(15A04605) MATLAB PROGRAMMING

Prepared By

Mrs. K. JAYASREE
Associate Professor.,
VEMU IT.

COURSE OUTCOMES

- C326.1 Outline the MATLAB Environment
- C326.2 Explain the different types of Arrays and its Operations
- C326.3 Define the MATLAB Functions with data files
- C326.4 Analyze the MATLAB programming techniques
- C326.5 Apply Linear Algebraic Equations and methods for solving the engineering problems using MATLAB tool

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech III-II Sem. (ECE) **L T P C**

3 1 0 3

1. **UNIT-I:Introduction to MATLAB**

MATLAB Interactive Sessions, Menus and the toolbar, computing with MATLAB, Script files and the Editor Debugger, MATLAB Help System, Programming in MATLAB.

2. **UNIT-II: Arrays**

Arrays, Multidimensional Arrays, Element by Element Operations, Polynomial

Operations Using Arrays, Cell Arrays, Structure Arrays.

3. **UNIT-III: Functions & Files**

Elementary Mathematical Functions, User Defined Functions, Advanced Function Programming, Working with Data Files.

4. **UNIT-IV: Programming Techniques**

Program Design and Development, Relational Operators and Logical Variables,

Logical Operators and Functions, Conditional Statements, Loops, the Switch

Structure, Debugging Mat Lab Programs. Plotting :XY- plotting functions, Subplots and Overlay plots, Special Plot types, Interactive plotting, Function Discovery, Regression, 3-D plots.

5. **UNIT-V:Linear Algebraic Equations**

Elementary Solution Methods, Matrix Methods for (Linear Equations), Cramer‟s

Method, Undet- ermined Systems, Order Systems.

TEXT BOOKS:

1. G. H. Golub and C. F. Van Loan, Matrix Computations, 3rd Ed., Johns Hopkins

University Press, 1996.

2. B. N. Datta, Numerical Linear Algebra and Applications, Brooks/Cole, 1994 (out of

print) 3. L. Elden, Matrix Methods in Data Mining and Pattern Recognition, SIAM Press, 2007

UNIT-1

INTRODUCTION TO MATLAB

MATLAB INTERACTIVE SESSIONS

What Is MATLAB?

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

• MATLAB was first developed by Prof. Cleve Moler of Mexican University in the mid 1970s.

• In Matlab any data can be accessed, represented, processed and displayed in terms of matrices and hence the name MATLAB.

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

INTRODUCTION 0f MATLAB

- 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
❖ Create models and applications.

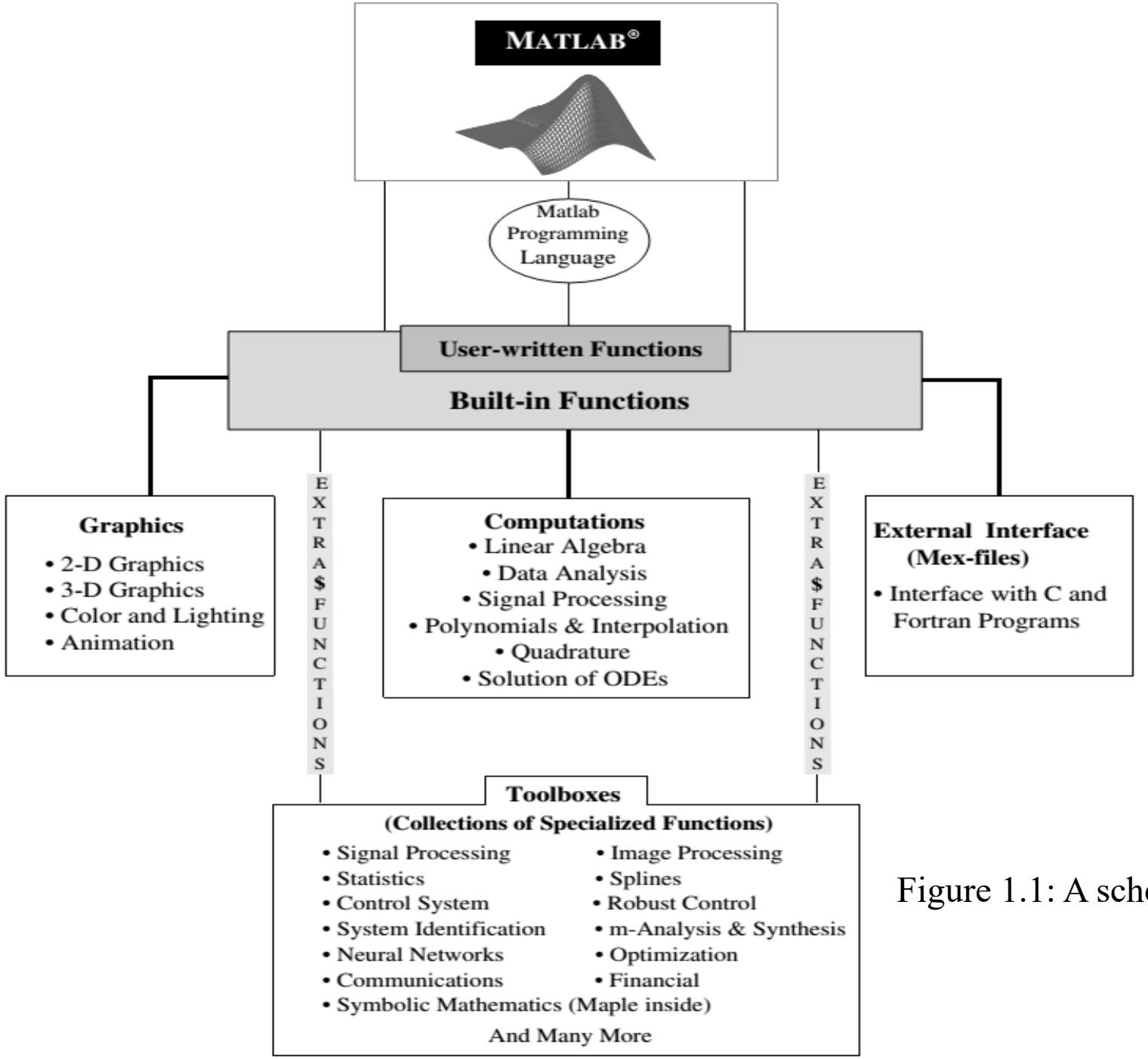


Figure 1.1: A schematic diagram of MATLAB's main features.

INTRODUCTION (CONTINUE):

- MATLAB's built-in functions provide excellent tools for
  - Linear algebra computations,
  - Data analysis,
  - Signal processing,
  - Optimization,
  - Numerical solution of ordinary differential equations (ODEs),
  - Quadrature, and many other types of scientific computations.

INTRODUCTION (CONTINUE):

- There are numerous functions for 2-D and 3-D graphics as well as for animation.
- Also, for those who cannot do without their FORTRAN or C codes, MATLAB even provides an external interface to run those programs from MATLAB.
- The user, however, is not limited to the built-in functions; he can write his own functions in the MATLAB language. Once written, these functions behave just like the built-in functions. MATLAB's language is very easy to learn and to use.
- There are also several optional 'Toolboxes' available from the developers of MATLAB. These Toolboxes are collections of functions written for special applications such as Symbolic Computation, Image Processing, Statistics Control System Design, Neural Networks, etc.
-
- The basic building block of MATLAB is the matrix. The fundamental data-type is the array. Vectors, scalars, real matrices and complex matrices are all automatically handled as special cases of the basic data-type.

Differences between the Matlab and Computer Algebra

S No	Matlab	Computer Algebra
1	It is easy to learn.	It is quite difficult to use.
2	It has a shallow learning curve(more learning with less effort).	It has a steep learning curve(more learning with more effort).
3	Can do numerical computations	Cannot do numerical computations.
4	Can do faster computations	Speed of computation is less compared to Matlab.
5	Provides a vast collection of in-built functions.	Built-in functions are limited.
6	Complexity of programming is less.	Complexity of programming is more.

Supported platforms

• MATLAB supports almost every computational platform. It supports all operating systems such as

- Windows
- AIX
- Digital Unix
- HP UX(Including UX10 and UX11)
- IRIX
- IRIX64
- Linux
- Solaris
- MacOS and
- Open VMS.

MATLAB's Power of Computational Mathematics:

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

•

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

Features of MATLAB:

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

Uses of MATLAB

- MATLAB is widely used in a range of applications including:

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

Advantages and Disadvantages of Matlab:

Advantages of MATLAB

1. Ease of Use
2. Platform Independence
3. Predefined Functions
4. Device Independent Plotting
5. Graphical User Interface
6. Matlab Compiler

Disadvantages of MATLAB

MATLAB has two principal disadvantages.

1. It is an interpreted language and therefore may execute more slowly than compiled languages.

2. The second disadvantage is cost

The MATLAB Environment

MATLAB works through three basic windows. They are

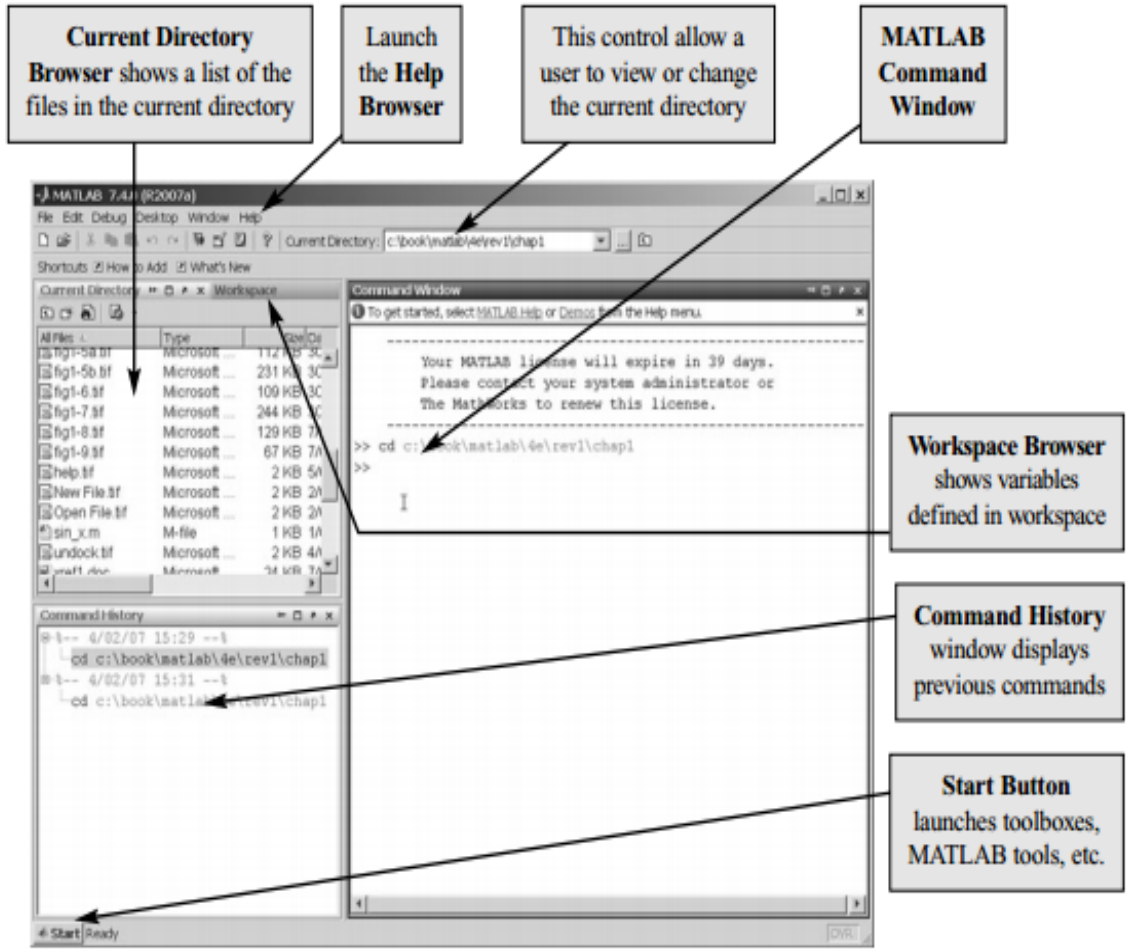
1) Matlab desktop.

2.) Editor Window.

3.) Figure Window.

1) Matlab desktop:

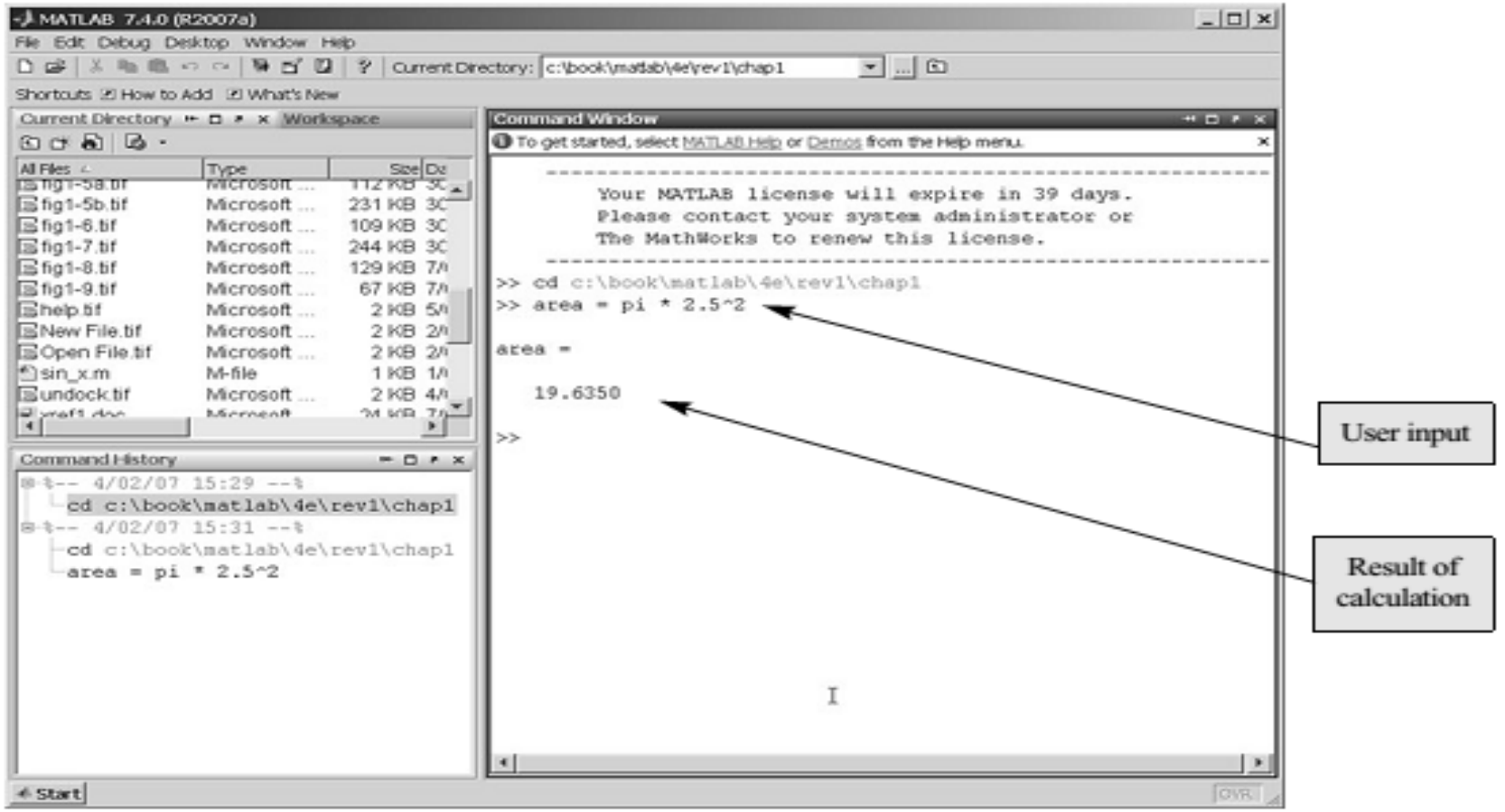
- The major tools within or accessible from the MATLAB desktop are
  - The Command Window.
  - The Command History Window.
  - The Start Button.
  - Workspace Browser.
  - Help Browser.
  - Path Browser/Current Directory/Current Folder



The Command Window:

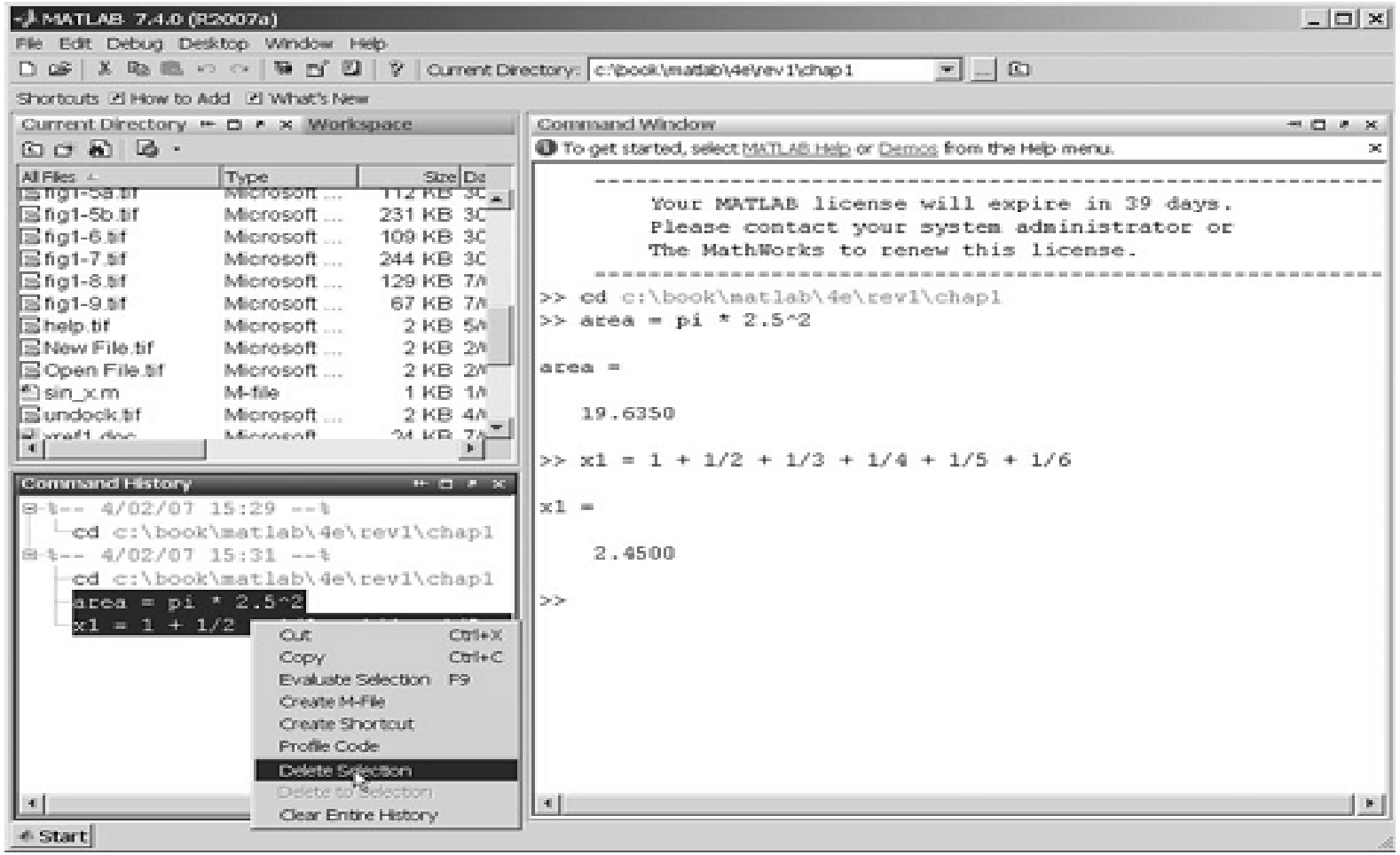
- The right-hand side of the default MATLAB desktop contains the Command Window. A user can enter interactive commands at the command prompt (») in the Command Window, and they will be executed on the spot.
- As an example of a simple interactive calculation, suppose that you want to calculate the area of a circle with a radius of 2.5 m. This can be done in the MATLAB Command Window by typing

```
» area = pi * 2.5^2

area =

   19.6350
```



The Command History Window:

- The Command History Window displays a list of the commands that a user has entered in the Command Window.



The Start Button

- The Start Button allows a user to access MATLAB tools, desktop tools, help files, and so on.

The MATLAB Workspace

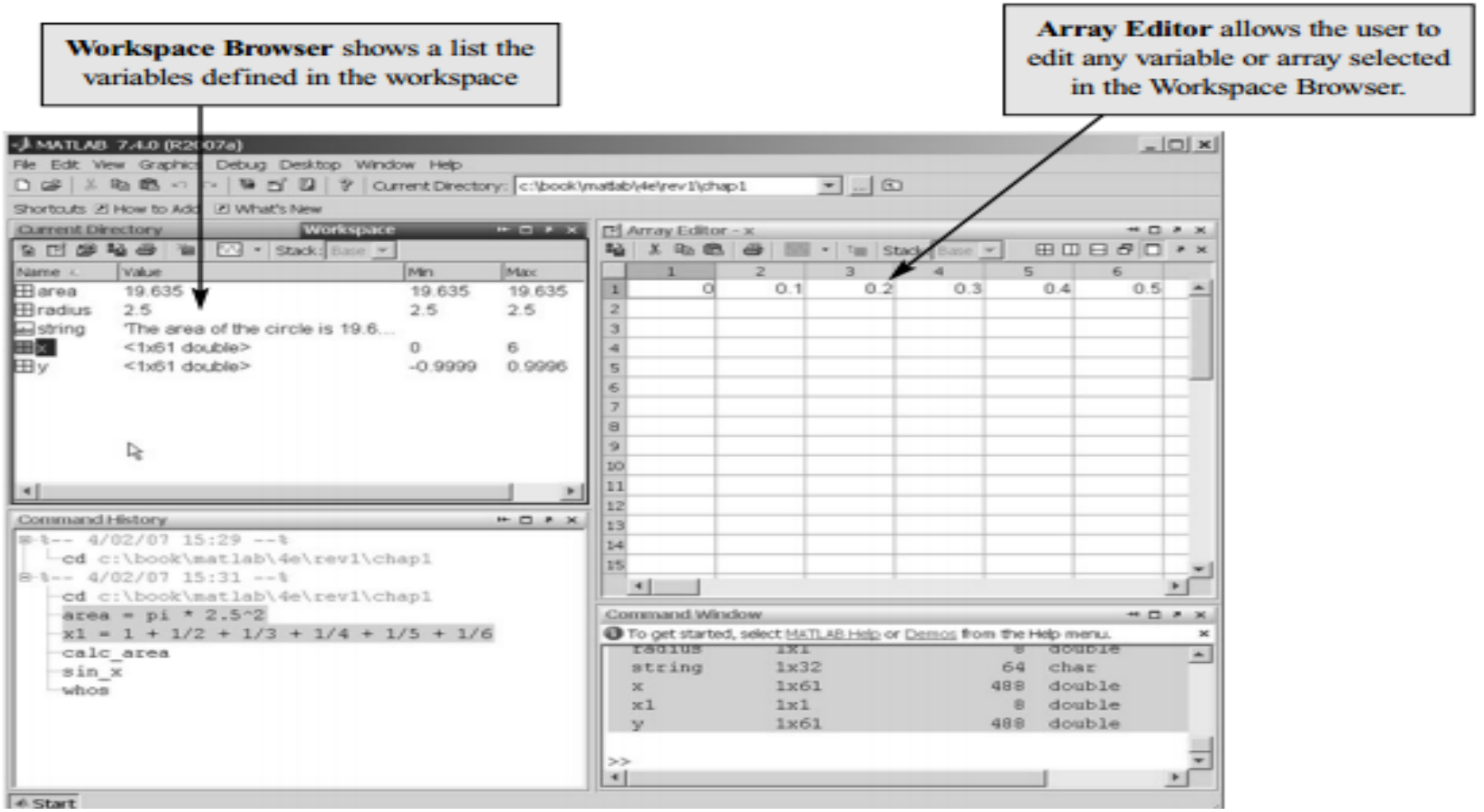
- A workspace is the collection of all the variables and arrays that can be used by MATLAB when a particular command, M-file, or function is executing.

- ***Example***:

```
» whos
  Name      Size      Bytes   Class     Attributes

  area      1x1           8   double
  radius    1x1           8   double
  string    1x32         64   char
  x         1x61        488   double
  y         1x61        488   double
```

The Workspace Browser:

• The contents of the current workspace can also be examined with a GUI-based Workspace Browser. The Workspace Browser appears by default in the upper left-hand corner of the desktop.

