image
- imshow - display image

References

Craig S. Lent (2013). Learning To Program With Matlab, Building GUI Tools, John Wiley & Sons

Learn MATLAB, Simply easy learning (2015). http://www.tutorialspoint.com/matlab/

MATLAB® Mathematics The MathWorks, Inc. 3 Apple Hill Drive Natick, MA 01760-2098, March 2013, Revised for MATLAB 8.1 (Release 2013a)

Matthias Hein (2008). Matlab Tutorial

Muhamad Zahim Sujod.   An Introductory on MATLAB and Simulink

Onifade, O. F. W. (2015). Tutorial PowerPoint on MATLAB, Computer Science Department, University of Ibadan, Ibadan, Nigeria.

http://docplayer.net/15715694-Introduction-to-matlab-basics-reference-from-azernikov-sergei-mesergei-tx-technion-ac-il.html

http://www.cs.ucf.edu/~czou/CDA6530-15/ppt/matlab.ppt

http://www.mathworks.com/support/tech-notes/1100/1109.html

http://www.science.smith.edu/~jcardell/Courses/EGR326/MatlabSimulinkTutorial.ppt

https://www.slideshare.net/Muhammad_Rizwan/matlab-basic-tutorial