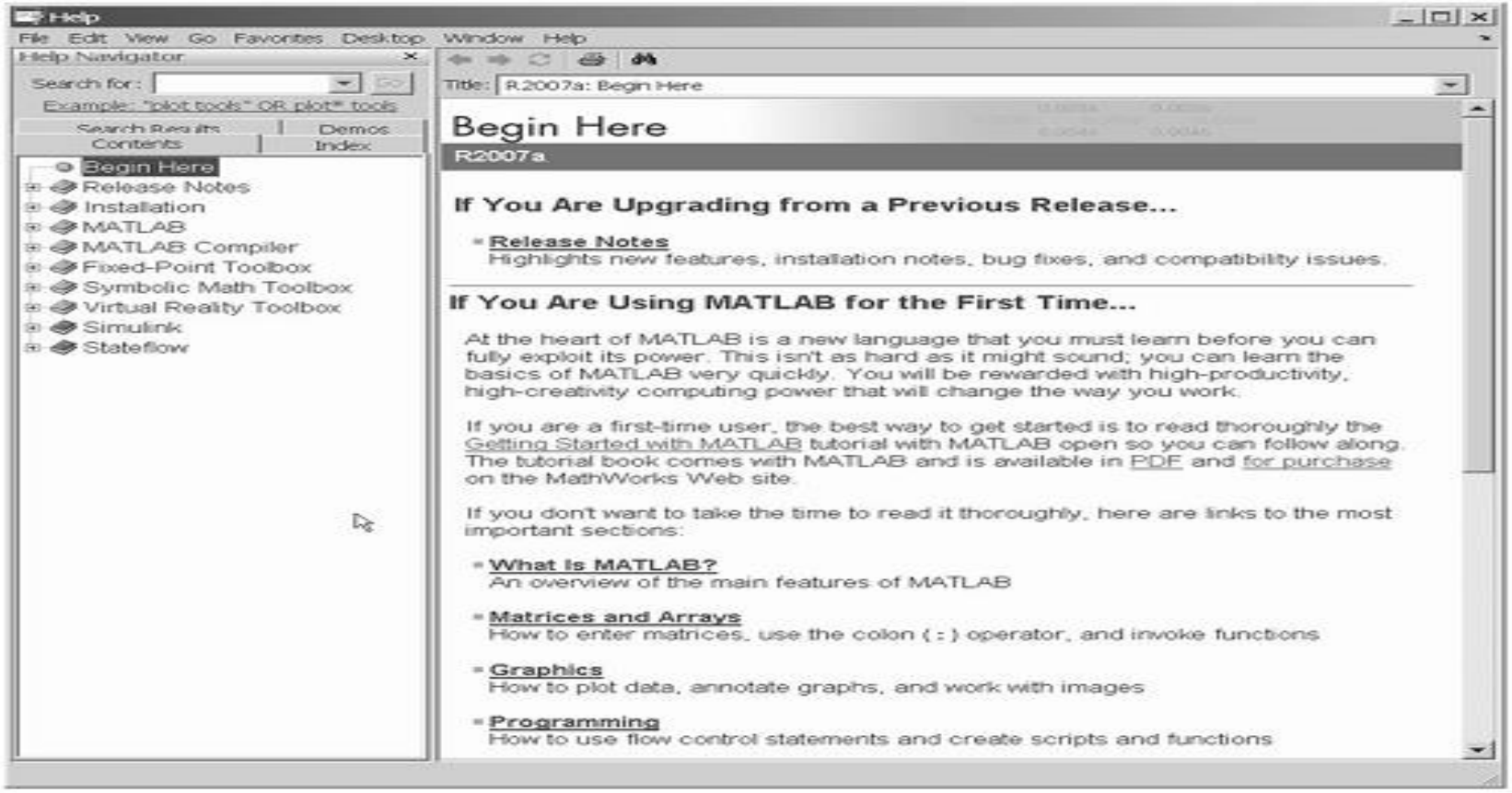


Help Browser:

- There are three ways to get help in MATLAB. The preferred method is to use the Help Browser.

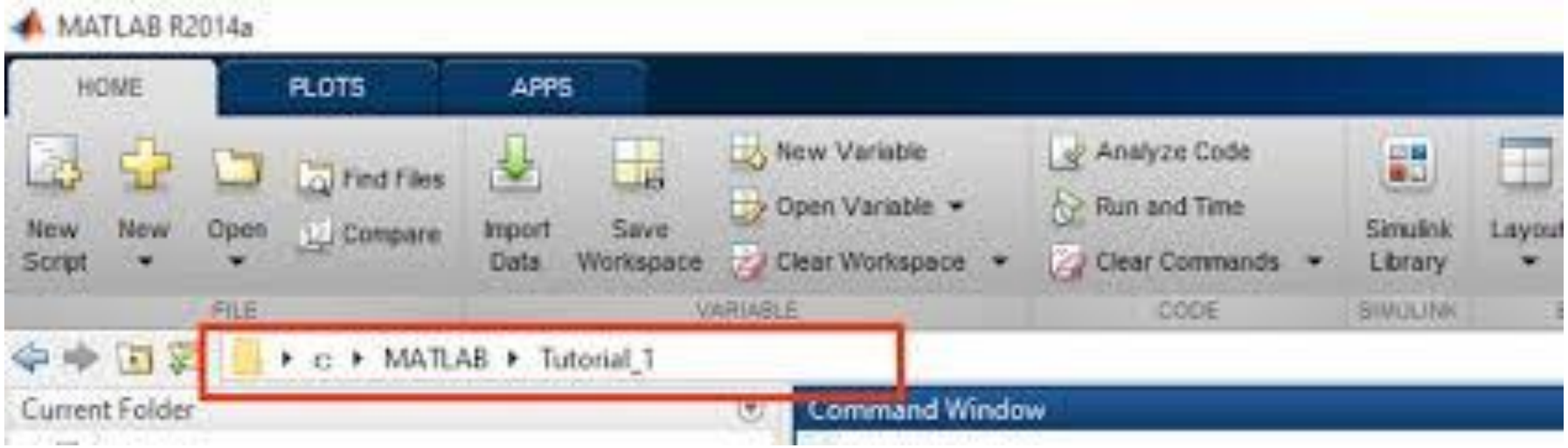
On-line help:

- **On-line documentation:** MATLAB provides on-line help for all its built-in functions and programming language constructs. The commands **lookfor**, help, **helpwin**, and **helpdesk** provide on-line help.
- **Demo:** MATLAB has a demonstration program that shows many of its features. The program includes a tutorial introduction that is worth trying.



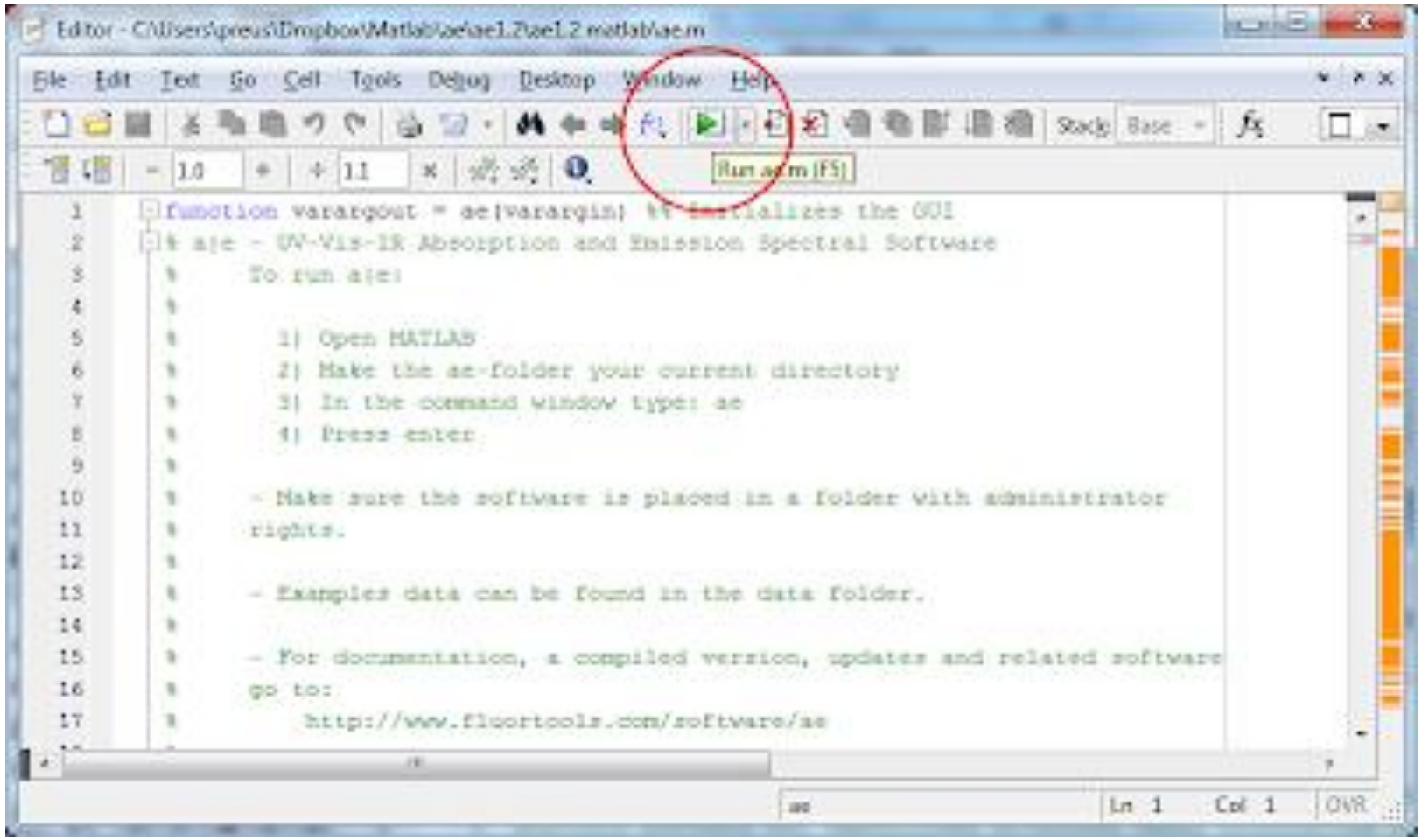
The MATLAB Search Path/ Current Folder/Current Directory:

- MATLAB has a search path that it uses to find M-files. MATLAB's M-files are organized in directories on your file system. Many of these directories of M-files are provided along with MATLAB, and users may add others.
- 1. It looks for the name as a variable.
- 2. It checks to see if the name is an M-file in the current directory. If it is, MATLAB executes
- that function or command.
- 3. It checks to see if the name is an M-file.
- 4. If it is, MATLAB executes that function or command.



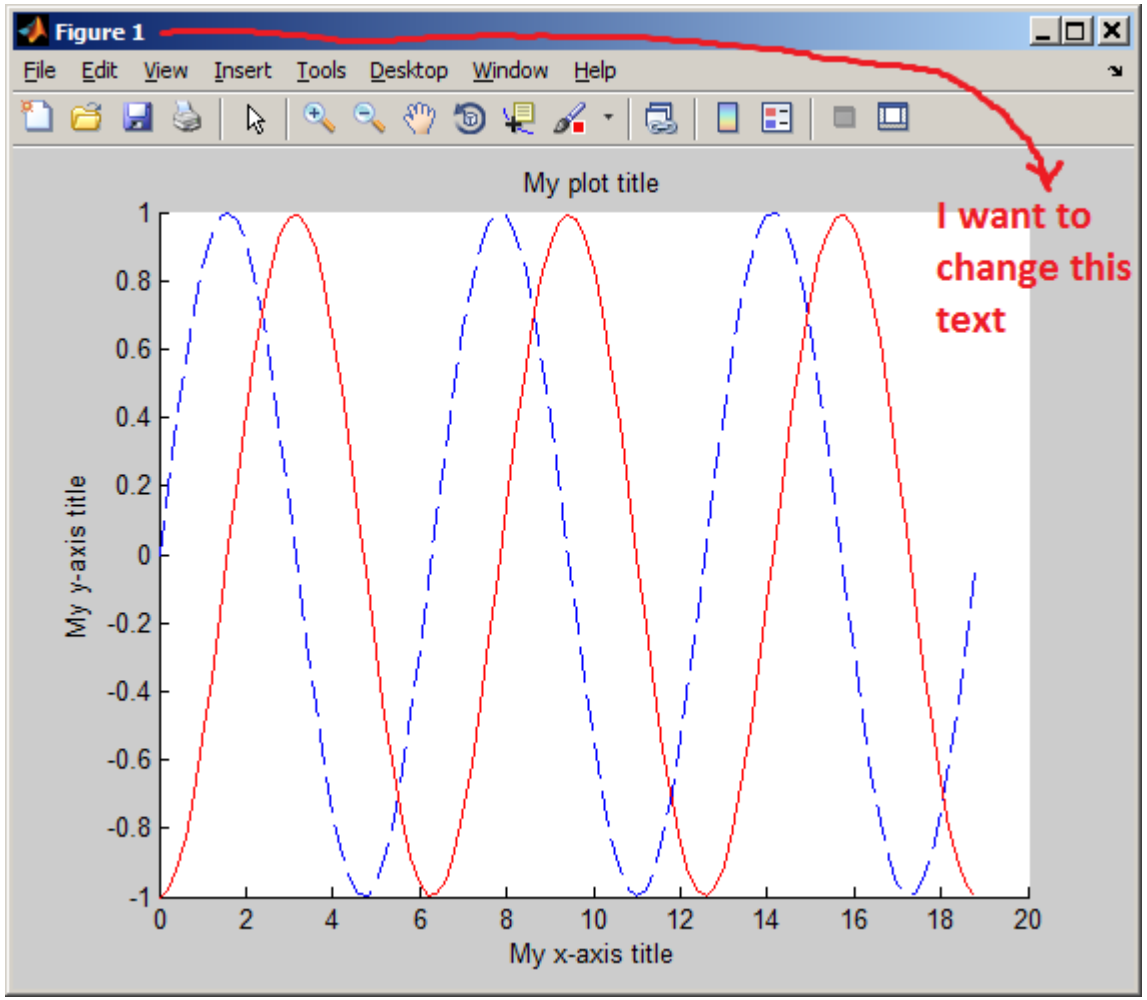
2) Editor Window:

This is where you write, edit, create, and save your own programs in files called ‘M-files’. You can use any text editor to carry out these tasks. On most systems, MATLAB provides its own built-in editor.



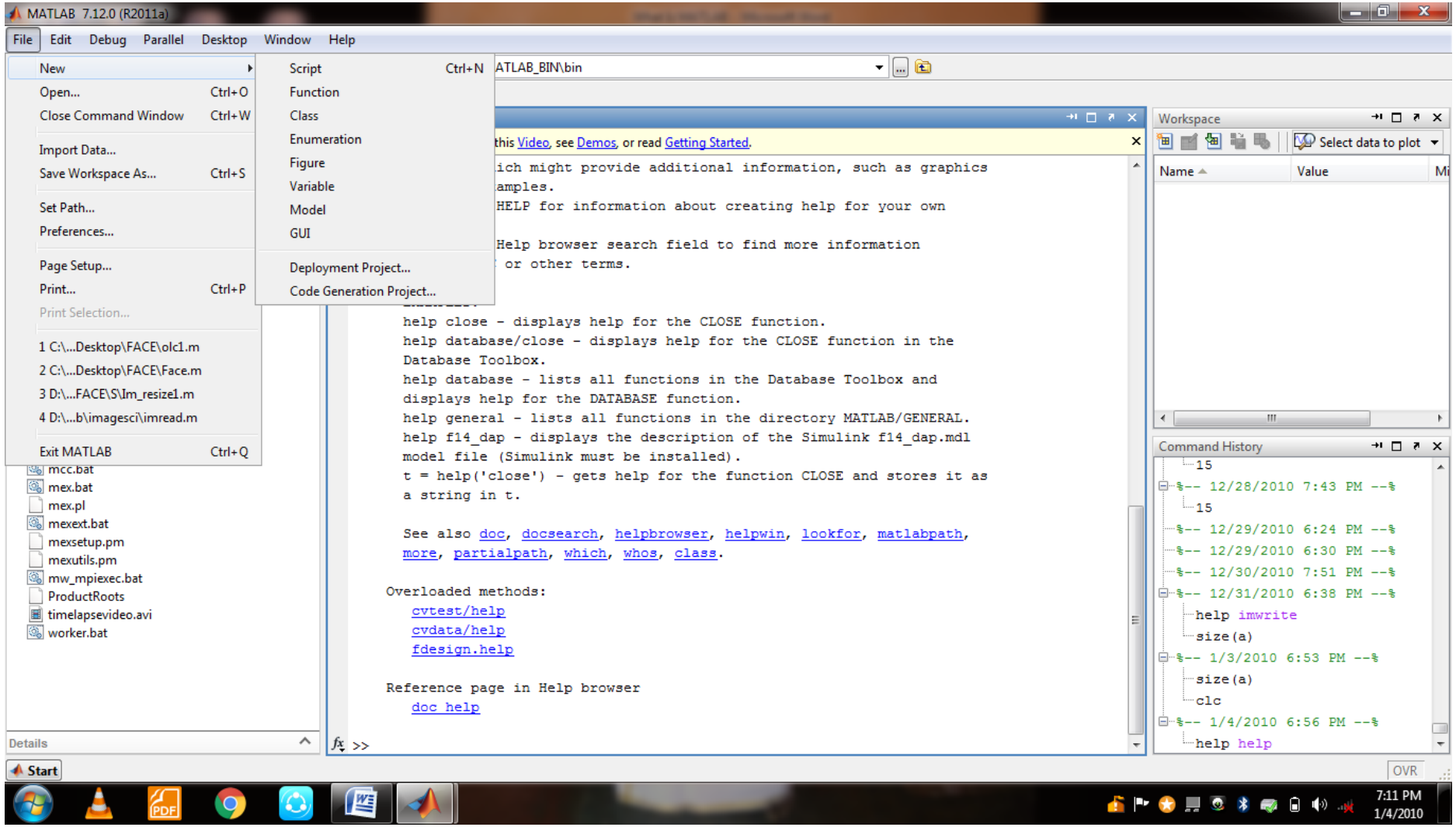
3) Figure Window

- The output of all graphics commands typed in the command window are flushed to the graphics or Figure window, a separate gray window with (default) white background color.



Menus and Tool bars

- The Matlab environment basically contains three major windows such as Matlab Desktop , Editor window and Figure Window.



Tools on Desktop window

- Matlab Desktop contains the following menus.

❖ **File menu:** This contains the options for file creation, file opening, file printing, page setup and file printing.

❖ **Edit Menu:** Which contains the options like Undo, Redo, Cut, Copy, Paste, Paste to Workshop, Select all and Delete.

❖ **Debug:** It contains the options required for debug and compilation.

❖ **Parallel:** It contains the options for configuration settings.

❖ **Desktop:** It contains the options for structuring the desktop view.

❖ **Window:** It contains the options for selecting different windows of the Matlab desktop.

❖ **Help-**It provides an instantaneous help on any topic.

File Menu

- **New…** Opens a dialog box that allows you to create a new program file, called an M-file, using a text editor called the Editor/Debugger, or a new Figure or Model file (a file type used by Simulink).
- **Open…** Opens a dialog box that allows you to select a file for editing.
- **Close Command Window…** Closes the Command window.
- **Import Data…** Starts the Import Wizard which enables you to import data easily.
- **Save Workspace As…** Opens a dialog box that enables you save a file.
- **Set Path…** Opens a dialog box that enables you to set the MATLAB search path.
- **Preferences…** Opens a dialog box that enables you to set preferences for such items as fonts, colors, tab spacing, and so forth.
- **Print…** Opens a ‘dialog box that enables you to print all of the Command window.
- **Print Selection…** Opens a dialog box that enables you to print selected portions of the Command window.
- **File List…** Contains a list of previously used files, in order of most recently used.
- **Exit…** MATLAB Closes MATLAB

Edit Menu

- **Undo…** Reverses the previous editing operation.
- **Redo…** Reverses the previous Undo operation.
- **Cut…** Removes the selected text and stores it for pasting later.
- **Copy…** Copies the selected text for pasting later, without removing it.
- **Paste…** Inserts any text on the clipboard at the current location of the cursor.
- **Paste Special…** Inserts the contents of the clipboard into the workspace as one or more variables.
- **Select All…** Highlights all text in the Command window.
- **Delete…** Clears the variable highlighted in the Workspace Browser.
- **Find…** Finds and replaces phrases.
- **Find Files….** Finds files.
- **Clear Command Window…** Removes all text from the Command window.
- **Clear Command History…** Removes all text from the Command History window.
- **Clear Workspace…** Removes the values of all variables from the workspace.

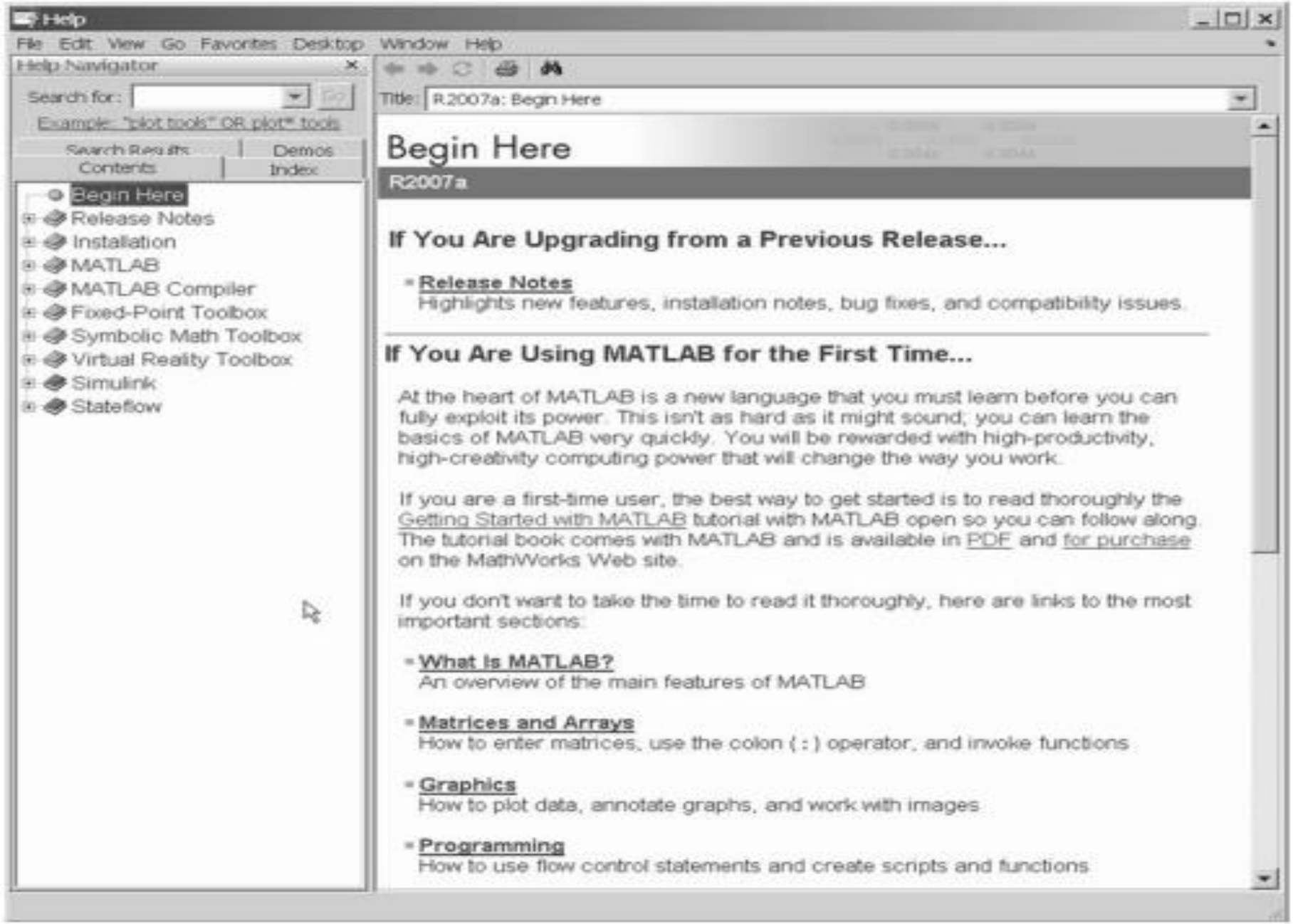
Matlab Help System

- There are three ways to get help in MATLAB. The preferred method is to use the Help Browser. The Help Browser can be started by selecting the icon from the desktop toolbar or by typing helpdesk or helpwin in the Command Window. A user can get help by browsing the MATLAB documentation, or he or she can search for the details of a particular command.

- There are also two command-line–oriented ways to get help.

1. The first way is to type help or help followed by a function name in the Command Window. If you just type help, MATLAB will display a list of possible help topics in the Command Window. If a specific function or a toolbox name is included, help will be provided for that particular function or toolbox.

2. The second way to get help is the lookfor command. The lookfor command differs from the help command in that the help command searches for an exact function name match, while the lookfor command searches the quick summary information in each function for a match.


Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search for:
Example: "plot tools" OR plot* tools
Search Results
Demos
Contents
Index
Begin Here
Release Notes
Installation
MATLAB
MATLAB Compiler
Fixed-Point Toolbox
Symbolic Math Toolbox
Virtual Reality Toolbox
Simulink
Stateflow
Title: R2007a: Begin Here
Begin Here
R2007a
If You Are Upgrading from a Previous Release...
Release Notes
Highlights new features, installation notes, bug fixes, and compatibility issues
If You Are Using MATLAB for the First Time...
At the heart of MATLAB is a new language that you must learn before you can fully exploit its power. This isn't as hard as it might sound; you can learn the basics of MATLAB very quickly. You will be rewarded with high-productivity, high-creativity computing power that will change the way you work.
If you are a first-time user, the best way to get started is to read thoroughly the Getting Started with MATLAB tutorial with MATLAB open so you can follow along. The tutorial book comes with MATLAB and is available in PDF and for purchase on the MathWorks Web site.
If you don't want to take the time to read it thoroughly, here are links to the most important sections:
What Is MATLAB?
An overview of the main features of MATLAB
Matrices and Arrays
How to enter matrices, use the colon (:) operator, and invoke functions
Graphics
How to plot data, annotate graphs, and work with images
Programming
How to use flow control statements and create scripts and functions

- » lookfor inverse
- INVHILB - Inverse Hilbert matrix.
- ACOS -Inverse cosine.
- ACOSH - Inverse hyperbolic cosine.
- ACOT - Inverse cotangent.
- ACOTH -Inverse hyperbolic cotangent.
- ACSC -Inverse cosecant.
- ACSCH -Inverse hyperbolic cosecant.
- ASEC -Inverse secant.
- ASECH -Inverse hyperbolic secant.
- ASIN -Inverse sine.
- ASINH -Inverse hyperbolic sine.
- ATAN -Inverse tangent.
- ATAN2 - Four quadrant inverse tangent.
- ATANH -Inverse hyperbolic tangent.
- ERFINV -Inverse error function.
- INV -Matrix inverse.
- PINV -Pseudo inverse.
- IFFT -Inverse discrete Fourier transform.
- IFFT2 -Two-dimensional inverse discrete Fourier transform.
- IFFTN -N-dimensional inverse discrete Fourier transform.
- IPERMUTE -Inverse permute array dimensions. From this list, we can see that the function being sought is named inv.

Matlab Script Files

- Scripts are the simplest kind of program file because they have no input or output arguments. They are useful for automating series of MATLAB® commands, such as computations that you have to perform repeatedly from the command line or series of commands you have to reference.

- You can create a new script in the following ways:

➢ Highlight commands from the Command History, right-click, and select **Create Script**.

➢ Click the **New Script**  button on the **Home** tab.

➢ Use the edit function. For example, edit *new_file_name* creates (if the file does not exist) and opens the file *new_file_name*. If *new_file_name* is unspecified, MATLAB opens a new file called Untitled.

- Save your script and run the code using either of these methods:

➢ Type the script name on the command line and press Enter. For example, to run the numGenerator.m script, type numGenerator.

➢ Click the Run  button on the Editor tab

SCRIPTS vs FUNCTONS

- Both scripts and functions allow you to reuse sequences of commands by storing them in program files.
- Scripts are the simplest type of program, since they store commands exactly as you would type them at the command line.
- However, functions are more flexible and more easily extensible.
- Create a script in a file named triarea.m that computes the area of a triangle:

```
b = 5;
h = 3;
a = 0.5*(b.*h)
```

After you save the file, you can call the script from the command line:

```
triareaa =     7.5000
```

- To make your program more flexible by converting it to a function as:

```
function a = triarea(b,h)
a = 0.5*(b.*h);
end
```

- After you save the file, you can call the function with different base and height values from the command line without modifying the script:

```
a1 = triarea(1,5)
a2 = triarea(2,10)
a3 = triarea(3,6)
a1 =    2.5000
a2 =    10
a3 =     9
```

Contents of Functions and Files

- The body of a function can include valid MATLAB expressions, control flow statements, comments, blank lines, and nested functions.
- Any variables that you create within a function are stored within a workspace specific to that function, which is separate from the base workspace.
- Program files can contain multiple functions. If the file contains only function definitions, the first function is the main function, and is the function that MATLAB associates with the file name. Functions that follow the main function or script code are called local functions. Local functions are only available within the file.

End Statements

- Functions end with either an end statement, the end of the file, or the definition line for a local function, whichever comes first. The end statement is required if:
- Any function in the file contains a nested function (a function completely contained within its parent).
- The function is a local function within a function file, and any local function in the file uses the end keyword.
- The function is a local function within a script file.
- Although it is sometimes optional, use end for better code readability.

Matlab Editor Debugger

- To debug your MATLAB program graphically, use the Editor/Debugger. Alternatively, you can use debugging functions in the Command Window.
- Before you begin debugging, make sure that your program is saved and that the program and any files it calls exist on your search path or in the current folder.
- If you run a file with unsaved changes from within the Editor, then the file is automatically saved before it runs.
- Set breakpoints to pause the execution of a MATLAB file so you can examine the value or variables where you think a problem could be. You can set breakpoints using the Editor, using functions in the Command Window, or both

Modify Section of Code While Debugging:

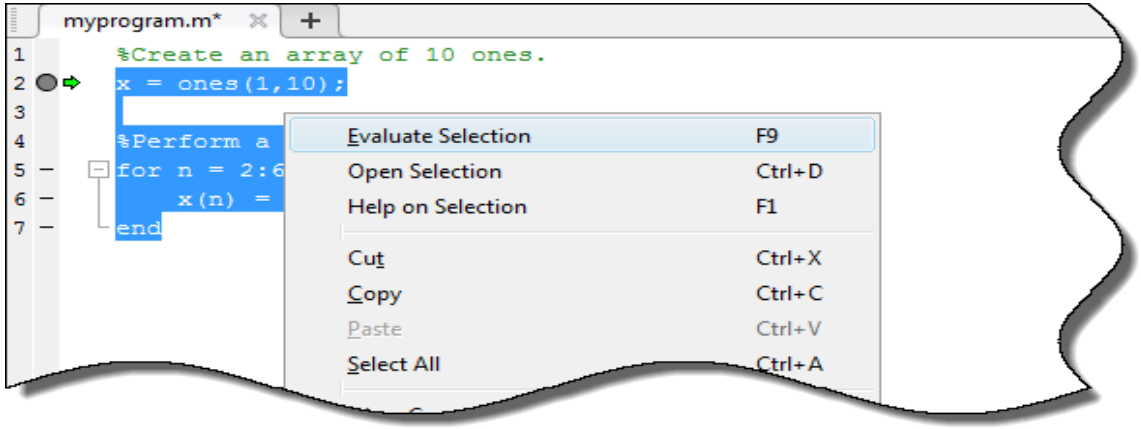
❖ ***Modify Section of Code While Debugging***

- You can modify a section of code while debugging to test possible fixes without having to save your changes.

Usually, it is a good practice to modify a MATLAB file after you quit debugging, and then save the modification and run the file. Otherwise, you might get unexpected results. However, there are situations where you want to experiment during debugging.

❖ To modify a program while debugging:

1. While your code is paused, modify a part of the file that has not yet run. Breakpoints turn gray, indicating they are invalid.
2. Select all the code after the line at which MATLAB is paused, right-click, and then select **Evaluate Selection** from the context menu.
3. After the code evaluation is complete, stop debugging and save or undo any changes made before continuing the debugging process.



This table describes available debugging actions and the different methods you can use to execute them.

Description	Toolbar Button	Function Alternative
Continue execution of file until the line where the cursor is positioned. Also available on the context menu.	**Run to Cursor**	None
Execute the current line of the file.	**Step**	dbstep
Execute the current line of the file and, if the line is a call to another function, step into that function.	**Step In**	dbstep in
Resume execution of file until completion or until another breakpoint is encountered.	**Continue**	dbcont
After stepping in, run the rest of the called function or local function, leave the called function, and pause.	**Step Out**	dbstep out
Pause debug mode.	**Pause**	None
Exit debug mode.	**Quit Debugging**	dbquit

BREAK POINTS:

- By default, MATLAB automatically opens files when it reaches a breakpoint. To disable this option:

1. From the **Home** tab, in the **Environment** section, click  **Preferences**.The Preferences dialog box opens.
2. Select **MATLAB** > **Editor/Debugger**.
3. Clear the **Automatically open file when MATLAB reaches a breakpoint** option and click **OK**.

- There are three types of breakpoints:

❖ Standard breakpoints.
❖ Conditional breakpoints
❖ Error breakpoints

Standard Breakpoints:

- A standard breakpoint pauses at a specified line in a file.
- To set a standard breakpoint click the breakpoint alley at an executable line where you want to set the breakpoint. The *breakpoint alley* is the narrow column on the left side of the Editor, to the right of the line number.
- Executable lines are indicated by a — (dash) in the breakpoint alley. If an executable statement spans multiple lines, you can set a breakpoint at each line in that statement, even though the additional lines do not have a — (dash) in the breakpoint alley.
- For example, in this code, you can set a breakpoint at all four lines:

```
1 –    if a ...
2              && b
3 –        c = 1;
4 –    end
```

- You also can set a standard breakpoint programmatically using the dbstop function. For example, to add a breakpoint at line 2 in a file named myprogram.m, type:

```
dbstop in myprogram at 2
```

MATLAB adds a breakpoint at line 2 in the function myprogram.

- MATLAB adds a breakpoint at line 2 in the function myprogram.

myprogram.m

```
1      %Create an array of 10 ones.
2      x = ones(1,10);
3
4      %Perform a calculation on items 2-6 in the array
5 -    for n = 2:6
6 -        x(n) = 2 * x(n-1);
7 -    end
```

- To examine values at increments in a for loop, set the breakpoint within the loop, rather than at the start of the loop.

➢ If you set the breakpoint at the start of the for loop, and then step through the file, MATLAB pauses at the for statement only once.

➢ However, if you place the breakpoint within the loop, MATLAB pauses at each pass through the loop

```
4      % Perform a calculation on items 2 - 6 in the array
5 -    for n = 2:6
6          x(n) = 2 * x(n - 1);
7 -    end
```

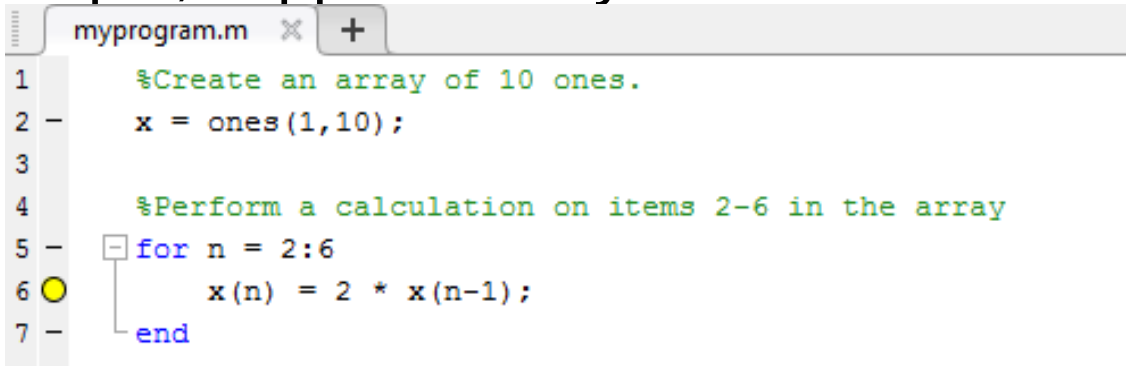
Conditional Breakpoints

- A conditional breakpoint causes MATLAB to pause at a specified line in a file only when the specified condition is met.
- Use conditional breakpoints when you want to examine results after some iterations in a loop.
- To set a conditional breakpoint,

➢ right-click the breakpoint alley at an executable line where you want to set the breakpoint and select Set/Modify Condition.

➢ When the Editor dialog box opens, enter a condition and click OK. A condition is any valid MATLAB expression that returns a logical scalar value.

- For example, suppose that you have a file called `myprogram.m`



- Add a breakpoint with the following condition at line 6:

  n >= 4

- A yellow, conditional breakpoint icon appears in the breakpoint alley at that line.
- You also can set a standard breakpoint programmatically using the dbstop function.
- For example, to add a conditional breakpoint in myprogram.m at line 6 type:

  dbstop in myprogram at 6 if n>=4

- When you run the file, MATLAB enters debug mode and pauses at the line when the condition is met.
- In the myprogram example, MATLAB runs through the for loop twice and pauses on the third iteration at line 6 when n is 4. If you continue executing, MATLAB pauses again at line 6 on the fourth iteration when n is 5.

Error Breakpoints

- An error breakpoint causes MATLAB to pause program execution and enter debug mode if MATLAB encounters a problem.
- Unlike standard and conditional breakpoints, you do not set these breakpoints at a specific line in a specific file.
- When you set an error breakpoint, MATLAB pauses at any line in any file if the error condition specified occurs. MATLAB then enters debug mode and opens the file containing the error, with the execution arrow at the line containing the error.
- To set an error breakpoint, on the Editor tab, click 

and select from these options:

➢**Pause on Errors** to pause on all errors.
➢**Pause on Warnings** to pause on all warnings.
➢**Pause on NaN or Inf** to pause on NaN (not-a-number) or Inf (infinite) values.

You also can set a breakpoint programmatically by using the dbstop function with a specified condition. For example, to pause execution on all errors, type

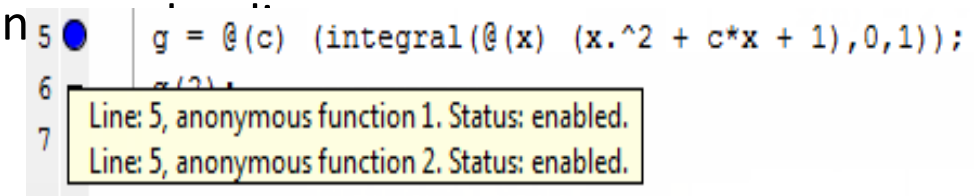
```
dbstop if error
```

To pause execution at the first run-time error within the try portion of a try/catch block that has a message ID ofMATLAB:ls:InputsMustBeStrings, type

```
dbstop if caught error MATLAB:ls:InputsMustBeStrings
```

❖Breakpoints in Anonymous Functions

- You can set multiple breakpoints in a line of MATLAB code that contains anonymous functions.
- If there is more than one breakpoint in a line, the breakpoint icon is blue, regardless of the status of any of the breakpoin

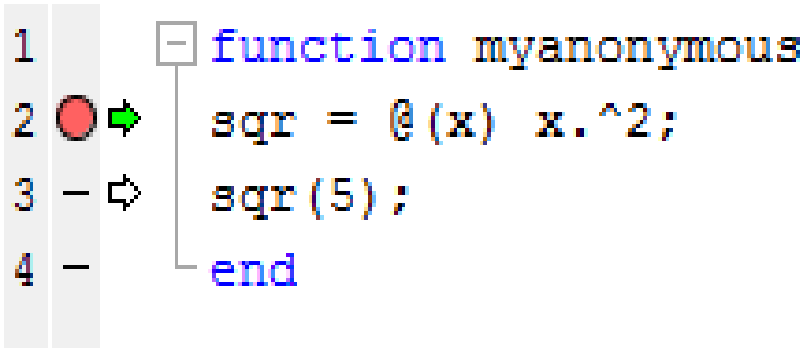


- When you set a breakpoint in an anonymous function, MATLAB pauses when the anonymous function is called.

➢ A green arrow shows where the code defines the anonymous function.

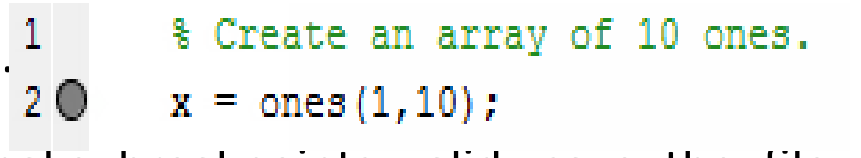
➢ A white arrow shows where the code calls the anonymous functions.

For example, in this code, MATLAB pauses the program at a breakpoint set for the anonymous function sqr, at line 2 in a file called myanonymous.m. The white arrow indicates that the sqr function is called from line 3



❖Invalid Breakpoints:

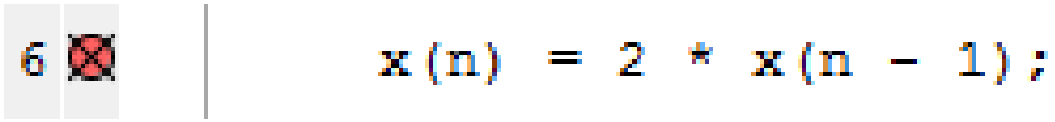
- A gray breakpoint indicates an invalid breakpoint.
- Breakpoints are invalid for these reasons:



1. There are unsaved changes in the file. To make breakpoints valid, save the file. The gray breakpoints become red, indicating that they are now valid.
2. There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. To make the breakpoint valid, fix the syntax error and save the file.

❖Disable Breakpoints:

You can disable selected breakpoints so that your program temporarily ignores them and runs uninterrupted. For example, you might disable a breakpoint after you think you identified and corrected a problem, or if you are using conditional breakpoints.
To disable a breakpoint, right-click the breakpoint icon, and select Disable Breakpoint from the context menu.
An X appears through the breakpoint icon to indicate that it is disabled.



- To reenable a breakpoint, right-click the breakpoint icon and select Enable Breakpoint from the context menu.The X no longer appears on the breakpoint icon and program execution pauses at that line. To enable or disable all breakpoints in the file, select Enable All in File or Disable All in File. These options are only available if there is at least one breakpoint to enable or disable.

❖Clear Breakpoints:

- All breakpoints remain in a file until you clear (remove) them or until they are cleared automatically at the end of your MATLAB session.
- To clear a breakpoint, right-click the breakpoint icon and select Clear Breakpoint from the context menu.
- You also can use the dbclear function. For example, to clear the breakpoint at line 6 in a file called myprogram.m, type

```
dbclear in myprogram at 6
```

- To clear all breakpoints in the file, right-click the breakpoint alley and select Clear All in File. You can also use the dbclear all command. For example, to clear all the breakpoints in a file called myprogram.m, type

```
dbclear all in myprogram
```

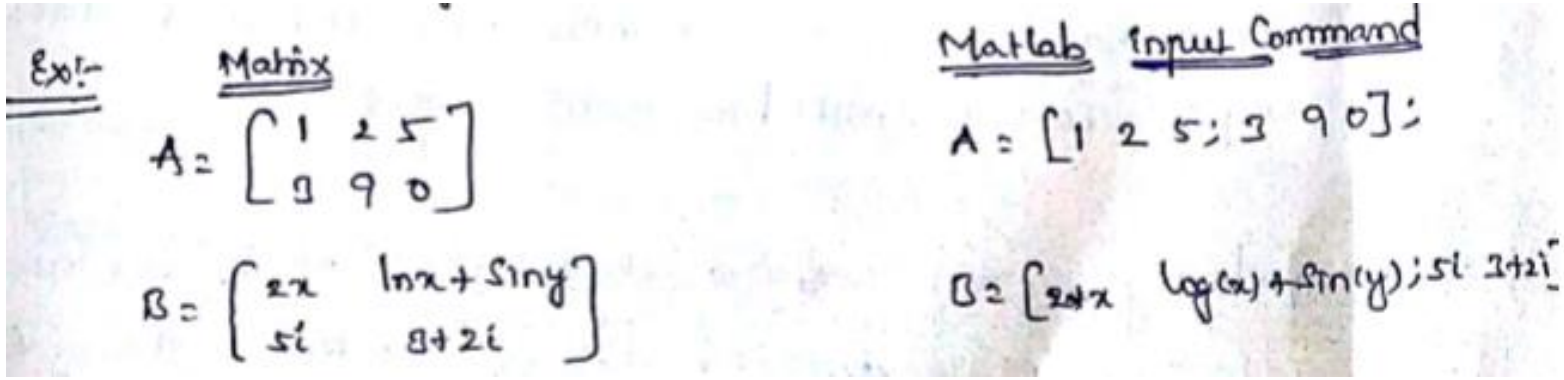
- To clear all breakpoints in all files, including error breakpoints, right-click the breakpoint alley and select Clear All. You also can use the dbclear all command.
- Breakpoints clear automatically when you end a MATLAB session. To save your breakpoints for future sessions, see the dbstatus function.

UNIT-2

ARRAYS

ARRAYS

- The group or collection of similar or identical elements which are referred by the same name is called Arrays.
- In matlab, a Matrix is entered row wise with conscentive elements of a row separated by semicolons .
- The entire matrix must be a enclosed within square brackets.
- Elements of the matrix may be real number, complex numbers or valid matlab expressions.
- Ex:



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

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

Circular Shifting of the Array Elements –

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

Live Demo

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

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

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

b =
   7     8     9
   1     2     3
   4     5     6

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

Sorting Arrays

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

Live Demo

```
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 you run the file, it displays the following result –

```
v =
   23    45    12     9     5     0    19    17
ans =
   0     5     9    12    17    19    23    45
m =
   2     6     4
   5     3     9
   2     0     1
ans =
   2     0     1
   2     3     4
   5     6     9
ans =
   2     4     6
   3     5     9
   0     1     2
```

- **Circshift**

Y = circshift(A,K)
Y = circshift(A,K,dim)

Description

Y = circshift(A,K) circularly shifts the elements in array A by K positions. If K is an integer, then circshift shifts along the first dimension of A whose size does not equal 1. If K is a vector of integers, then each element of K indicates the shift amount in the corresponding dimension of A.

Example:

If A = 1 2 3 4 5 6 7 8 9 10
Y = circshift(A,3)
Y =
8 9 10 1 2 3 4 5 6 7

Y = circshift(A,K,dim) circularly shifts the values in array A by K positions along dimension dim. Inputs K and dim must be scalars

Example:

1.A = 'racecar’; Y = circshift(A,3) Y = 'carrace’

2. Matrix A is Y = circshift(A,1,2) Y = circshift(A,[1 1])

A = 4×4

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

Y = 4×4

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

Y = 4×4

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

rotate

Rotate object about specified origin and direction

Syntax

rotate(h,direction,alpha)

rotate(...,origin)

Description

The rotate function rotates a graphics object in three-dimensional space.

- rotate(h,direction,alpha) rotates the graphics object h by alpha degrees. Specify h as a surface, patch, line, text, or image object. direction is a two- or three-element vector that describes the axis of rotation in conjunction with the origin of the axis of rotation.
- rotate(...,origin) specifies the origin of the axis of rotation as a three-element vector [x0,y0,z0].

permute

Syntax

B = permute(A,dimorder)

Description

B = permute(A,dimorder) rearranges the dimensions of an array in the order specified by the vector dimorder. For example, permute(A,[2 1]) switches the row and column dimensions of a matrix A.

```
A =
A(:,:,1) =

    0.8147    0.9134    0.2785    0.9649
    0.9058    0.6324    0.5469    0.1576
    0.1270    0.0975    0.9575    0.9706


A(:,:,2) =

    0.9572    0.1419    0.7922    0.0357
    0.4854    0.4218    0.9595    0.8491
    0.8003    0.9157    0.6557    0.9340
```

```
B = permute(A,[3 2 1])
```

```
B =
B(:,:,1) =

    0.8147    0.9134    0.2785    0.9649
    0.9572    0.1419    0.7922    0.0357


B(:,:,2) =

    0.9058    0.6324    0.5469    0.1576
    0.4854    0.4218    0.9595    0.8491


B(:,:,3) =

    0.1270    0.0975    0.9575    0.9706
    0.8003    0.9157    0.6557    0.9340
```

reshape

Reshape array

Syntax

B = reshape(A,sz)
B = reshape(A,sz1,...,szN)

Description

- B = reshape(A,sz) reshapes A using the size vector, sz, to define size(B). For example, reshape(A,[2,3]) reshapes A into a 2-by-3 matrix. sz must contain at least 2 elements, and prod(sz) must be the same as numel(A).

- B = reshape(A,sz1,...,szN) reshapes A into a sz1-by-...-by-szN array where sz1,...,szN indicates the size of each dimension. You can specify a single dimension size of [] to have the dimension size automatically calculated, such that the number of elements in B matches the number of elements in A. For example, if A is a 10-by-10 matrix, then reshape(A,2,2,[]) reshapes the 100 elements of A into a 2-by-2-by-25 array.

- Example:

1) A = 1:10;

B = reshape(A,[5,2])

```
A = 1:10;
B = reshape(A,[5,2])
```

```
B = 5×2

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

2)

```
A = 4×4

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

```
B = reshape(A,[],2)
```

```
B = 8×2

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

Array Operations:

- For function-specific information, click the link to the function reference page in the last column.

Operator	Purpose	Description
+	Addition	A+B adds A and B.
+	Unary plus	+A returns A.
-	Subtraction	A-B subtracts B from A
-	Unary minus	-A negates the elements of A.
.*	Element-wise multiplication	A.*B is the element-by-element product of A and B.
.^	Element-wise power	A.^B is the matrix with elements A(i,j) to the B(i,j) power.
./	Right array division	A./B is the matrix with elements A(i,j)/B(i,j).
.\	Left array division	A.\B is the matrix with elements B(i,j)/A(i,j).
.'	Array transpose	A.' is the array transpose of A. For complex matrices, this does not involve conjugation.

- The following table provides a summary of matrix arithmetic operators in MATLAB.

Operator	Purpose	Description
*	Matrix multiplication	C = A*B is the linear algebraic product of the matrices A and B. The number of columns of A must equal the number of rows of B.
\	Matrix left division	x = A\B is the solution to the equation $Ax = B$. Matrices A and B must have the same number of rows.
/	Matrix right division	x = B/A is the solution to the equation $xA = B$. Matrices A and B must have the same number of columns. In terms of the left division operator, B/A = (A'\B')'.
^	Matrix power	A^B is A to the power B, if B is a scalar. For other values of B, the calculation involves eigenvalues and eigenvectors.
'	Complex conjugate transpose	A' is the linear algebraic transpose of A. For complex matrices, this is the complex conjugate transpose.

Polynomial Functions

Functions

poly	Polynomial with specified roots or characteristic polynomial
polyeig	Polynomial eigenvalue problem
polyfit	Polynomial curve fitting
residue	Partial fraction expansion (partial fraction decomposition)
roots	Polynomial roots
polyval	Polynomial evaluation
polyvalm	Matrix polynomial evaluation
conv	Convolution and polynomial multiplication
deconv	Deconvolution and polynomial division
polyint	Polynomial integration
polyder	Polynomial differentiation

Cell Array

- A *cell array* is a data type with indexed data containers called *cells*, where each cell can contain any type of data. Cell arrays commonly contain either lists of text, combinations of text and numbers, or numeric arrays of different sizes. Refer to sets of cells by enclosing indices in smooth parentheses, (). Access the contents of cells by indexing with curly braces, {}.

Syntax

C = cell(n)

C = cell(sz1,...,szN)

C = cell(sz)

D = cell(obj)

Description

`C = cell(n)` returns an `n`-by-`n` cell array of empty matrices.

`C = cell(sz1,...,szN)` returns a `sz1`-by-...-by-`szN` cell array of empty matrices where `sz1,...,szN` indicate the size of each dimension. For example, `cell(2,3)` returns a 2-by-3 cell array.

`C = cell(sz)` returns a cell array of empty matrices where size vector `sz` defines `size(C)`. For example, `cell([2 3])` returns a 2-by-3 cell array.

Structure Array

- A *structure array* is a data type that groups related data using data containers called *fields*. Each field can contain any type of data. Access data in a field using dot notation of the form structName.fieldName.

Syntax

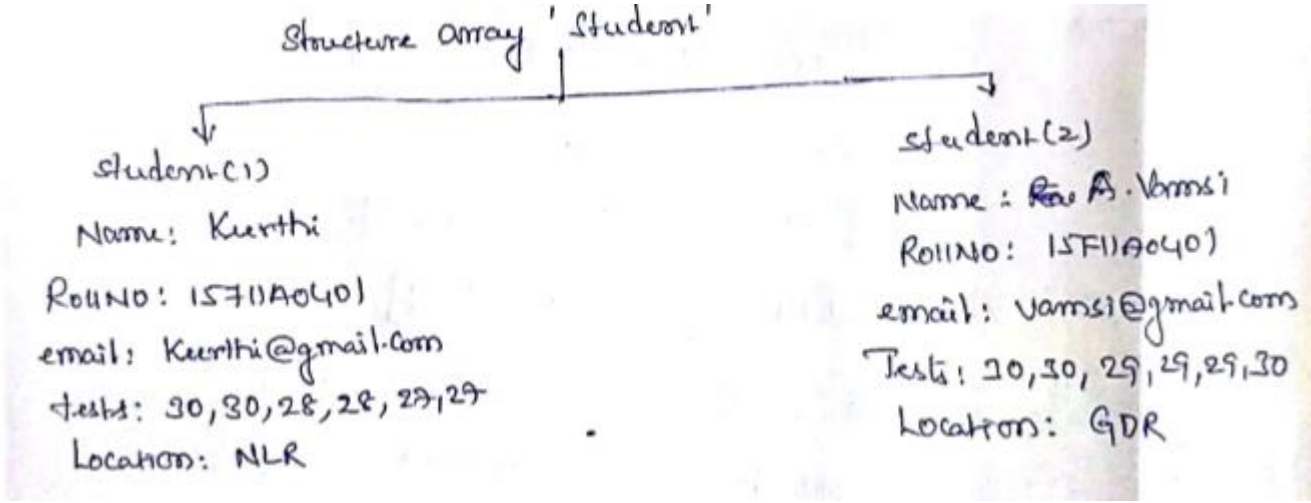
s = struct

s = struct(field,value)

s = struct(field1,value1,...,fieldN,valueN)

s = struct([])

s = struct(obj)



Description

`s = struct` creates a scalar (1-by-1) structure with no fields.

`s = struct(field,value)` creates a structure array with the specified field and value. The `value` input argument can be any data type, such as a numeric, logical, character, or cell array.

- If `value` is *not* a cell array, or if `value` is a scalar cell array, then `s` is a scalar structure. For instance, `s = struct('a',[1 2 3])` creates a 1-by-1 structure, where `s.a = [1 2 3]`.
- If `value` is a nonscalar cell array, then `s` is a structure array with the same dimensions as `value`. Each element of `s` contains the corresponding element of `value`. For example, `s = struct('x',{'a','b'})` returns `s(1).x = 'a'` and `s(2).x = 'b'`.
- If `value` is an empty cell array `{}`, then `s` is an empty (0-by-0) structure.

`s = struct(field1,value1,...,fieldN,valueN)` creates a structure array with multiple fields.

- If none of the `value` inputs are cell arrays, or if all `value` inputs that are cell arrays are scalars, then `s` is a scalar structure.
- If any of the `value` inputs is a nonscalar cell array, then `s` has the same dimensions as that cell array. Also, if two or more `value` inputs are nonscalar cell arrays, then they all must have the same dimensions.

  For any `value` that is a scalar cell array or an array of any other data type, `struct` inserts the contents of `value` in the relevant field for all elements of `s`. For example, `s = struct('x',{'a','b'},'y','c')` returns `s(1).x = 'a'`, `s(2).x = 'b'`, `s(1).y = 'c'`, and `s(2).y = 'c'`.
- If any `value` input is an empty cell array, `{}`, then output `s` is an empty (0-by-0) structure. To specify an empty field and keep the values of the other fields, use `[]` as a `value` input instead.

`s = struct([])` creates an empty (0-by-0) structure with no fields.

`s = struct(obj)` creates a scalar structure with field names and values that correspond to properties of `obj`. The `struct` function does not convert `obj`, but rather creates `s` as a new structure. This structure does not retain the class information, so private, protected, and hidden properties become public fields in `s`. The `struct` function issues a warning when you use this syntax.

Unit-3

FUNCTIONS AND FILES

FUNCTION :

- DEFINITION:

➢ A Function is a group of statements that together perform a task.

➢In MATLAB, functions are defined in separate files.

➢The file and of the function name should be the Same.

Elementary Mathematical Functions:

- Elementary Graphical Functions
- Elementary Number Functions
- Elementary Exponential and Logarithmic Functions
- Elementary Trigonometric Functions
- Elementary Hyperbolic Functions
- Elementary Combinational Functions

Elementary Graphical Functions:

- In MATLAB, the Syntax to represent graphical Representation is

  plot()

- If the function is y=f(x), then the graphical MATLAB command used is

  plot(x,y)

where x= independent variable and

y=dependent variable

Example: graphical representation of absolute values of a vector x is

```
x=[-1:0.01:1]
plot(x,abs(x))
```

- The MATLAB command to Plot User defined functions is

  fplot()

Example: fplot(f,xlim) for y=sin|x| is plotted in the range of $-2\pi<x<2\pi$

Elementary Number Functions:

The elementary number functions are as follows:

- Abs()-Absolute values of a real number or magnitude of a complex number

ex: >> abs([-1.4 0.0 1.4]) // Absolute values of a real number

ans= 1.4000

>>abs(4+3j) // magnitude of a complex number

ans=5

- Ceil()-Ceiling function; Rounds up a floating point number

ex: >>ceil(-2.1) // rounds up

ans=-2

>>ceil(2.1) // rounds up

ans=3

- Floor()-Rounds down floating poin

ex: >>floor(-2.5) //rounds down

ans=-3

>>floor(2.5) // rounds down

ans=2

- Fix()-Rounds a floating point number towards zero

ex: >>fix(4.534) ans=4

Continued Elementary Number Functions

- Mod()-Arithmatic reminder function

  ex: >> mod(7.2,3.1) //modulus value

  ans= 1.0000

- Rat()- Rational approximation f a Floating point number

  ex: >>[n,d]=rat(0.777) //rational number

  n=777; d=1,000

- Round()- Rounds to nearest integer

  ex: >> round([2.23 3.61 1.33]) // rounds up or rounds down

  ans= 2 4 1

- Sign()-sign function

  ex: >> sign(-3.2 0.0 5.4) //sign of the value

  ans= -1 0 1

- Sqrt()-square root function

  ex: >> sqrt(16) //square root of 16

  ans= 4

Continued Elementary Number Functions

➢Commands for Complex number functions:

- abs() - absolute value and complex magnitude

ex: if x = [1.3 -3.56 8.23 -5 -0.01] then y=abs(x)

y =1.3000 3.5600 8.2300 5.0000 0.0100

- angle() - phase angle

ex: if z= 2*exp(i*0.5) then y=angle(z)

y =0.5000

- complex() - create complex array

ex: y=complex(3,4) then y=3.0000+i4.0000

y=complex(a, b) where a=[1;2] and b=[2;4] then y=2 *x* 1 column vector

1+2i

2+4i

➢Commands for Complex number functions:

- conj() - complex conjugate

  ex: if Z = 2+3i then y=conj(z)

  y = 2.0000-3.0000i

- imag() - imaginary part of complex number

  ex: if Z = 2+3i then y=imag(z)

  y = 3.0000

  if Z = [0.5i 1+3i -2.2]; Y = imag(Z)

  Y = *1×3*

  0.5000 3.0000 0

- Real() - Real part of complex number

  ex: if Z = 2+3i then y=real(z)

  y = 2.0000

  if Z = [0.5i 1+3i -2.2]; Y = real(Z)

  Y = *1×3*

  0 1.0000 -2.2000

Continued Elementary Number Functions

➢Commands for Arithmetic functions:

- rem() - remainder after division
- sum() - sum of array elements
- prod() - product of array elements
- cumprod() -cumulative product of array elements

Exponential and Logarithmic function:

- The Exponential function ‘exp’ returns the  for a real number of x where ‘e’ constitutes of natural Logarithm.
- The value of ‘e’ is calculate as

$\exp(1)=2.7182818=2.7183$

ex: >>exp([-1 0 1 2])

ans: 0.3679 1.0000 2.7183 7.3891

- The exponential function applied to a complex number can be easily interpreted as follows:

if z=x+iy then

$\exp(z)=e^{z}=e^{(x+iy)}= e^{x}\, e^{iy}= e^{x}(\cos y+i\sin y)= e^{x}\cos y+ ie^{x}\sin y$

Continued Exponential and Logarithmic function:

- Inverse function to the exponential function is Natural Logarithm

  >> ln(x)

- Properties:
  - If y=ln(x) then x=exp(y)
  - ln(exp(x))=x
  - Exp(ln(x))=x
- Example: >>ln([1 2 3])

  ans=0 0.6931 1.0986

Continued Exponential and Logarithmic function:

➢Command used in exponential and Logarithm functions:

exp	Exponential
expm1	Compute exp(x)-1 accurately for small values of x
log	Natural logarithm
log10	Common logarithm (base 10)
log1p	Compute log(1+x) accurately for small values of x
log2	Base 2 logarithm and floating-point number dissection
nextpow2	Exponent of next higher power of 2
nthroot	Real nth root of real numbers
pow2	Base 2 power and scale floating-point numbers
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays
sqrt	Square root
Logm()-	Matrix Logarithm
sqrtm()	Matrix square root
Expm()	Matrix exponential
Funm(A, fun)	Evaluates general matrixfunctions funm(A,@log)

Elementary Trigonometric function:

Function	Description
sin	Sine of the input
cos	Cosine of the input
tan	Tangent of the input
asin	Inverse sine of the input
acos	Inverse cosine of the input
atan	Inverse tangent of the input
atan2	Four-quadrant inverse tangent of the input
sinh	Hyperbolic sine of the input
cosh	Hyperbolic cosine of the input
tanh	Hyperbolic tangent of the input
asinh	Inverse hyperbolic sine of the input
acosh	Inverse hyperbolic cosine of the input
atanh	Inverse hyperbolic tangent of the input

D = rad2deg(R) ex: rad2deg(π)=180

R= deg2rad (D) ex: deg2rad(180)=π

Elementary Hyperbolic Functions

Hyperbolic Functions

$\sinh(x)=\frac{e^x-e^{-x}}{2}$	$\operatorname{csch}(x)=\frac{1}{\sinh(x)}=\frac{2}{e^x-e^{-x}}, x\neq 0$
$\cosh(x)=\frac{e^x+e^{-x}}{2}$	$\operatorname{sech}(x)=\frac{1}{\cosh(x)}=\frac{2}{e^x+e^{-x}}$
$\tanh(x)=\frac{\sinh(x)}{\cosh(x)}=\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$\coth(x)=\frac{1}{\tanh(x)}=\frac{e^x+e^{-x}}{e^x-e^{-x}}, x\neq 0$

$$\sinh^{-1}x = \ln\left(x+\sqrt{x^2+1}\right)$$

$$\cosh^{-1}x = \ln\left(x+\sqrt{x^2-1}\right)$$

$$\tanh^{-1}x = \frac{1}{2}\ln\left(\frac{1-x}{1+x}\right), \quad |x|<1$$

$$\coth^{-1}x = \frac{1}{2}\ln\left(\frac{1+x}{1-x}\right), \quad |x|>1$$

$$\operatorname{sech}^{-1}x = \cosh^{-1}\left(\frac{1}{x}\right)$$

$$\operatorname{csch}^{-1}x = \sinh^{-1}\left(\frac{1}{x}\right)$$

Elementary Combinational Functions

- Combinational functions is nothing but combining two or more elementary functions

  ex: abs(sin30)- it is a combination of number and trigonometric functions.

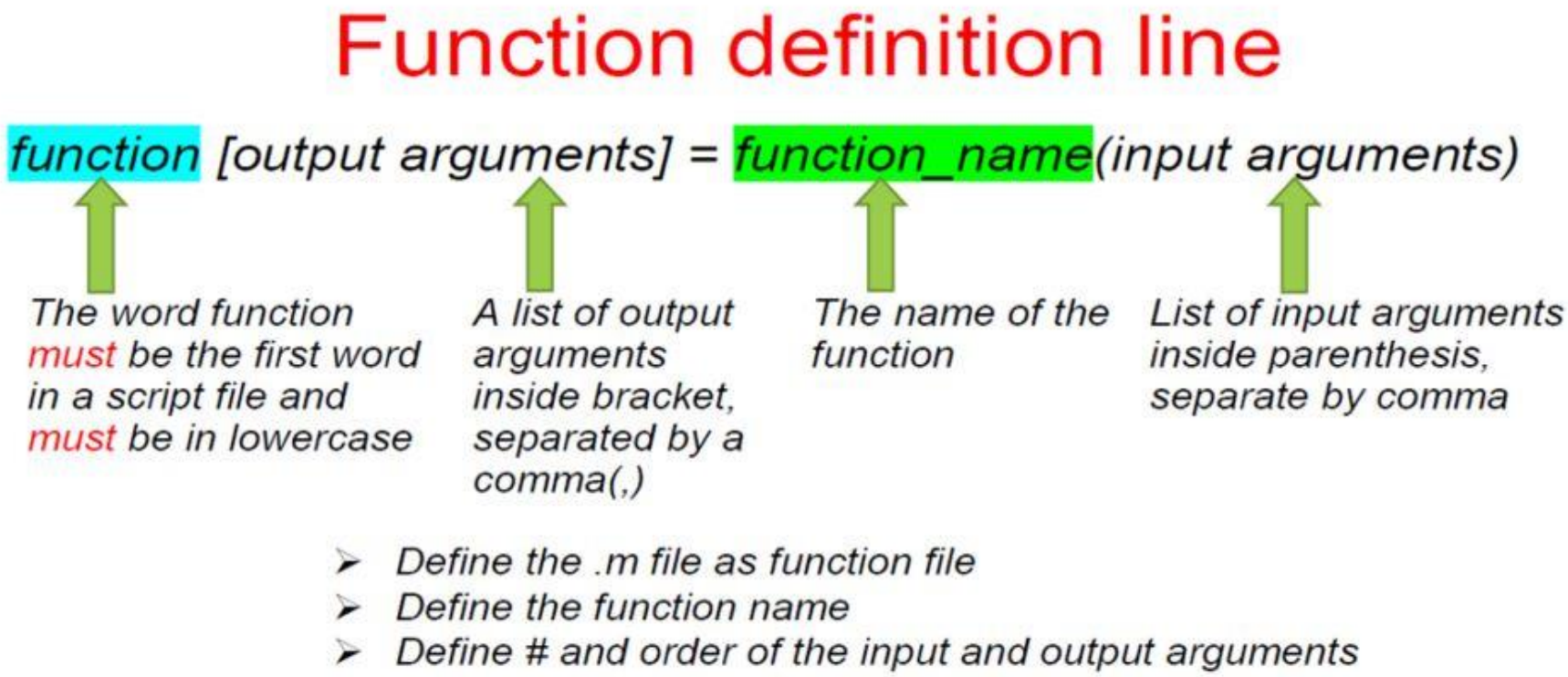
- Special functions:

- Besselh(): Bessel function of third kind
- Beta(): beta function
- Erf(): Error function

User-defined function:

- When you use a mathematical function f(x) many times for different variables in a MATLAB program, a user-defined function is beneficial.
- A function file is a .m file, but different from a script file.
- Scripts are the simplest type of program since they store commands exactly as you would type them at the command line.
- However, .m functions are more flexible and more easily extensible.
- Instead of manually updating the script each time, you can make your program more flexible by changing it to a .m function.

Syntax of User defined functions:



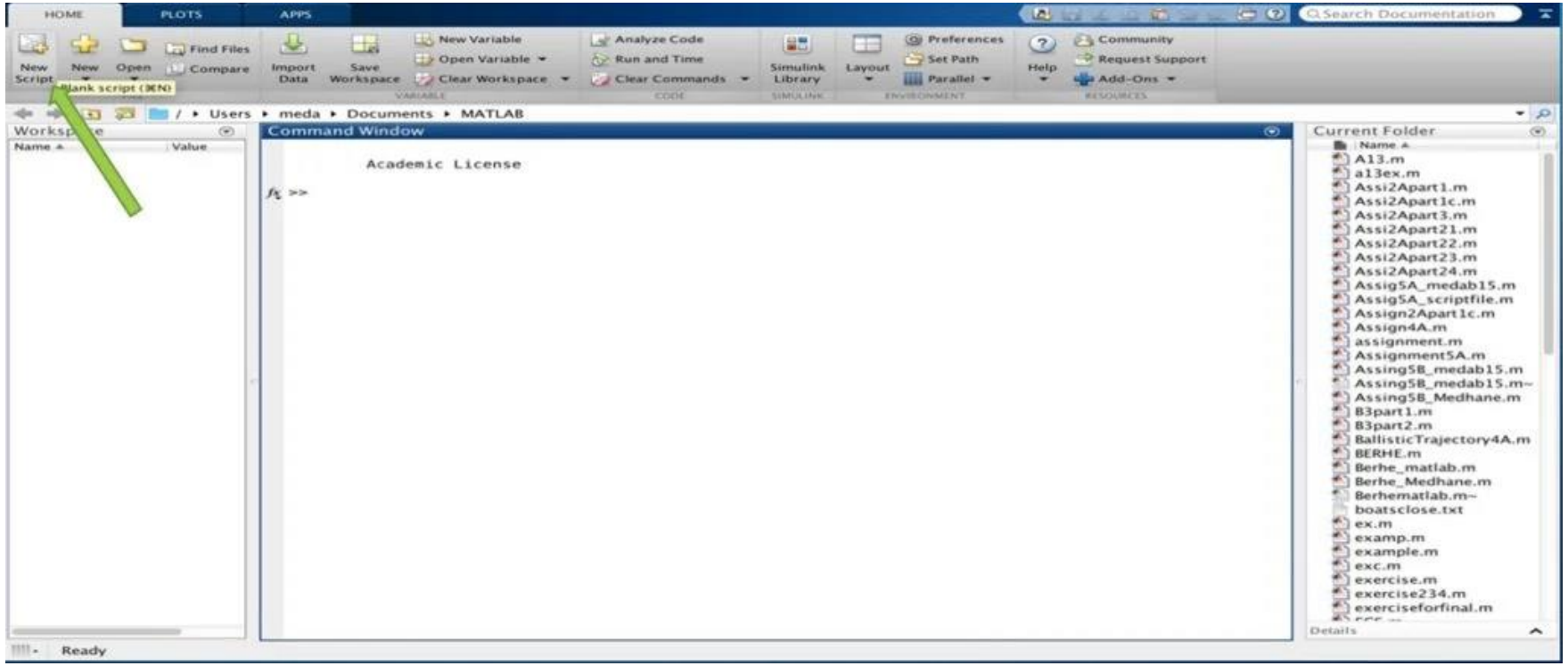
The Function name must be the same as file name i.e.,function_name.m

ex: function[area]=square(side)

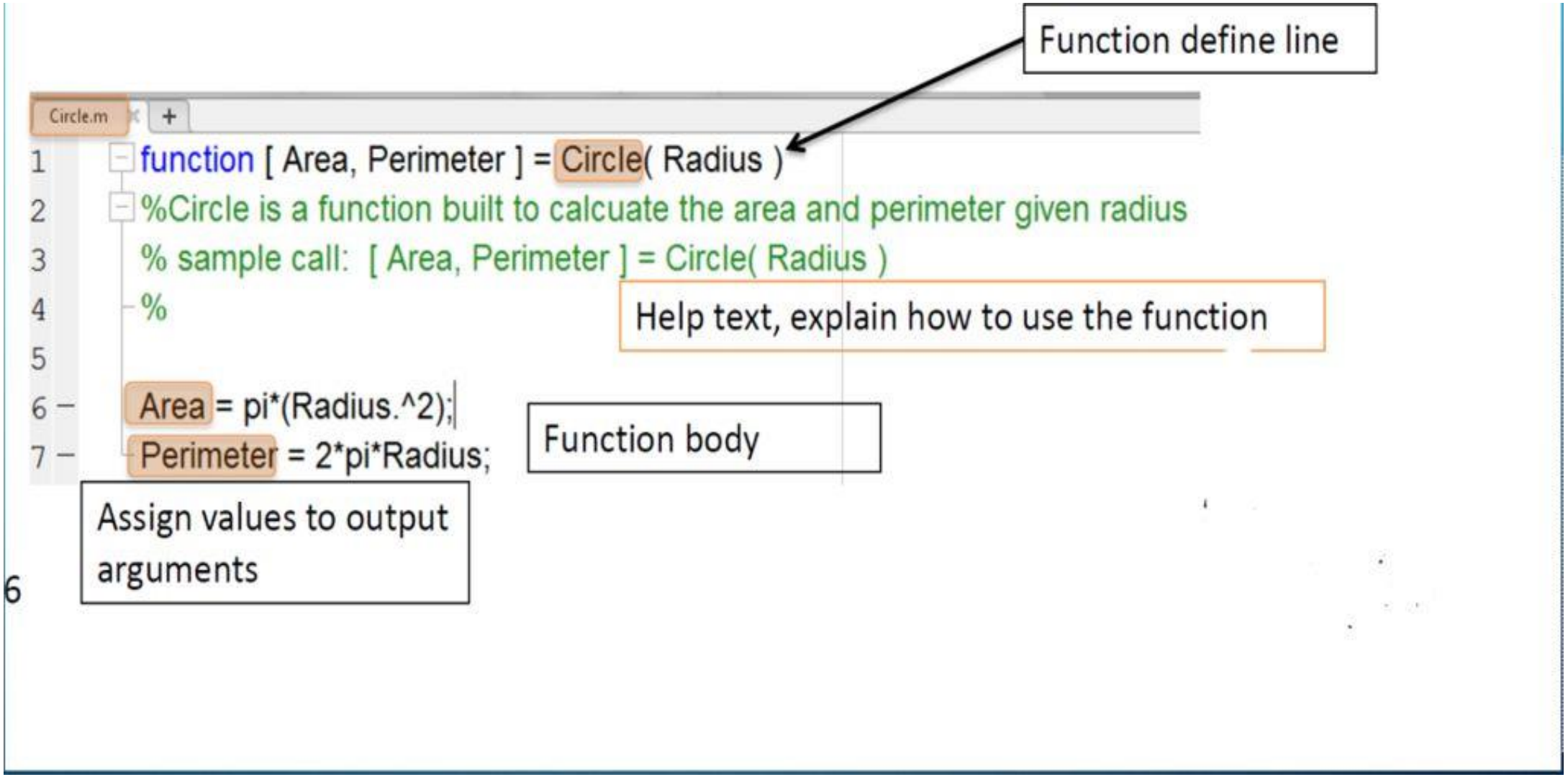
Save this file as its function name i.e., square.m

Steps to create User defined function:

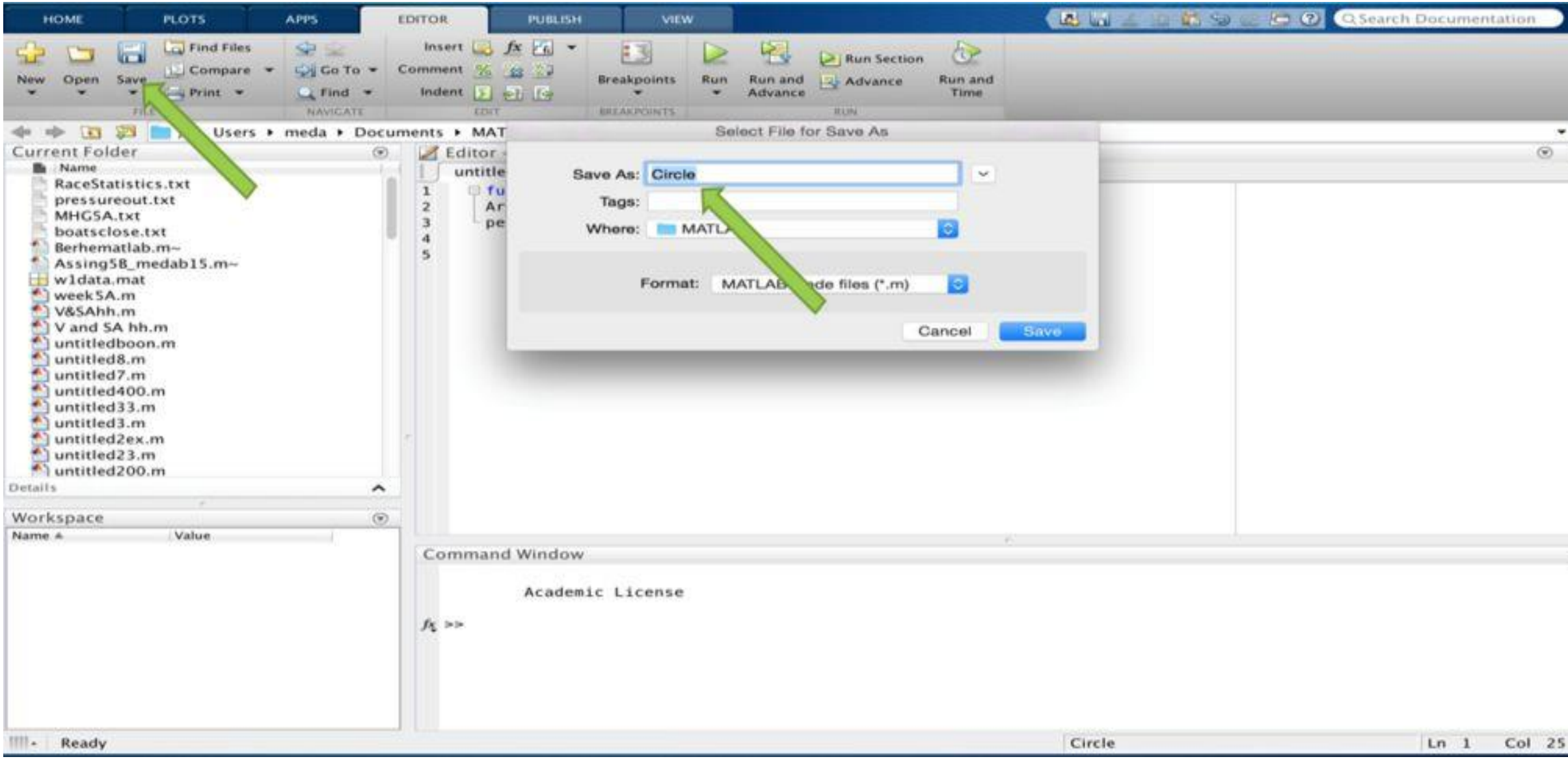
- 1) Open the MATLAB software on your computer.
- 2) Once you open MATLAB, open a new script file by double clicking the “new script” icon on the top left of the MATLAB file, as in the picture above.



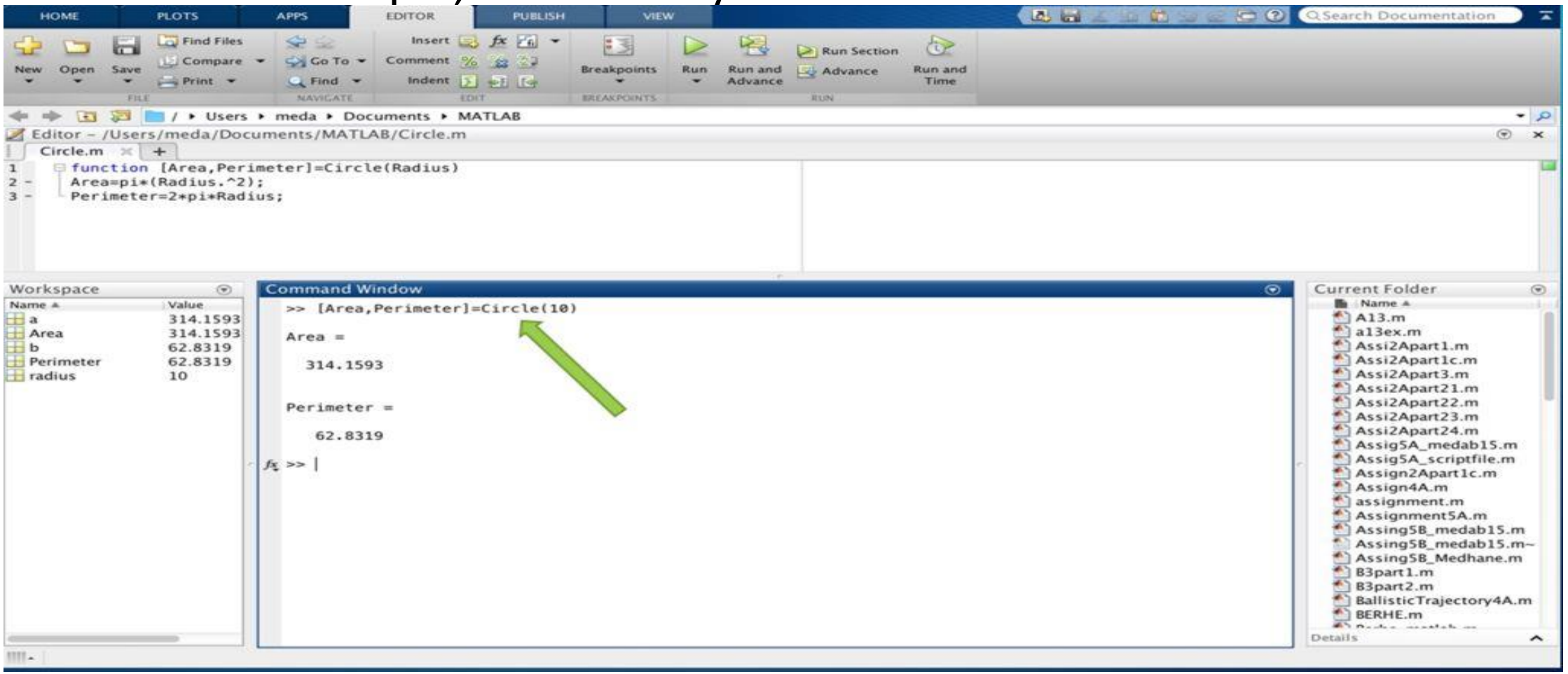
- 3) Writing Your Function in a Script File(area and perimeter of circle)



- 4)save the function in the name of your function



- 5) successfully created your first user-defined MATLAB function.
- 6) Once you run the program by providing input at command window
- As an example, I choose my radius to be 10. In the command



Examples of format of functional file:

- Note: you can give any names for the output but you can't change the function name.

Function Definition Line	File Name
1) function[area_square]=square(radius)	square.m
2) Function area_square =square(radius)	square.m
3) function[volume_Box]=box(height, width, length)	box.m
4) function[area, circuf]= circle(radius)	circle.m

Examples to define and call the user define function

➢Example :1

In editor window:

```
%function definition
Function[y1,y2,y3]=myfun(x)
Y1=x*10;
Y2=x.^2;
Y3=sqrt(x);
% save this file with the name “myfun”
```

In command window:

```
% function call
[y1,y2,y3]=myfun(4)
% output
Y1=40
Y2=16
Y3=2
```

In command window

```
%function call
[y1,y2,y3]=myfun(4:2:10)
%output
Y1=40    60    80    100
Y2=16    36    64    100
Y3=2     2.44  2.82  3.16
```

➢Example :2 (MATLAB program to find area of triangle, square and rectangle)

In editor window:

```
%function definition
Function[t,s,r]=tsrarea(b,h)
t=0.5*b*h;
s=b*b;
r=b*h;
% save this file with the name “tsrarea”
```

In command window:

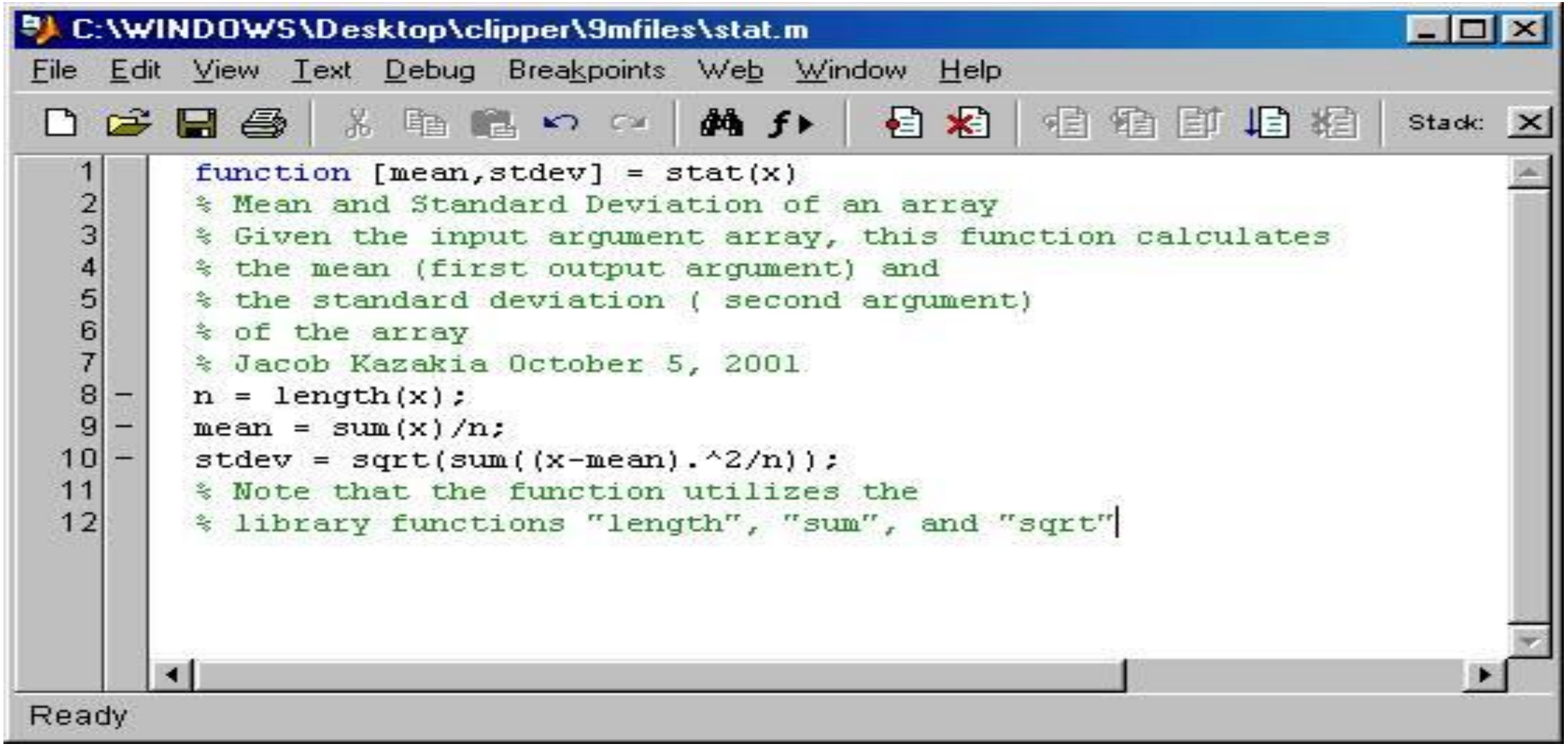
```
% function call
[t,s,r]= tsrarea(2,3)
% output
t=1.56
s=4
r=6
```

In command window

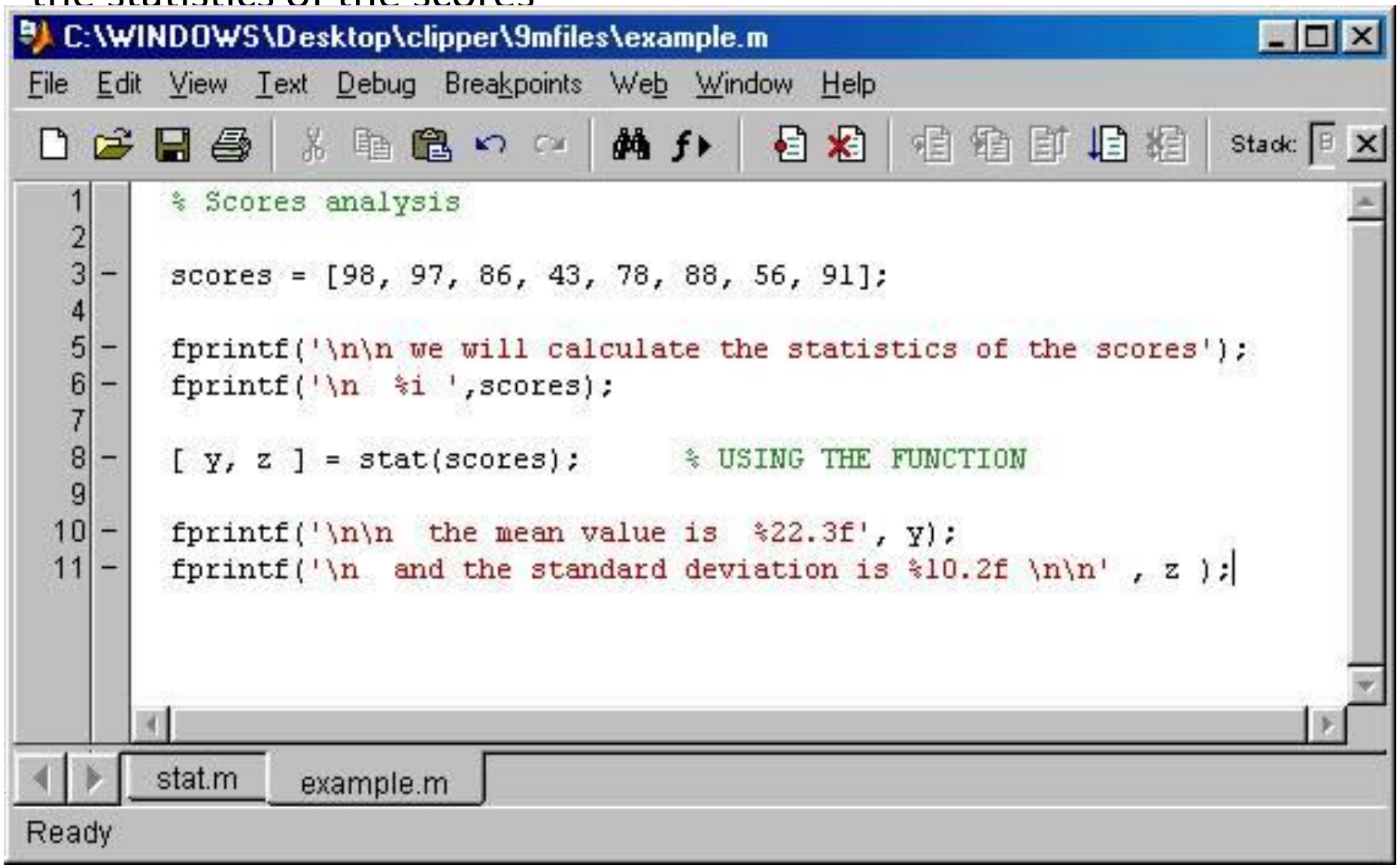
```
%function call
[t,s,r]= tsrarea([4:6],[1:3])
%output
t=2     5     9
s=16    25    36
r=4     10    18
```

➢Example :3 (calculating mean and standard deviation)

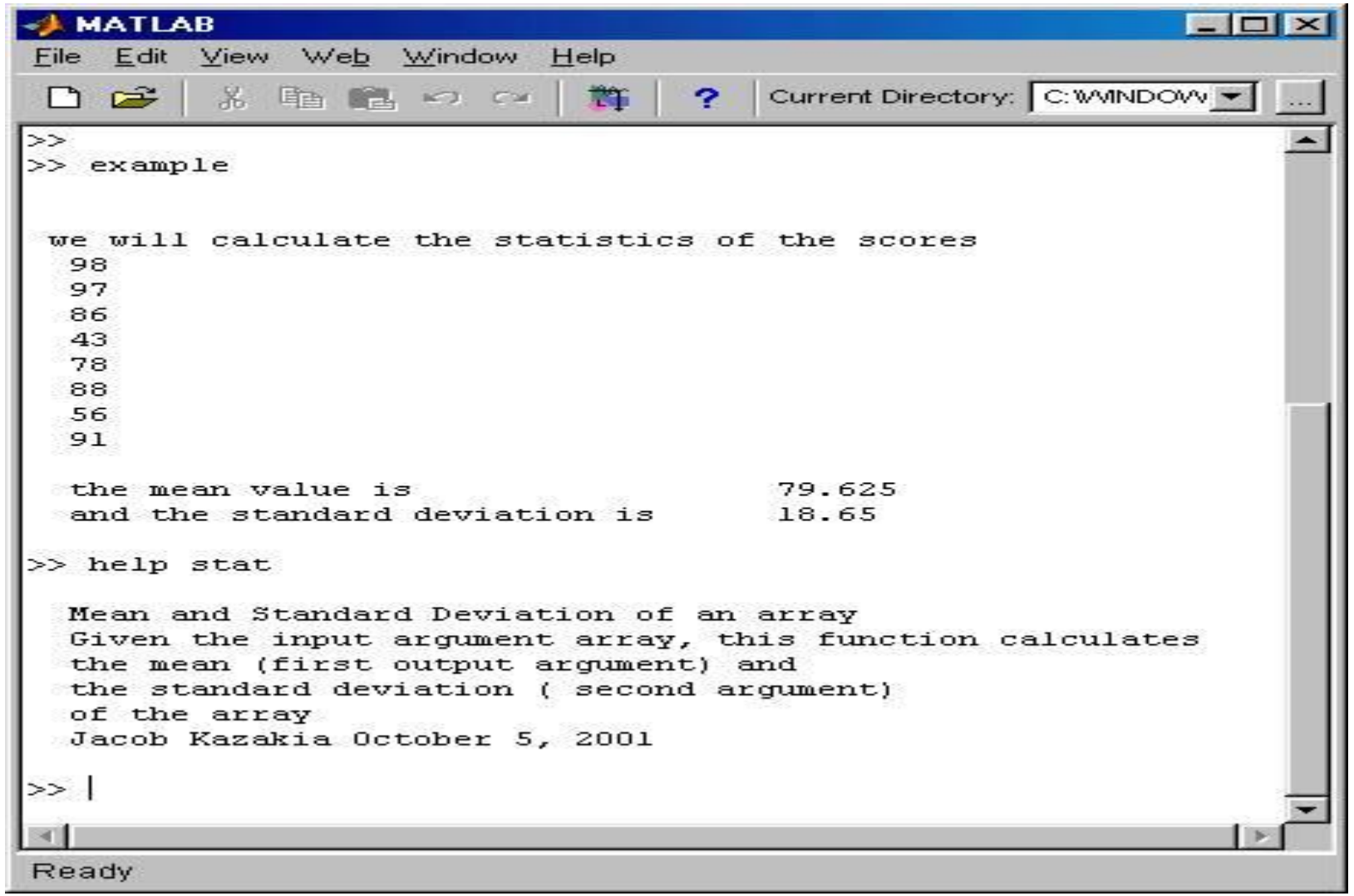
Save the program by its file name stat.m



➢Using User defined function in another program: We can now use this function by simply invoking its name. The use of it is illustrated in the following picture of a script called , **example.m** where we define an array of scores and then use the function to calculate the statistics of the scores



➢ Run the example Program: The command window illustrates the results obtained when we type the name of the file example at the MATLAB command line.



Example:4 (Program Selection Sort:)

Editor - C:\Users\hung\Documents\MATLAB\ssort.m

ssort.m

```
function X=ssort(X, c)
% ssort: selection sort
% X is an arbitrary array
% c is either 'a' for ascending
%   or 'd' for descending

n=length(X);
for i=1:n-1
    m=i;
    for j=i+1:n
        % select the target
        if (c=='a'&& X(j)<X(m)).
        || (c=='d'&& X(j)>X(m))
            m=j;
        end
    end

    % swap data
    if m~=i
        tmp=X(i);
        X(i)=X(m);
        X(m)=tmp;
    end
end
```

ssort Ln 1 Col 23

Command Window

```
>> edit ssort
>> a=[20 38 33 9 2 15 29]
a =
    20    38    33     9     2    15    29
>> b='Fibonacci'
b =
Fibonacci
>> ssort(a,'a')
ans =
     2     9    15    20    29    33    38
>> ssort(a,'d')
ans =
    38    33    29    20    15     9     2
>> ssort(b,'a')
ans =
Fabcciino
>> ssort(b,'d')
ans =
oniiccbaF
>>
```

Workspace Current Folder

Name	Value	Size	Bytes	Class
a	[20,38,33,9,2,15,29]	1x7	56	double
ans	'oniiccbaF'	1x9	18	char
b	'Fibonacci'	1x9	18	char

- F i b o n a c c i F a b c c i i n o ascending order
- F i b o n a c c i o n i i c c b a F descending order

Editor window:

```
a=input('a=');
For i=1:length(a)
        j=1:length(a)
            if(a(i)>a(j))
                c=a(i);
                a(i)=a(j);
                a(j)=c;
End;
End;
End;
Disp(a);
```

Local variable:

- The names of the input variables given in the function-definition line are local to that function. This means that other variable names can be used when you call the function.
- All variables inside a function are erased after the function finishes executing, except when same variable names appear in the output variable list used in the function call
- Example:

In editor window:

```
%function definition
function (dist,vel) = drop(a,vo,t);
vel = a*t + vo;
dist = 0.5*a*t.^2 + vo*t;
% save this file with the name “drop”
```

In command window:

```
% function call
[feet_dropped, speed] = drop(32.2,10,5)
                    OR
[feet_dropped,speed]=drop(32.2,10, (0:1:5))
```

Dist=feet_drop
Vel=speed

Global Variables:

- The global command declares certain variables global, and therefore their values are available to the basic workspace and to other functions that declare these variables global.
- The syntax to declare the variables a, x, and q is

```
global a x q.
```

- If the global variable doesn't exist the first time you issue the global statement, it will be initialized to the empty matrix.

```
global [ ]
```

- If a variable with the same name as the global variable already exists in the current workspace, MATLAB issues a warning and changes the value of that variable to match the global.
- In a user-defined function, make the global command the first executable line.

Advanced Functions

- In MATLAB we have some advance functions they are:

1. Anonymous Function
2. Inline Function
3. Sub Function
4. Nested Function

1. Anonymous Function

- Anonymous function enables to create a simple function without creating M-file in it.
- Construct Anonymous function by using either command window or any other functions or script.
- Syntax:

  function_handle=@(argument list) expression

  where

  arguments list separated by using commas

  Expression is a single valid MATLAB expression

  function_handle enables to invoke the functions

  @ is called the function handled

Example:

% Function definition:

```
sqr = @(x) x.^2;
```

where Variable sqr is a function handle.
The @ operator creates the handle, and
the parentheses () immediately after the @ operator include the function input arguments.
This anonymous function accepts a single input x, and implicitly returns a single output, an array the same size as x that contains the squared values.

Calling o a function: Find the square of a particular value (5) by passing the value to the function handle, just as you would pass an input argument to a standard function.

% Function call:

```
a = sqr(5)
a =
    25
```

% Function call:

```
a = sqr([4 5])
a =
    16   25
```

Variables in the Expression

- Function handles can store not only an expression, but also variables that the expression requires for evaluation.

For example,

%create a handle to an anonymous function that requires coefficients a, b, and c.

```
a = 1.3;
 b = .2;
c = 30;
 parabola = @(x) a*x.^2 + b*x + c;
```

```
a = -3.9;
 b = 52;
 c = 0;
parabola = @(x) a*x.^2 + b*x + c;
```

% Function call: (The values persist within the function handle even if you clear the variables)

```
clear a b c
 x = 1;
 y = parabola(x)
y = 31.5000
```

```
x = 1;
 y = parabola(1)
y = 48.1000
```

Note: 1)To supply different values for the coefficients, you must create a new function handle

2) You can save function handles and their associated values in a MAT-file and load them in a subsequent MATLAB session using the save and load functions, such as save myfile.mat parabola

Ways and types calling Anonymous functions:

1) Multiple input argument: Create Anonymous function with more than one input variable

Example 1: write a MATLAB code to the function myfunc=x^2+y^2+xy

```
%function definition
myfunc =@(x,y) x.^2+y.^2+x*y;
%calling a function
z=myfunc(1,10)                    or        z=myfunc(1:2,2:3)
z=                                          z=
     111                                        7      19
```

Example 1: write a MATLAB code to the function z=ax+by

```
%function definition                        %calling a function
>> a=1;b=2                                  >>z(2,3)
>>z =@(x,y) a*x+b*y;                       ans=
                                                  8
```

2) No input argument: If a function does not require any inputs, use empty parentheses when you define and call the anonymous function. For example:

```
% definition:
>>t = @() date;
%calling
>>d = t()
d =
22-Apr-2020
```

Omitting the parentheses in the assignment statement creates another function handle, and does not execute the function:

```
%calling
>>d = t
                d =
d = @() date
```

3)Calling one function within another function:

One argument function can call another function to implement function composition

- For example: f(x)=sqrt(x) and g(x)= $2X^3$

```
% function definition:
>>f = @(x) sqrt(x);
>>g=@(x) 2*x.^3;
>>z=@(x) g(f(x));
%calling of function
>>z(4)
ans = 16
```

Advantages:

- In MATLAB, by using anonymous functions we can easily implement complex mathematical quadratic equations into simple form.
- Speed of execution is high.
- There no such defined name to these functions we can give any name at the time of function definition.
- The anonymous function saves memory as well as supports reusability property, therefore, no need to write big and complex expression again and again.

Inline functions:

- The `inline` command lets you create a function of any number of variables by giving a string containing the function followed by a series of strings denoting the order of the input variables.
- It is similar to creating MATLB: Anonymous function with some significant differences.
- Syntax:

  F=inline('function formula')

- Ex: $f(x)=x^2 \sin x$ coded as

```
%Function  definition
f=inline('x.^2*sin(x)');
% function calling
C=f(3)
C=
        3.63
```

- This method is good for relatively simple functions that will not be used that often and that can be written in a single expression.

Sub function

- Each function file contains a required primary function that appears first and any number of optional sub-functions that comes after the primary function and used by it.
- Primary functions can be called from outside of the file that defines them, either from command line or from other functions, but sub-functions cannot be called from command line or other functions, outside the function file.
- The use of sub function enables to reduce the number of files that define in the your function
- Example :

In editor window:

```
%function definition
function b = myfun(a)
        b = squareMe(a)+doubleMe(a);
end
function y = squareMe(x)    %sub function
        y = x.^2;
end
function y = doubleMe(x)    %sub function
        y = x.*2;
end
% save this file with the name "myfun"
```

In command window:

```
% function call
z=myfun(4)
% output
   Z=
        24
```

Nested function

- A **Nested function** is a **function** that is completely contained within a parent **function**.
- Any **function** in a **program** file can include a **nested function**. The primary difference between **nested functions** and other types of **functions** is that they can access and modify variables that are defined in their parent **functions.**
- Example:

```
% Function defition
 function p = makeParabola(a,b,c)
 p = @parabola;
function y = parabola(x)      %nested function
 y = a*x.^2 + b*x + c;
end
end
%calling function
p = makeParabola(1.3,.2,30);
X = 25;
Y = p(X)

Y =
          847.5000
```

Visibility of Nested Functions

Every function has a certain *scope*, that is, a set of other functions to which it is visible. A nested function is available:

- From the level immediately above it. (In the following code, function `A` can call `B` or `D`, but not `C` or `E`.)
- From a function nested at the same level within the same parent function. (Function `B` can call `D`, and `D` can call `B`.)
- From a function at any lower level. (Function `C` can call `B` or `D`, but not `E`.)

```
function A(x, y)                    % Main function
 B(x,y)
D(y)
        function B(x,y)             % Nested in A
        C(x)
        D(y)
                function C(x)       % Nested in B
                D(x)
                end
        end
        function D(x)               % Nested in A
        E(x)
                function E(x)       % Nested in D
                disp(x)
                end
        end
end
```

Working with Data Files:

- The MATLAB environment instead of editing(working with ASCII data or binary data), it provides many ways to bring data created by other applications into the MATLAB workspace, a process called importing data, and to package workspace variables so that they can be used by other applications, a process called exporting data. instead of editing(working with ASCII data or binary data).
- To make importing data easier, both ASCII and binary, MATLAB includes a graphical user interface, called the Import Wizard, that leads you through the import process

- **Importing Spreadsheet Files**
  Some spreadsheet programs store data in the . wk1 format. You can use the command M= wk1 read (‘f ile name ‘) to import this data into MATLAB and store it in the matrix M.
- The command A =xlsread{‘ file name’) imports the Microsoft Excel workbook file filename. xls into the array A.
- The command [A, B]= xlsread (‘file name’) imports all numeric data into the array A and all text data into the cell array B.

The Import Wizard

- **Importing ASCII Data**
- **Importing Binary Data Files(include MAT-files)**
- **Exporting Delimited ASCII Data Files(**»dlmwrite (‘my_data. out’ ,A, ‘;’)**)**

UNIT-IV

PROGRAMMING TECHNIQUES

DESIGN AND DEVELOPEMENT

- The design and development of a programs is easier in the MATLAB compare to any other Languages.
- Program design and program development are two consequence operations and tasks to develop MATLAB or any other programming language.
- Program design is a initial and vital task in program development
- A symmetrical hierarchical planning of operation required to perform the program development.

continued…

- The step by step procedural algorithm is called program plan or program design of a program under development.
- According to the sequence of steps described in the program design, program coding will be carried out in the prescribed sequential order.
- After completing the program design, each and every design step of the program design will be transformed into its equivalent MATLAB code or instructions.
- This process of writing equivalent MATLAB code or instructions to each and every step of the program design is called as “Program Development”

Example:

Ex: Write program design and development for generating the Fibonacci series.

Program design:

Step1: Start
Step2: Read the length of the series
Step3: Define initial elements and empty Fibonacci vector.
Step4: Store initial elements into the Fibonacci vector
Step5: Add previous two elements and store the result as the next element in the series.
Step6: Repeat Step 5 for length-2 times
Step7: Display the Fibonacci vector.

Program development:

```
clc;
clear all;
close all;
n=input('length of the series:');
a=0;
b=1;
g=[];
g=[a,b];
For i=1:n-2
    c=a+b;
    g(i+2)=c;
      a=g(i);
      b=g(i+1);
end
fprintf('Generated Fibonacci series:%f',g);
```

Relational operators

Relational operators compare the elements in two arrays and return logical true or false values to indicate where the relation holds. For more information, see Array Comparison with Relational Operators.

Functions

`==`	Determine equality
`>=`	Determine greater than or equal to
`>`	Determine greater than
`<=`	Determine less than or equal to
`<`	Determine less than
`~=`	Determine inequality
`isequal`	Determine array equality
`isequaln`	Determine array equality, treating NaN values as equal

Relational Functions

Symbol	Function Equivalent	Description
`<`	`lt`	Less than
`<=`	`le`	Less than or equal to
`>`	`gt`	Greater than
`>=`	`ge`	Greater than or equal to
`==`	`eq`	Equal to
`~=`	`ne`	Not equal to

Example for relational operators:

For example,
if you compare two matrices of the same size, then the result is a logical matrix of the same size with elements indicating where the relation is true.

```
A = [2 4 6; 8 10 12]
A =     2    4    6
        8   10   12
B = [5 5 5; 9 9 9]
B =     5    5    5
        9    9    9
A < B
ans =   1    1    0
        1    0    0
```

Logical operators:

Functions

`Short-circuit &&, ||`	Logical operations with short-circuiting
`&`	Find logical AND
`~`	Find logical NOT
`|`	Find logical OR
`xor`	Find logical exclusive-OR
`all`	Determine if all array elements are nonzero or `true`
`any`	Determine if any array elements are nonzero
`false`	Logical 0 (false)
`find`	Find indices and values of nonzero elements
`islogical`	Determine if input is logical array
`logical`	Convert numeric values to logicals
`true`	Logical 1 (true)

Inputs		and	or	xor	not
A	B	A&B	A|B	xor(A,B)	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Example for Logical operator:

- Write a program to find the largest number by using the logical operator:
- Program:

```
clc;
clear all;
close all;
a=input('enter the first number:');
b=input('enter the second number:');
c=input('enter the third number:');
if((a>b)&&(a>c))
fprintf('%f is the largest number ',a);
else if((b>a)&&(b>c))
fprintf('%f is the largest number ',b);
else
fprintf('%f is the largest number ',c);
end
end
```

Output:

```
enter the first number: 3
enter the second number: 5
enter the third number:2
5 is the largest number.
```

Bitwise Operations:

- MATLAB provides various functions for bit-wise operations like 'bitwise and', 'bitwise or' and 'bitwise not' operations, shift operation, etc.
- The following table shows the commonly used bitwise operations −

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

Example

Create a script file and type the following code −

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

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

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

Set Operations:

- MATLAB provides various functions for set operations, like union, intersection and testing for set membership, etc.
- The following table shows some commonly used set operations −

Sr.No.	Function & Description
1	**intersect(A,B)** Set intersection of two arrays; returns the values common to both A and B. The values returned are in sorted order.
2	**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.
3	**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).
4	**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).

Continued..

5	**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.
6	**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.
7	**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.
8	**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.
9	**setxor** Sets exclusive OR of two arrays
10	**union** Sets union of two arrays
11	**unique** Unique values in array

Example of set operation:

Example

Create a script file and type the following code −

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

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

```
a =

    7   23   14   15    9   12    8   24   35

b =

    2    5    7    8   14   16   25   35   27

u =

    2    5    7    8    9   12   14   15   16   23   24   25   27   35

i =

    7    8   14   35

s =

    9   12   15   23   24
```

Conditional statement:

- This group of control statements enables to select at run-time, which block of code is executed.
- In MATLAB there are two types of conditional statements

1) If statement
2) Switch statement

- To make this selection based on whether a statement is true or false, use the if statement (which may include else or elseif).
- To select from several possible options depending on the value of an expression, use the switch and case statements.

1) if conditional statement:

- The **if** is a conditional statement that provides the functionality to choose a block of code to execute at run time.
- A predefined condition is checked, and the remaining code executes based on the output of the condition.
- The **if** statement is defined by the **if** keyword.
- The control flows within the **if** block if the condition is found true.

➢The if conditional statements are classified as follows:

  - Simple if statement
  - if –else statement
  - if –else-if statement
  - if –else if –else statement
  - Nested if statement

➢Simple if statement:

The syntax for simple if statement is

```
if (condition (or) expression)
        Statements  (or) block of instructions
end
```

Example: finding even number:

```
clc;
clear all;
close all;
n=input('Enter the number:');
if (rem(n,2)==0)
        fprintf('%f is an even
number',n);
        b= n/2;
end
```

Simulation:

```
>> Enter the number:   4
    4.0000 is an even number
```

➢if- else conditional statement:

The syntax for if-else statement is

```
if (condition (or) expression)
        Statements  (or) block of instructions
else
        Statements  (or) block of instructions
end
```

Example: finding even number:

```
clc;
clear all;
close all;
n=input('Enter the number:');
if (rem(n,2)==0)
        fprintf('%f is an even number',n);
        b= n/2;
else
        fprintf('%f is an odd number',n);
end
```

Simulation:

```
>> Enter the number:   9
     9.0000 is an odd number
```

➢ if- elseif conditional statement:

The syntax for if-elseif statement is

```
if (condition)
        block of instructions
elseif(condition)
        block of instructions
end
```

Simulation:

```
enter the first number: 3
enter the second number: 5
enter the third number:2
5 is the largest number.
```

Example: finding largest number:

```
clc;
clear all;
close all;
a=input('enter the first number:');
b=input('enter the second number:');
c=input('enter the third number:');
if((a>b)&&(a>c))
    fprintf('%f is the largest number ',a);
else if((b>a)&&(b>c))
    fprintf('%f is the largest number ',b);
else if((c>a) && (c>b))
     fprintf('%f is the largest number ',c);
end
```

➢if- elseif –else conditional statement:

The syntax for if-elseif-else statement is

```
if (condition)
        block of instructions
elseif(condition)
        block of instructions
else
        block of instructions
end
```

Simulation:

enter the first number: 3
enter the second number: 5
enter the third number:2
5 is the largest number.

Example: finding largest number:

```
clc;
clear all;
close all;
a=input('enter the first number:');
b=input('enter the second number:');
c=input('enter the third number:');
if((a>b)&&(a>c))
    fprintf('%f is the largest number ',a);
else if((b>a)&&(b>c))
    fprintf('%f is the largest number ',b);
else
    fprintf('%f is the largest number ',c);
end
```

➢Nested if statement:

The syntax for Nested if statement is

```
if (condition)
        if (condition)
                if (condition)
                        block of instructions
                end
        end
end
```

Simulation:

enter the first number: 3
enter the second number: 5
enter the third number:2
5 is the largest number.

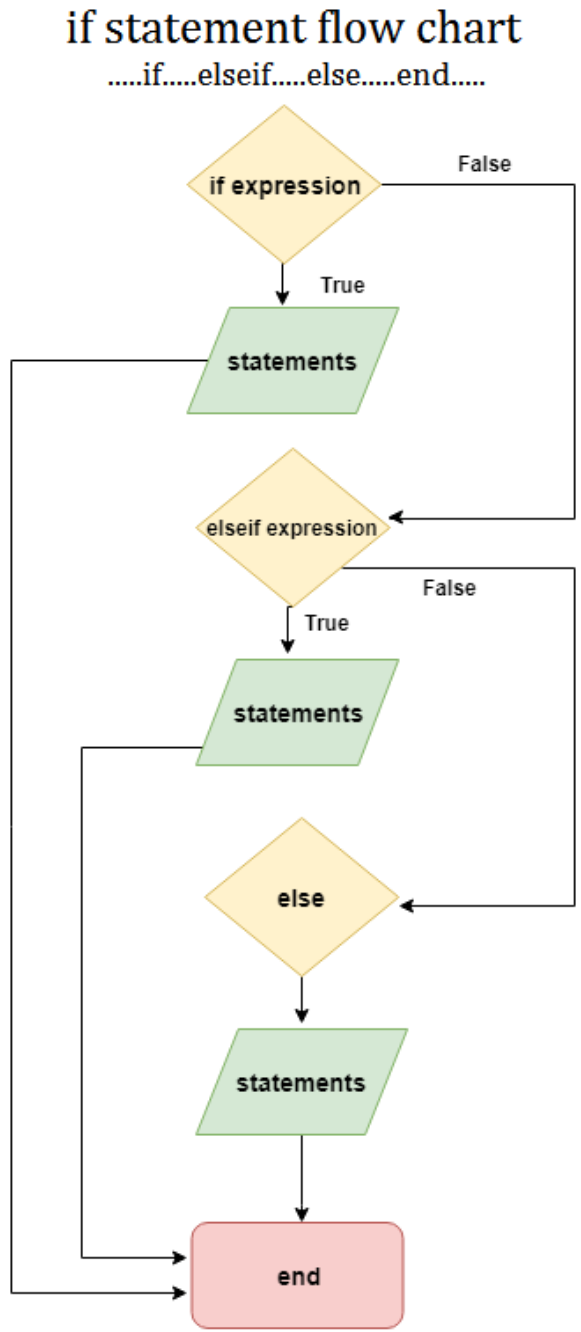
Example: finding largest number:

```
clc;
clear all;
close all;
a=input('enter the first number:');
b=input('enter the second number:');
c=input('enter the third number:');
if(a>b)
        if(a>c)
        fprintf('%f is the largest number ',a);
        end
end
if(b>a)
        if(b>c)
        fprintf('%f is the largest number ',b);
        end
end
if(c>a)
        if(c>b)
        fprintf('%f is the largest number ',c);
        end
end
```

Flow chart of if statement:

Following are the points while using if statement in MATLAB:

•Always close the if block with the **end** keyword.
•The **elseif** and **else** are optional, use them if there are more conditions to be checked.
•The **elseif** and **else** are an optional parts of the **if** statement and don't require additional **end** keyword.
•Remember not to use space in between **else** & **if** of the **elseif** keyword, because this leads to nested **if** statement and each if block needs to be close with the end keyword.
•There is no limit to use multiple **elseif** statements.
•We can nest the if statement by using it within another **if** statement



2) Switch conditional statement:

- The switch is another type of conditional statement and executes one of the group of several statements.
- If we want to test the equality against a pre-defined set of rules, then the switch statement can be an alternative of the if statement.

Syntax:

```
switch switch_expression(or) variable (or) condition
    case case_expression1
        Statements  (or) block of instructions
    case case_expression2
        Statements
                    "
                    "
    case case_expressionN
        Statements
    otherwise
        Statements
end
```

Example 1:

```
1.% program to check whether the entered number is a weekday or not
2.a = input('enter a number : ')
3.switch  a
4.    case 1
5.        disp('Monday')
6.    case 2
7.        disp('Tuesday')
8.    case 3
9.        disp('Wednesday')
10.    case 4
11.       disp('Thursday')
12.    case 5
13.       disp('Friday')
14.    case 6
15.       disp('Saturday')
16.    case 7
17.       disp('Sunday')
18.    otherwise
19.       disp('not a weekday')
20.end
```

Simulation:

>>enter a number: 4

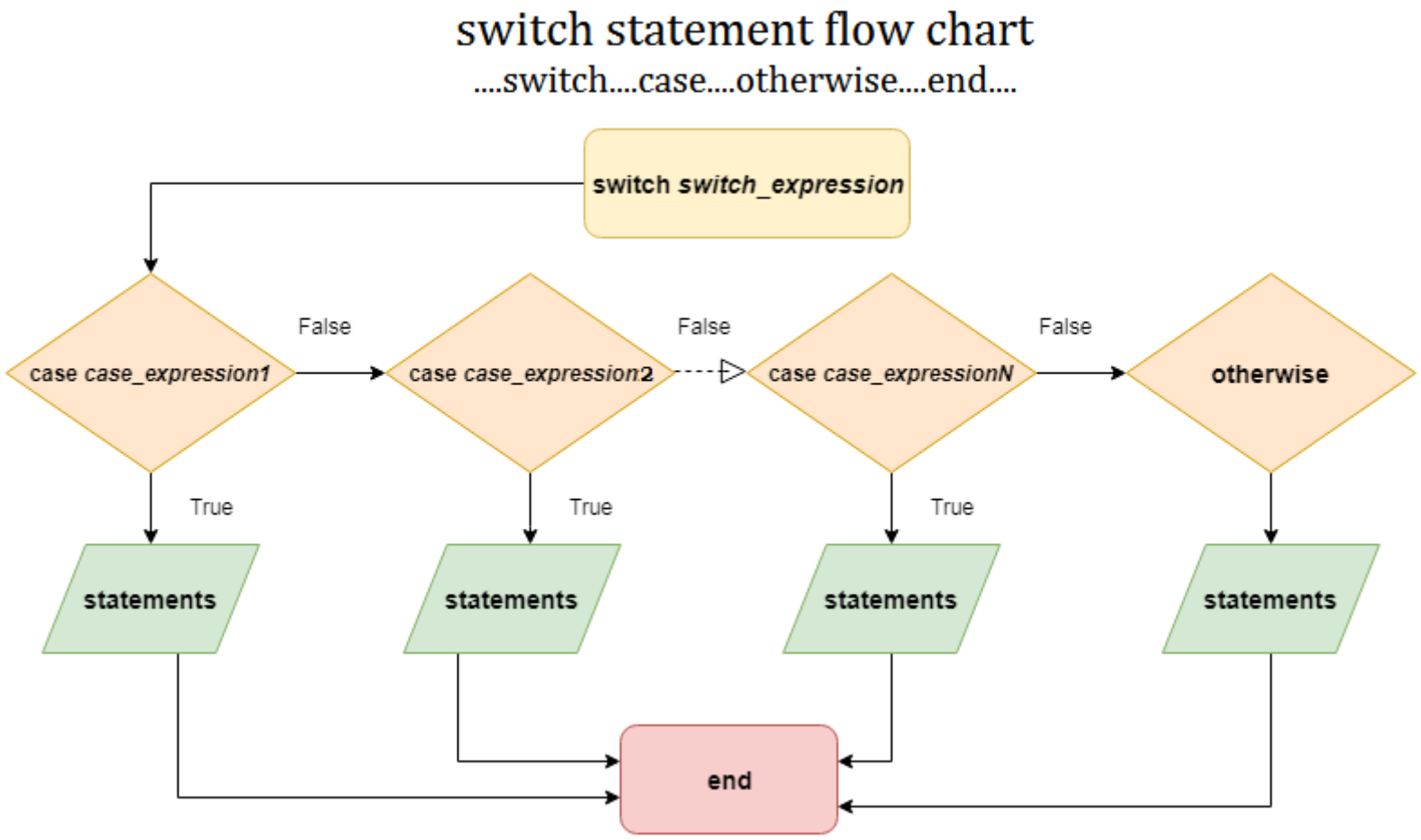
Thursday

Following are the points while using switch in MATLAB:

Similar to **if** block, the **switch** block tests each case until one of the **case_expression** is true. It is evaluated as:

- case & switch must be equal for numbers as- **case_expression == switch_expression**.
- For character vectors, the result returned by the **strcmp** function must be equal to 1 as - **strcmp(case_expression, switch_expression) == 1**.
- For object, **case_expression == switch_expression**.
- For a cell array, at least one of the elements of the cell array in **case_expression** must match **switch_expression**.
- switch statement doesn't test for inequality, so a **case_expression** cannot include relational operators such as **<** or **>** for comparison against the **switch_expression**.

Flow chart of Switch statement:



MATLAB Plotting Function:

- XY-Plotting function:
- ✓XY-plotting functions are used to create linear 2D plot of the linear functions of the function y=f(x).
- ✓such functions are called as XY-functions
- ✓The MATLAB functions which are used to plot such XY-functions are called the XY-plotting functions.
- ✓The basic graphical command in MATLB is 'plot'
- Syntax:

plot(x , y, 'style option');

Where x-independent variable

y-dependent variable

style option- is an optional argument that specifies the color(eg: g,r,b etc), the line style(eg:solid, dashed, dotted etc) and the plot marker style(eg: o,+,* etc)

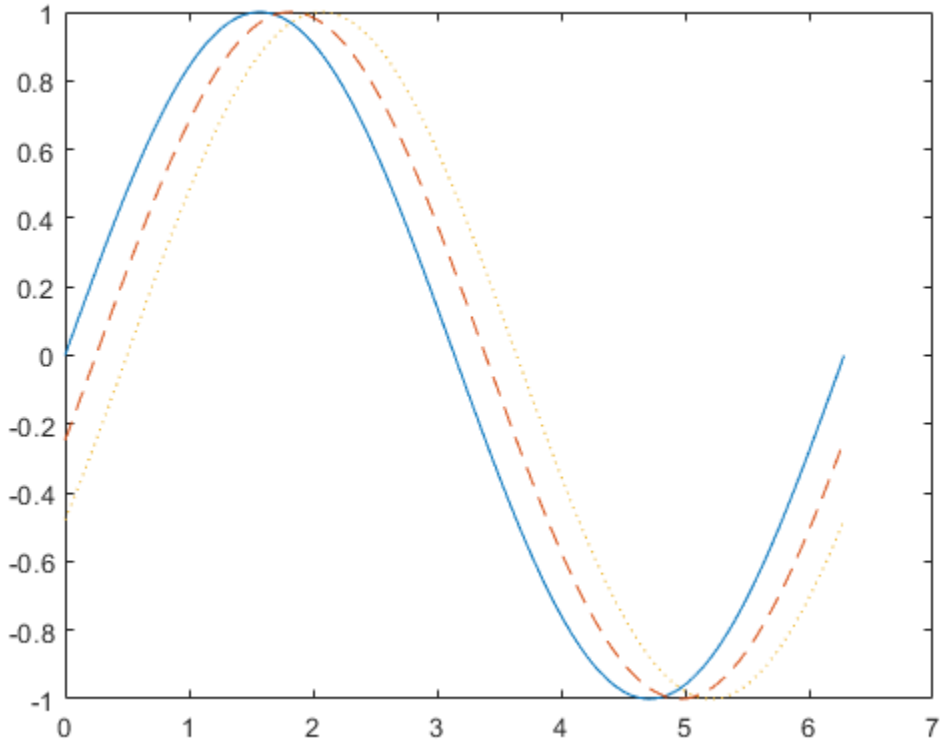
➢Example: plot(x,y)-plots y vs x with solid line

Plot(x,y,’—’)- plots y vs x with a dashed line.

plot(x,y,’r*’)-plots yvs x wth solid line of red color and points are marked by *

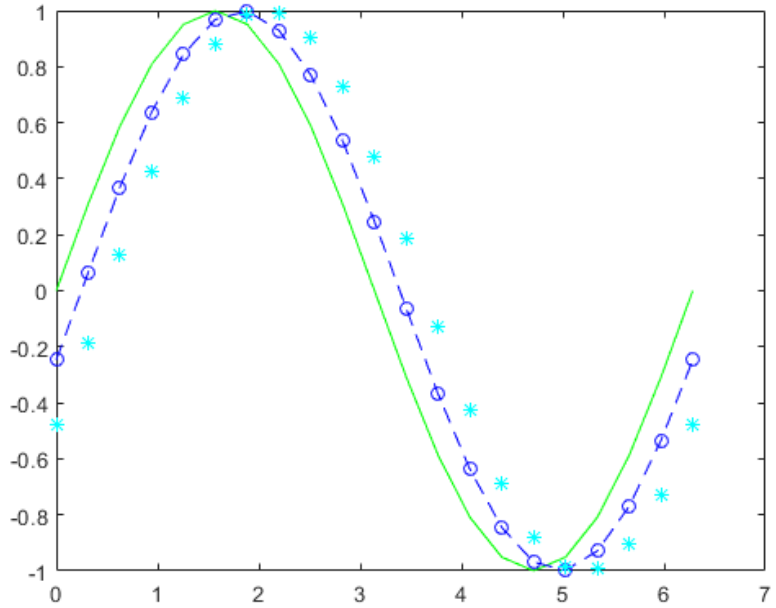
Example:1

```
x= 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);
 figure
plot(x,y1,x,y2,'--',x,y3,':')
```



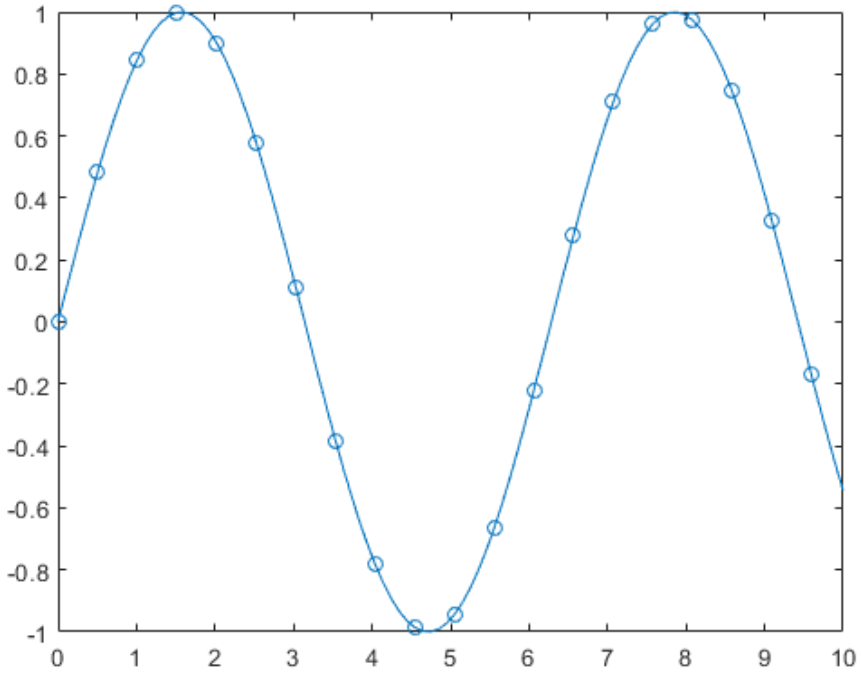
Example:2

```
x = 0:pi/10:2*pi;
y1 = sin(x);
y2 = sin(x-0.25);
y3 = sin(x-0.5);
Figure
 plot(x,y1,'g',x,y2,'b--o',x,y3,'c*')
```



Example:3

```
x = linspace(0,10);
y = sin(x);
plot(x,y,'-o','MarkerIndices',1:5:length(y))
```



Style options:

- The style option in the plot command is a character string that consists of one, two or three characters that specify the color and/ or line style. There are several color, line and marker style options.

- Properties:

the properties of lines used for plotting. MATLAB graphics give you control over these visual characteristics:

- LineWidth — Specifies the width (in points) of the line.
- MarkerEdgeColor — Specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- MarkerFaceColor — Specifies the color of the face of filled markers.
- MarkerSize — Specifies the size of the marker in points (must be greater than 0).

In addition, you can specify the LineStyle, Color, and Marker properties instead of using a line specification character vector.

Marker Specifiers

Specifier	Marker Type
'+'	Plus sign
'o'	Circle
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)

Color Specifiers

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

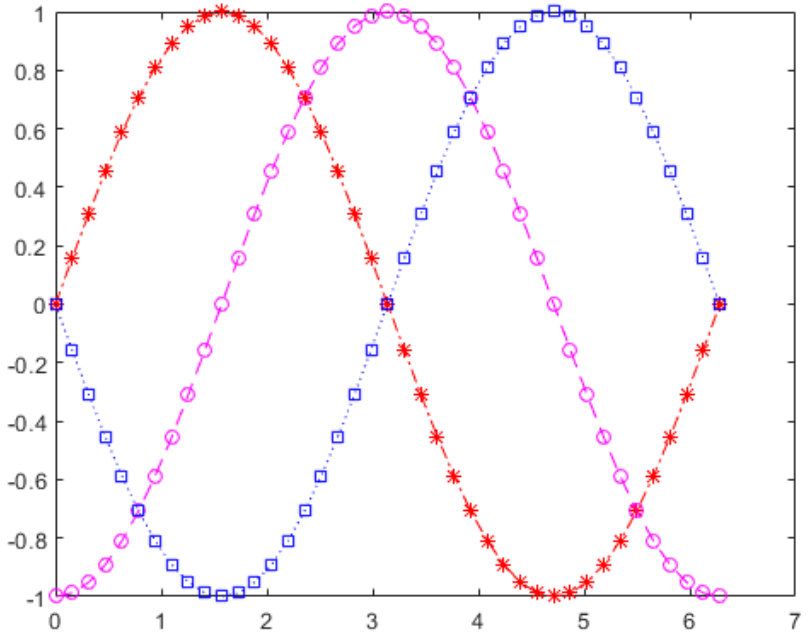
Line Style Specifiers

You indicate the line styles, marker types, and colors you want to display, detailed in the following tables:

Specifier	LineStyle
'-'	Solid line (default)
'--'	Dashed line
':'	Dotted line
'-.'	Dash-dot line

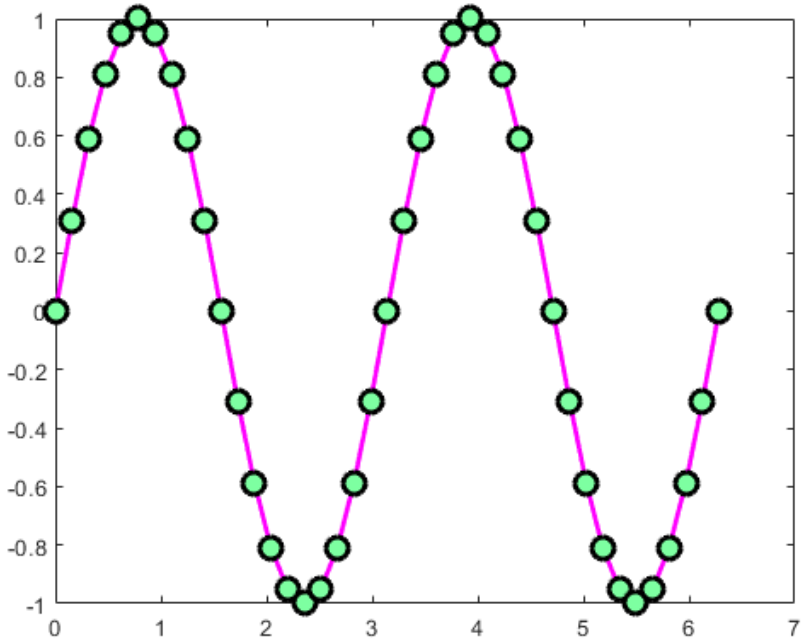
Example:

```
figure t = 0:pi/20:2*pi;
plot(t,sin(t),'-.r*’)
hold on
plot(t,sin(t-pi/2),'--mo’)
plot(t,sin(t-pi),':bs’)
hold off
```



Example:

```
t = 0:pi/20:2*pi;
figure plot(t,sin(2*t),'-mo',...
'LineWidth',2,...
'MarkerEdgeColor','k',...
'MarkerFaceColor',[.49 1 .63],...
'MarkerSize',10)
```



subplot

- Create axes in tiled positions

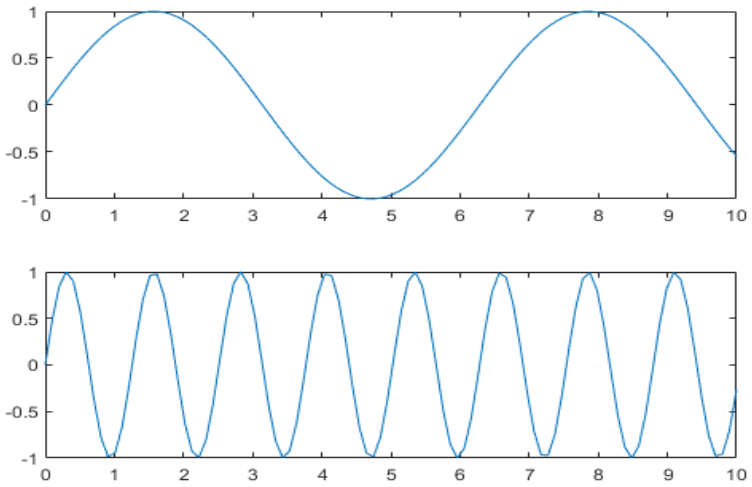
Syntax

```
subplot(m,n,p)
subplot(m,n,p,'replace’)
subplot(m,n,p,'align’)
subplot(m,n,p,ax)
subplot('Position',pos)
subplot(___,Name,Value)
ax = subplot(___)
subplot(ax)
```

Description

- subplot(m,n,p) divides the current figure into an m-by-n grid and creates axes in the position specified by p.
- Example:

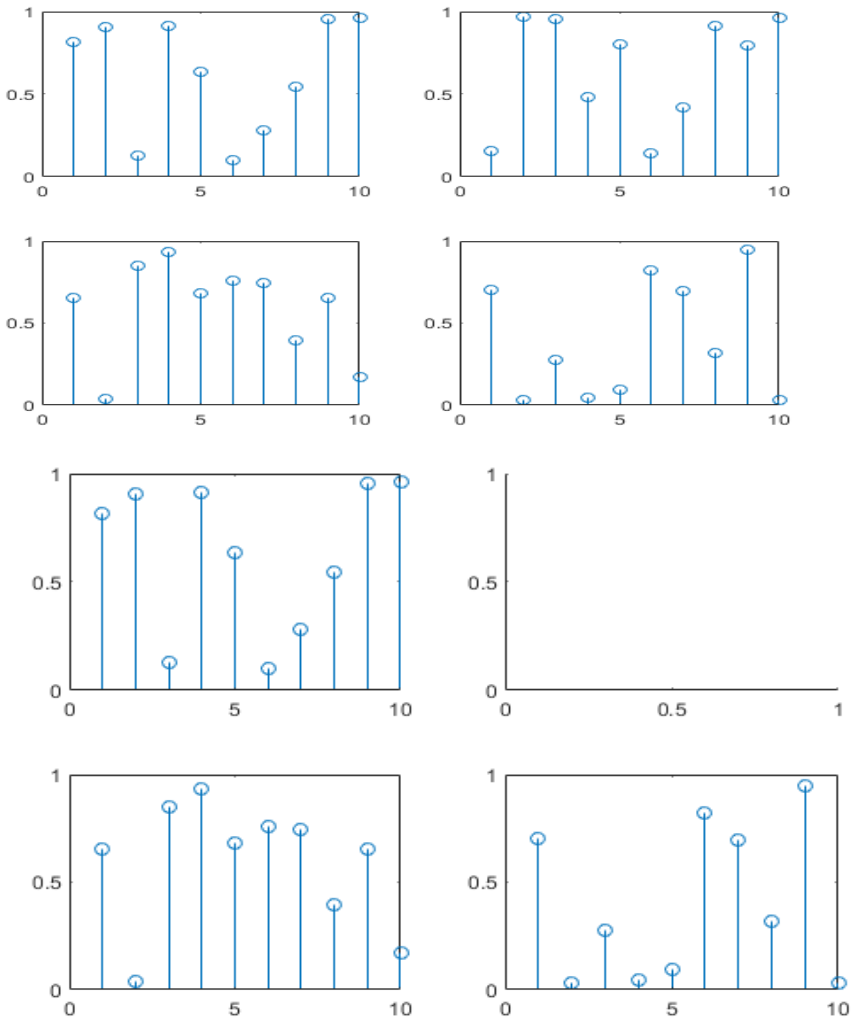
```
subplot(2,1,1);
x = linspace(0,10);
y1 = sin(x);
plot(x,y1)
subplot(2,1,2);
 y2 = sin(5*x);
plot(x,y2)
```



- subplot(m,n,p,'replace') deletes existing axes in position p and creates new axes.
- Example:

```
for k = 1:4
data = rand(1,10);
 subplot(2,2,k)
stem(data)

 end
```

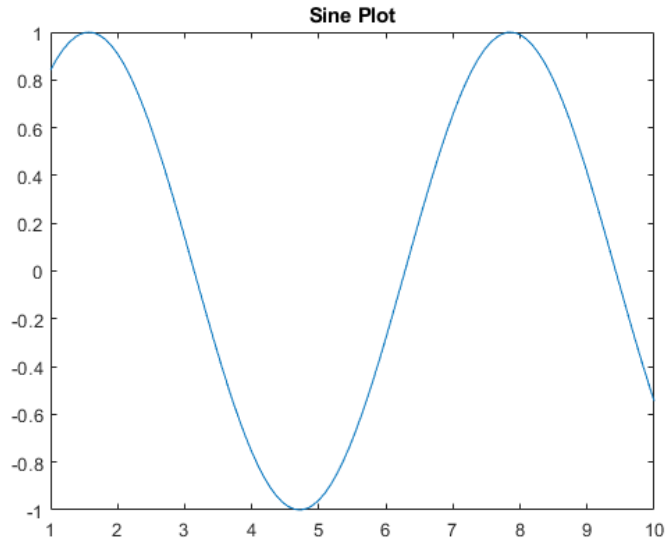
```
subplot(2,2,2,'replace')
```

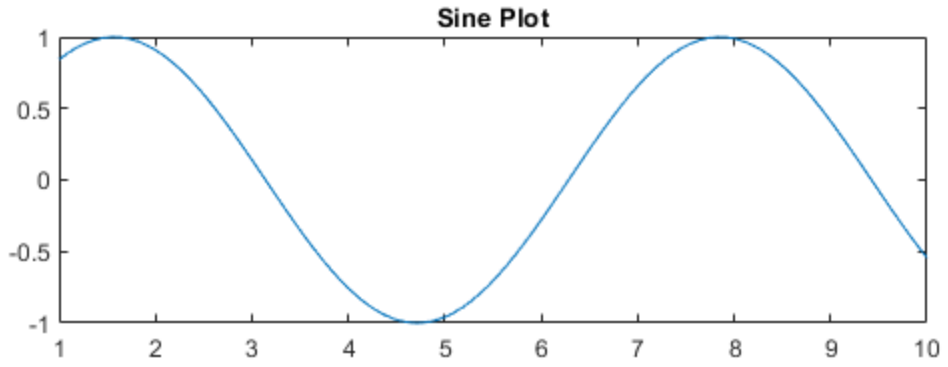


- subplot(m,n,p,'align') creates new axes so that the plot boxes are aligned. This option is the default behavior.
- subplot(m,n,p,ax) converts the existing axes, ax, into a subplot in the same figure.
- Example:

```
x = linspace(1,10);
 y = sin(x);
plot(x,y)

title('Sine Plot')
```

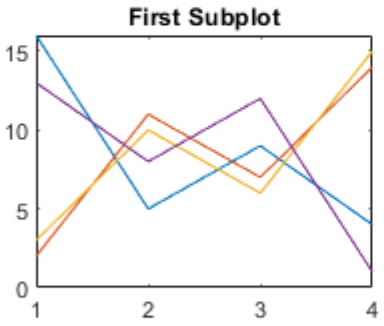
```
ax = gca; % current axes

subplot(2,1,2,ax)
```

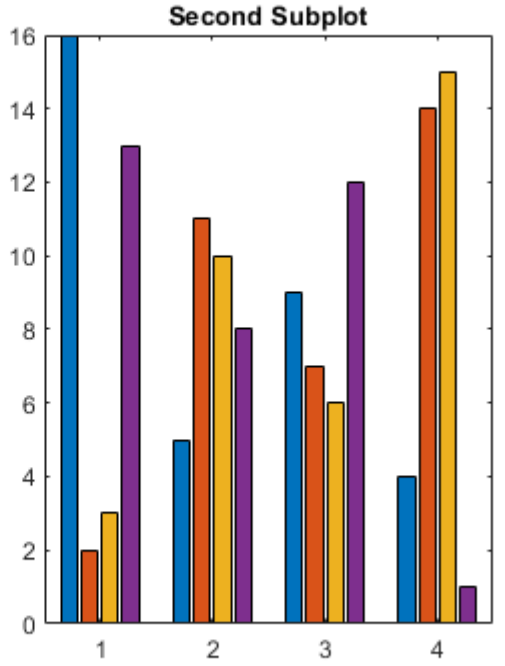




- subplot('Position',pos) creates axes in the custom position specified by pos. Use this option to position a subplot that does not align with grid positions. Specify pos as a four-element vector of the form [left bottom width height]. If the new axes overlap existing axes, then the new axes replace the existing axes.
- Example:

```
pos1 = [0.1 0.3 0.3 0.3];
subplot('Position',pos1)
y = magic(4); plot(y)
title('First Subplot’)
 pos2 = [0.5 0.15 0.4 0.7];
subplot('Position',pos2)
bar(y)

title('Second Subplot’)
```

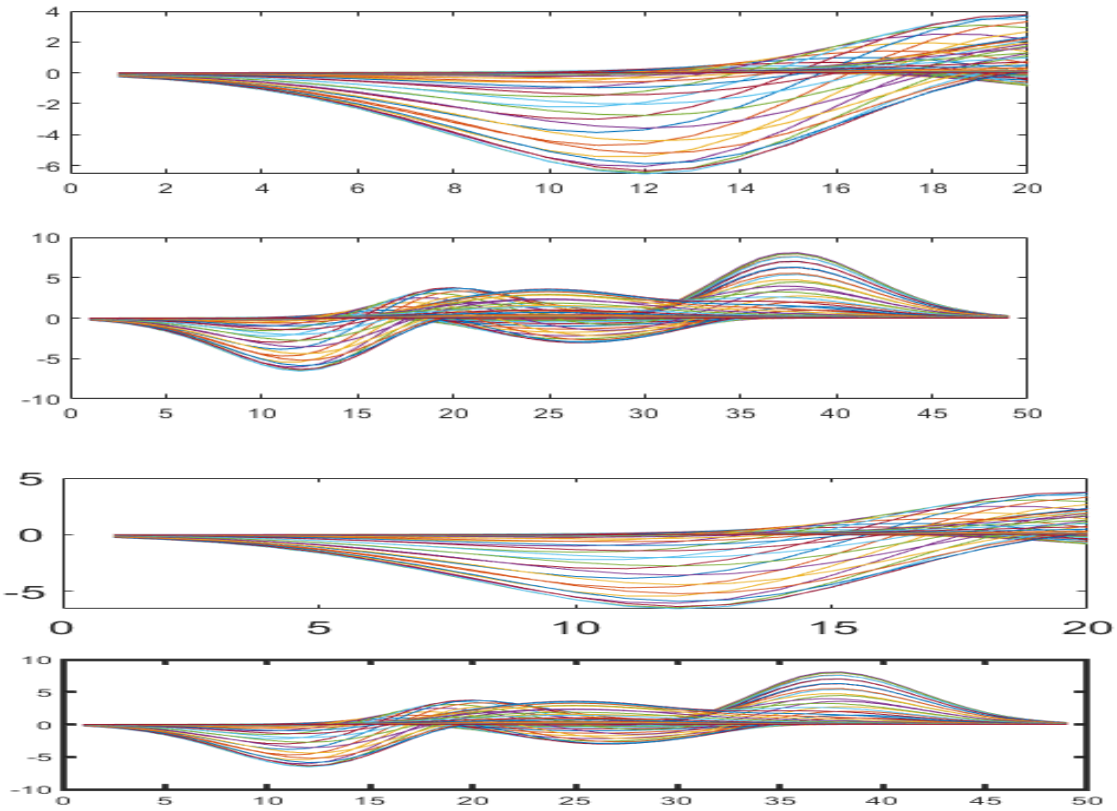




- subplot(___,Name,Value) modifies axes properties using one or more name-value pair arguments. Set axes properties after all other input arguments
- ax = subplot(___) creates an Axes object, PolarAxes object, or GeographicAxes object. Use ax to make future modifications to the axes
- Example:

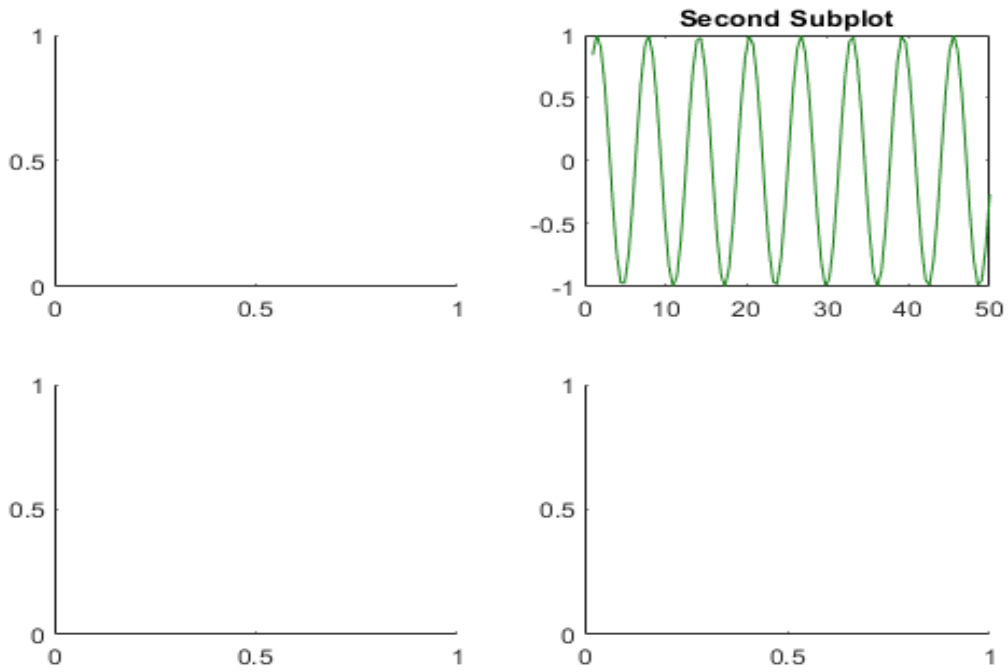
```
ax1 = subplot(2,1,1);
Z = peaks; plot(ax1,Z(1:20,:))
ax2 = subplot(2,1,2);

plot(ax2,Z)
```

```
ax1.FontSize = 15;

ax2.LineWidth = 2;
```



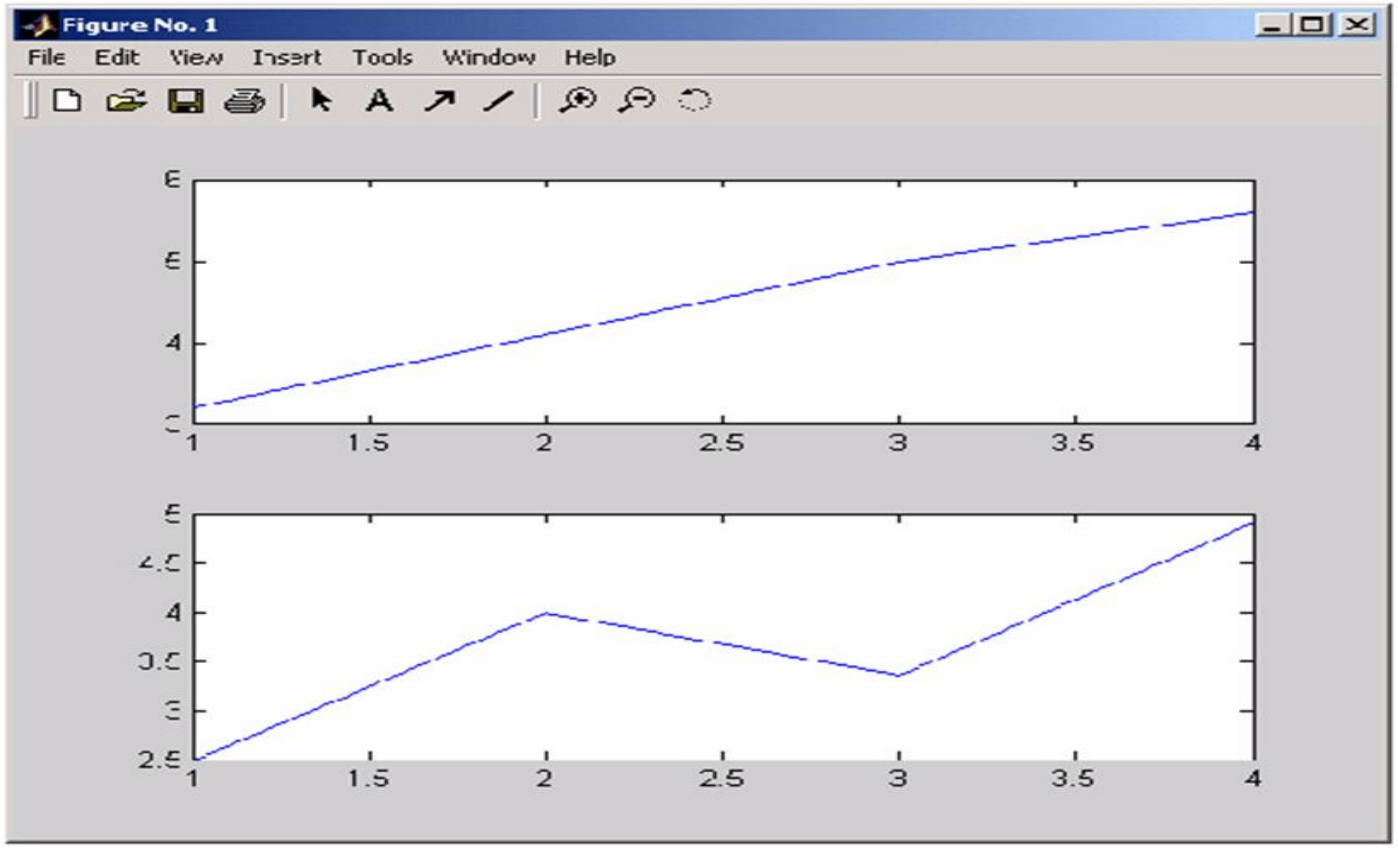
- subplot(ax) makes the axes specified by ax the current axes for the parent figure. This option does not make the parent figure the current figure if it is not already the current figure.

```
for k = 1:4
ax(k) = subplot(2,2,k);
end
subplot(ax(2))
 x = linspace(1,50);
y = sin(x);
plot(x,y,'Color',[0.1, 0.5, 0.1])
title('Second Subplot’)

axis([0 50 -1 1])
```

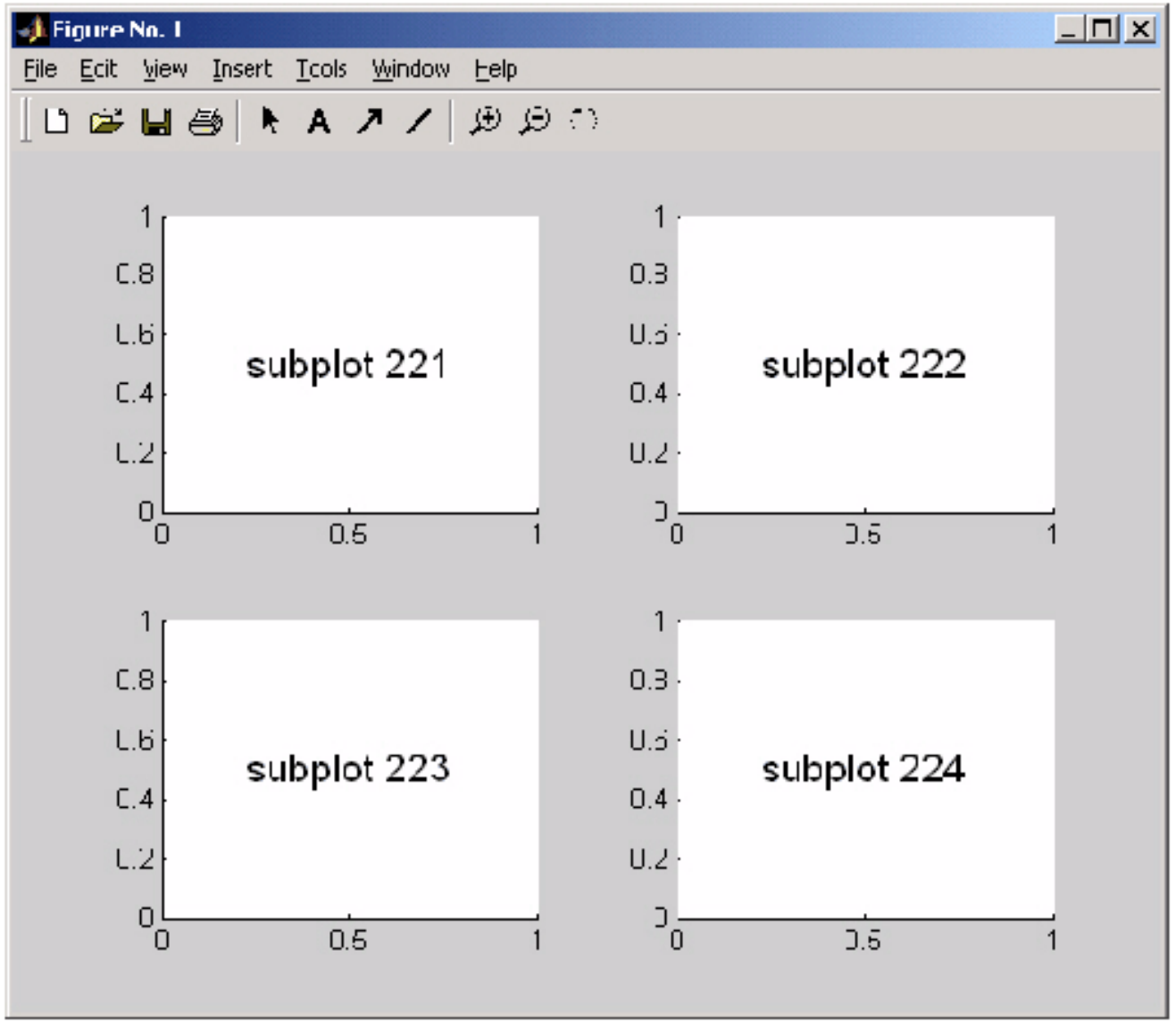


To plot income in the top half of a figure and outgo in the bottom half,

- income = [3.2 4.1 5.0 5.6];
- outgo = [2.5 4.0 3.35 4.9];
- subplot(2,1,1);
- plot(income)
- subplot(2,1,2);
- plot(outgo)



- The following illustration shows four subplot regions and indicates the command used to create each



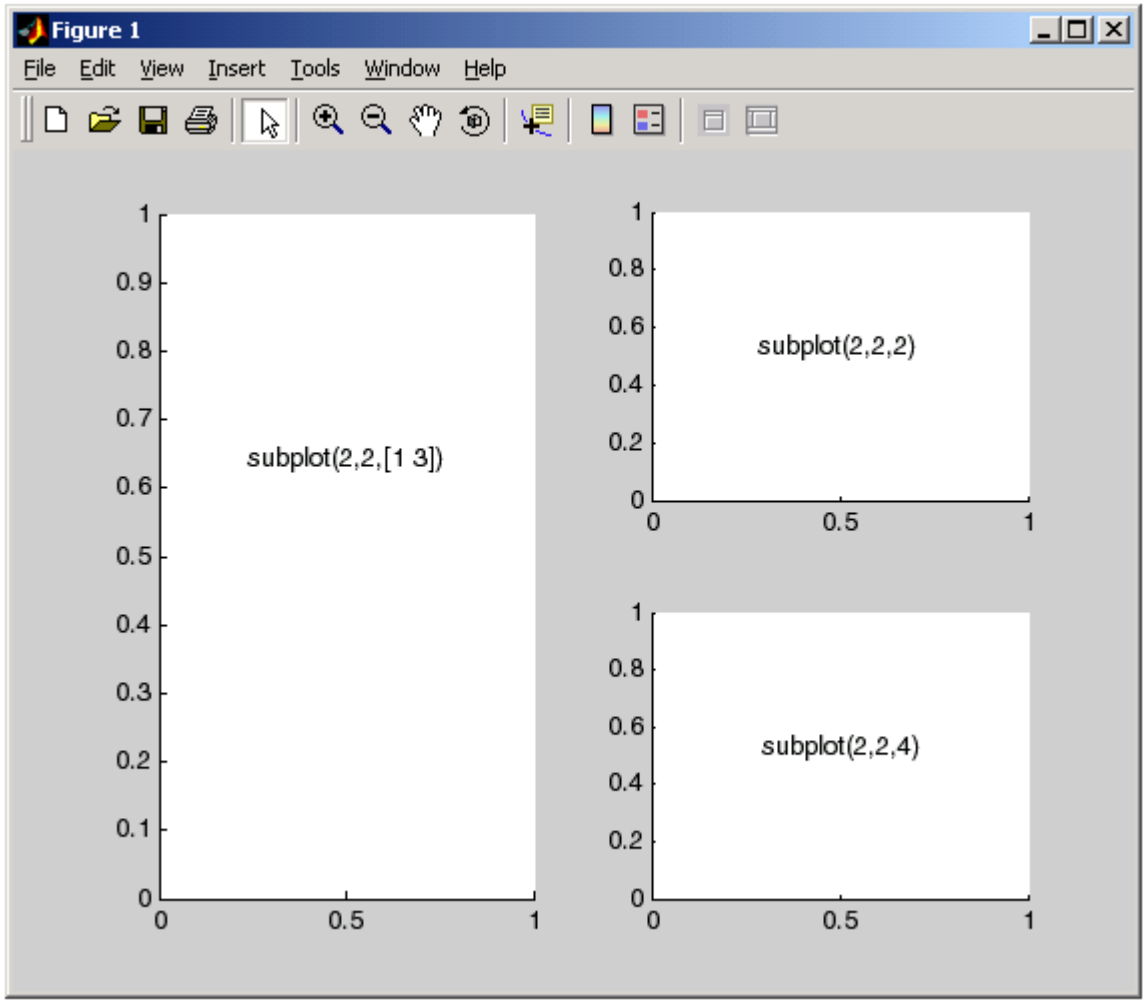


Figure 1
File Edit View Insert Tools Window Help
subplot(2,2,[1 3])
subplot(2,2,2)
subplot(2,2,4)

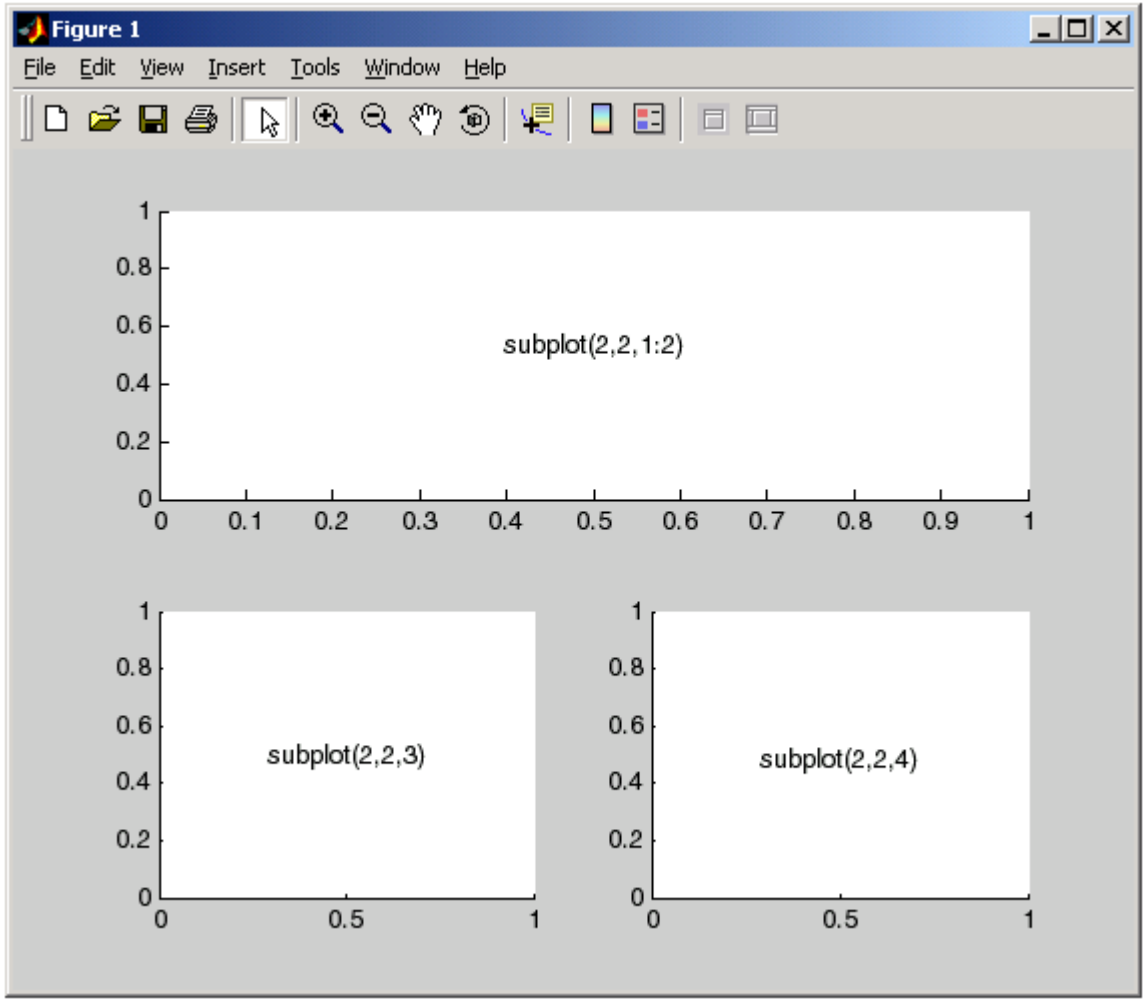


Figure 1
File Edit View Insert Tools Window Help
subplot(2,2,1:2)
subplot(2,2,3)
subplot(2,2,4)

Overlay plots:

➢There are three different ways to generate overlay plots. They are:

1) By using a single plot() command with different data arguments.
2) By using hold command
3) By using line command

By using a single plot() command with different data arguments

- IF the entire set of data is available, plot() command withmultible arguments may be used to generate an overlay plot.
- Example: plot the 3 functions (x_1,y_1), (x_2,y_2) and (x_3,y_3) on a single plot.

the command is:

plot(x_1, y_1, x_2, y_2,'--', x_3, y_3,':')

- It plots (x_1,y_1) with a solid line,
- It plots (x_2,y_2) with a dashed line,
- It plots (x_3,y_3) with a dotted line
- Consider two column matrices/vectors x= [x_1, x_2, x_3] and y= [y_1, y_2, y_3] of length n, plot(x,y) produces a plot with three lines drawn in different colors.

➢Example:

Clc;

x_1 = -5:0.01:5;

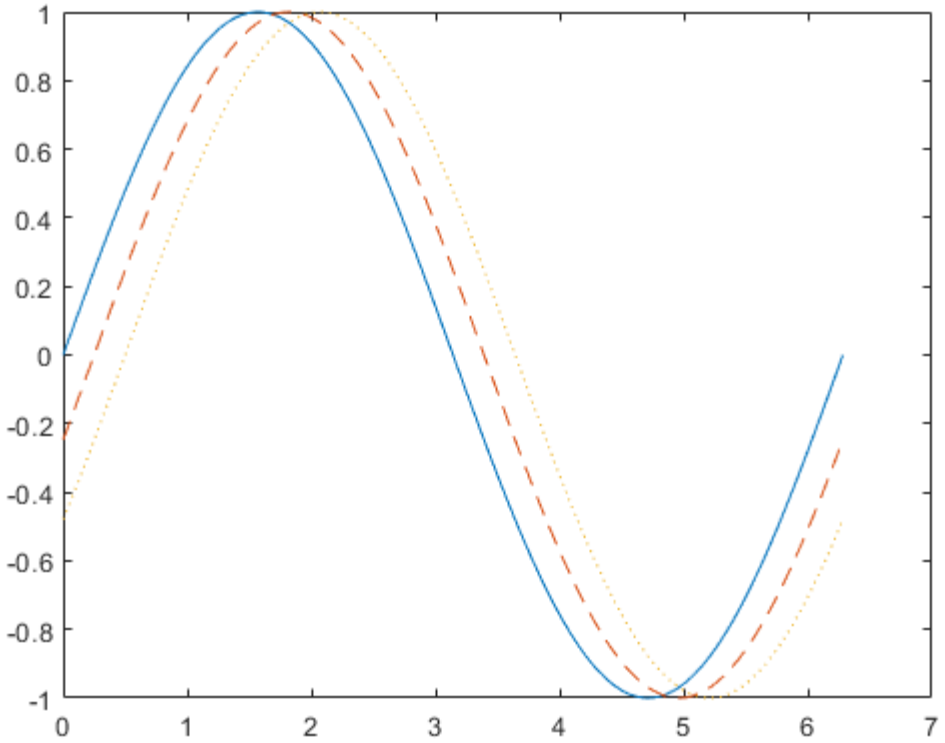
$X_{2=}$ -5:0.1:10;

$X_{3=}$ -5:0.01:10

y1 = sin(2*pi* x_1);

y2 = sin(2*pi* x_2);

y3 = sin(2*pi* x_3);



figure

plot(x_1, y_1, x_2, y_2,'--','linewidth',2, x_3, y_3,':', 'linewidth',2)

xlabel('time:');

ylabel('Amplitude:');

Title(overlay (or)over lapped plot using single plot command');

By using hold command

- Invoking hold on at any point during a session freezes the current plot in the graphics window.
- All subsequent plots generated by the plot command are simply added to the existing plot.
- Example:

```
x1 = -5:0.01:5;
X2= -5:0.1:10;
X3= -5:0.01:10
y1 = sin(2*pi* x1);
y2 = sin(2*pi* x2);
y3 = sin(2*pi* x3);
 figure
plot(x1, y1)
Hold on;
plot(x2, y2,'--','linewidth',2)
Hold on;
plot( x2, y2,'--','linewidth',2)
xlabel('time:');
ylabel('Amplitude:');
Title( overlay (or)over lapped plot using hold command');
```

By using Line command:

- It is a low level graphic command used by plot() command to generate lines.
- Once a plot exists in the graphics window, additional lines may be added by using the line() command directly.
- The line command takes pair of vector directly followed by parameter name/ parameter value pair as arguments.
- syntax:

```
line(xdata,ydata,parameter name,parameter value);
```

➢Example:

```
Clc; clear all; close all;
t=linespace(0, 2*pi,100);
y1 = sin(t);
y2 = t;
y3 = (t.^3)/6;
 figure
plot(t, y)
line(t, y2, 'linestyle', '- -' );
line(t, y3, 'marker', 'o' );
axis([0 5 -1 5]);
xlabel('time:');
ylabel('amplitude of sin(t)');
Title(' sinusoidal function');
```

Loops in mat lab:

- Loops are generally used to iteratively execute a bock of instructions or the specified number of times.
- In MATLAB there are two types of loops are available. They are

1) For loop
2) While loop

For Loop:

- for loop to repeat specified number of times
- A for loop is used to repeat a statement (or) group of statements for a fixed number of times.
- Syntax:

```
for index = values
        statements
end
```

```
Ex:1  for m=1:10
          num=1/(m+1);
      end
```

```
Ex:2  for n=100:-2:0
          k=1/(exp(n));
      end
```

while Loop:

- while loop to repeat when condition is true
- A for loop is used to execute a statement (or) group of statements from indefinite number of times until the condition specified.
- Syntax:

```
while expression
    statements
end
```

Ex:

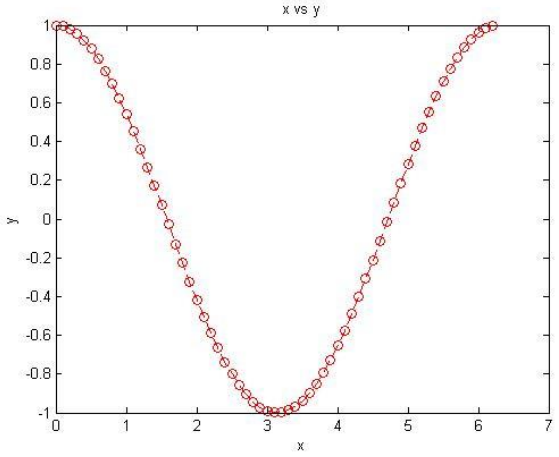
```
n = 10;
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
```

Specialized 2D plots:

- There are many specialized graphics functions for 2D plotting . They are used as alternatives to the plot() command
- A list of function commands used for plotting xy- data are shown bellow:
- **1. Plot command:** Graphical representation for simple functions.

%% Ex 1: Plot cos(x)

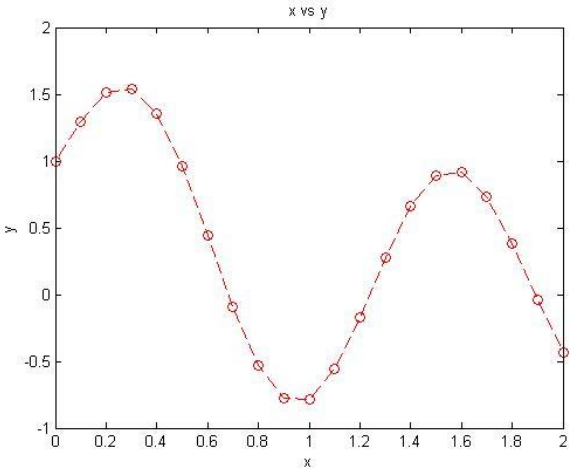
```
x=0:0.1:2*pi;
y=cos(x);
plot(x,y,'r–o')
xlabel('x')
ylabel('y')
title('x vs y')
```



%% Ex 2: Plot

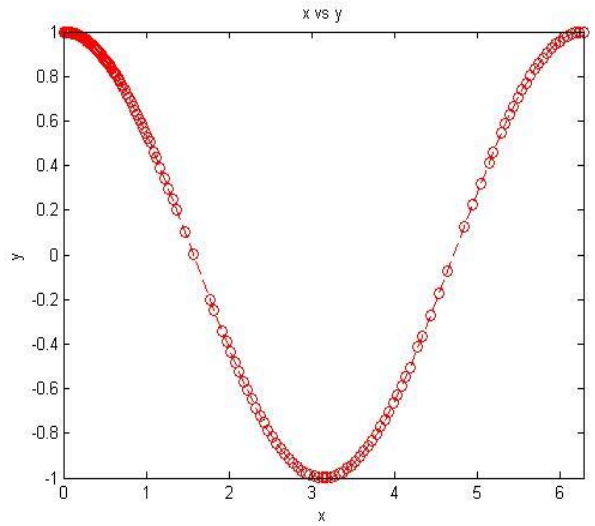
$\sin(5x)+\cos(0.3x)+e^{-2x}$

```
x=0:0.1:2;
y=sin(5*x).*cos(0.3*x)+exp(-2*x);
plot(x,y,'r–o')
xlabel('x')
ylabel('y')
title('x vs y')
```

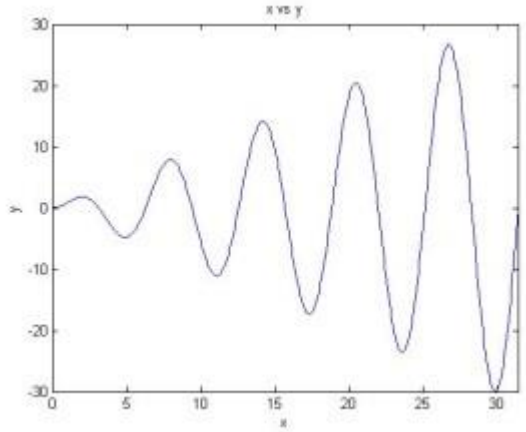


➢2. fplot command: plots a function of a single variable

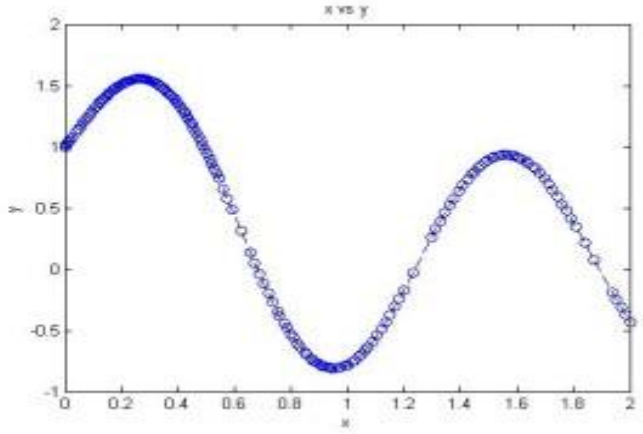
%% fplot ex 1: Plot cos(x)
fplot('cos(x)',[0 2*pi],'r–o')
xlabel('x')
ylabel('y')
title('x vs y')



%% fplot ex 3 : Plot x.sin(x)
fplot('x.*sin(x)',[0 10*pi])
xlabel('x')
ylabel('y')
title('x vs y')



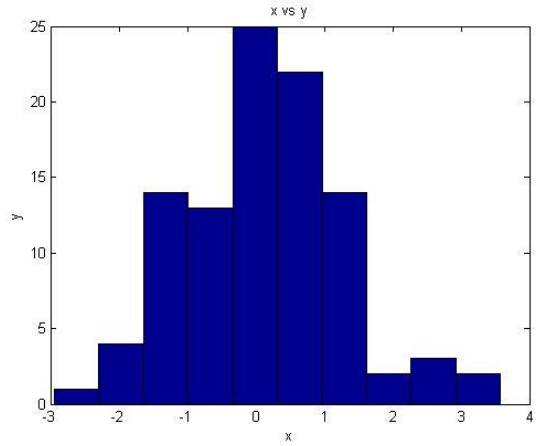
%% fplot ex 2: Plot $\sin(5x)+\cos(0.3x)+e^{-2x}$
fplot('sin(5*x)*cos(0.3*x)+exp(-2*x)',[0 2],'b–o')
xlabel('x')
ylabel('y')
title('x vs y')



➢ 3. HIST Plot: Makes histogram

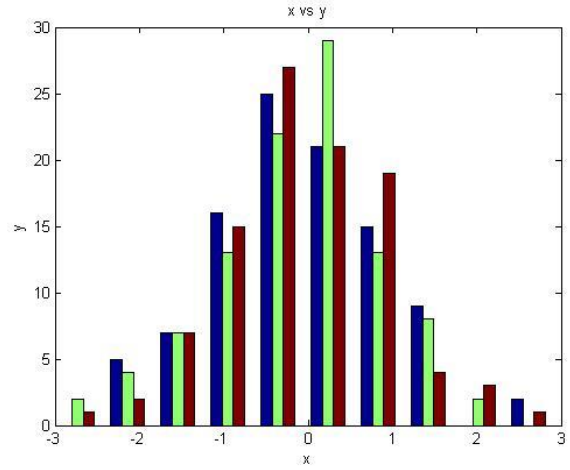
%% hist plot ex 1: Plot random 100 values

```
y = randn(100,1);
hist(y)
xlabel('x')
ylabel('y')
title('x vs y')
```



%% hist plot ex 2:Plot 100×3 values

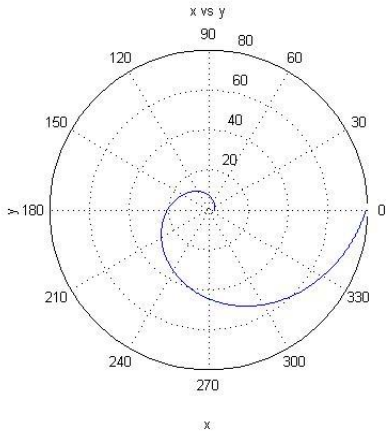
```
y = randn(100,3);
hist(y)
xlabel('x')
ylabel('y')
title('x vs y')
```



➢4. Polar Plot: Plots curves in polar coordinates

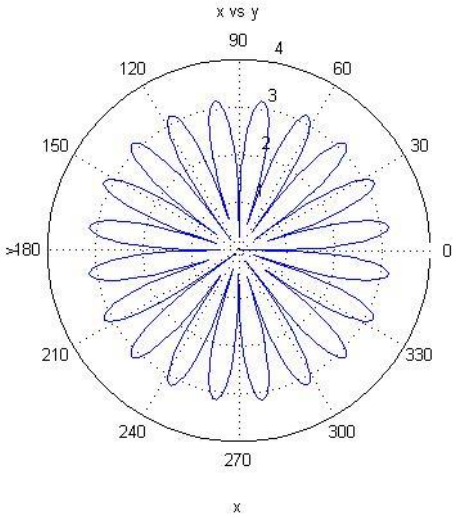
%% polar plot ex 1: Draw the polar plot of $3\cos2(0.3t)+2t^2$

```
t=0:0.01:2*pi;
r=3*cos(0.3*t).^2+2*t.^2;
polar(t,r)
xlabel('x')
ylabel('y')
title('x vs y')
```



%% polar plot ex 2:Draw the polar plot of $r2=10*\sin(10t)$

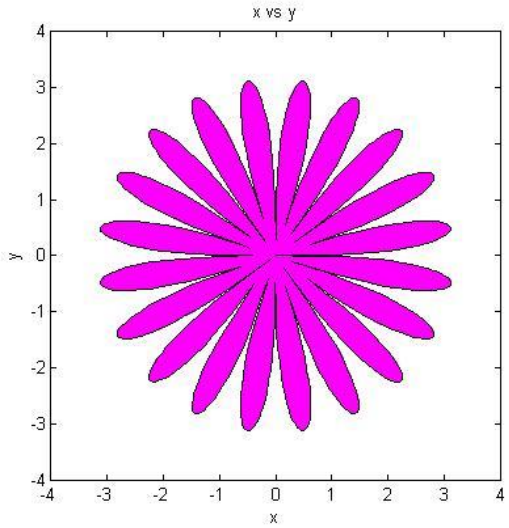
```
t=0:0.01:2*pi;
r = sqrt(abs(10*sin(10*t))) ;
polar (t,r )
xlabel('x')
ylabel('y')
title('x vs y')
```



➢5. FILL Plot: plots filled polygons of specified colors

%% fill plot ex 1: Draw the polar plot of $r^2=10*\sin(10t)$

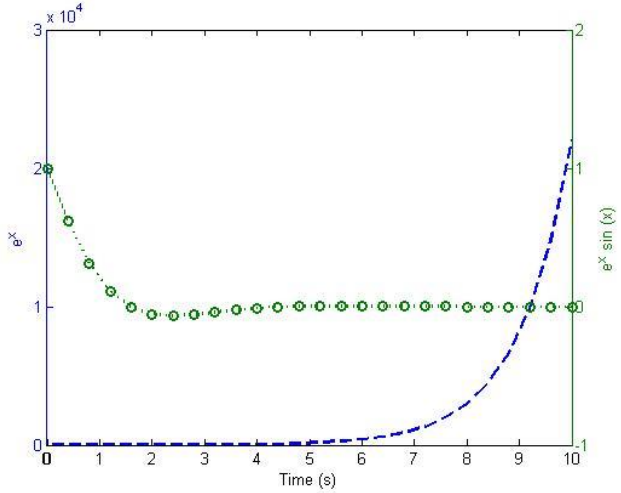
```
t=0:0.01:2*pi;
r = sqrt( abs ( 10* sin ( 10 *t ) ) ) ;
x = r.*cos( t ) ;
y= r.*sin(t ) ;
fill(x , y , ‘ m ‘ ) ,
axis(‘square’ )
xlabel(‘x’)
ylabel(‘y’)
title(‘x vs y’)
```



➢6. Plotyy :Plot two data sets on one graph using two y-axes

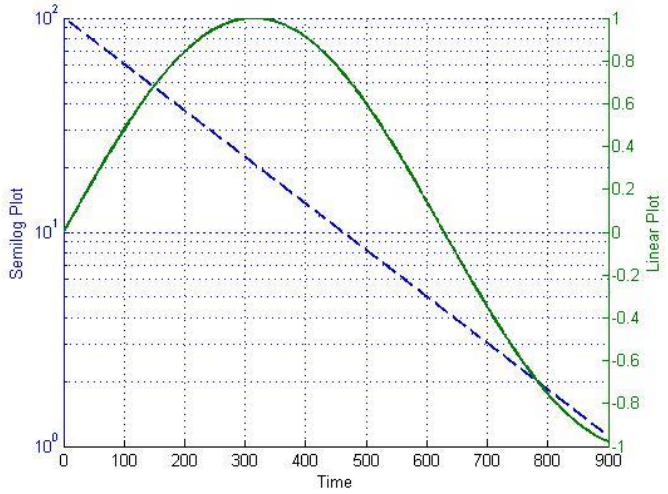
%% plotyy ex 1: Ploty1=ex, y2=e^{-x}∗cos(x)

```
 x = 0:.4 : 10 ;
y1 = exp(x) ;
y2=exp (- x ).*cos( x ) ;
[HA,HL1,HL2] = plotyy(x ,y1 ,x ,y2 ) ;
ylabel(HA(1),'e^x')
ylabel(HA(2),'e^x sin (x)')
xlabel(HA(1),'Time (s)')
% xlabel(HA(2),'Hello')
set(HL1,'LineStyle','–','LineWidth',2);
set(HL2,'LineStyle',':','marker','o','LineWidth',2);
```



%% plotyy ex 2 :Plotz1=100∗$e^{-0.005*t}$, z2=sin(0.005∗t)

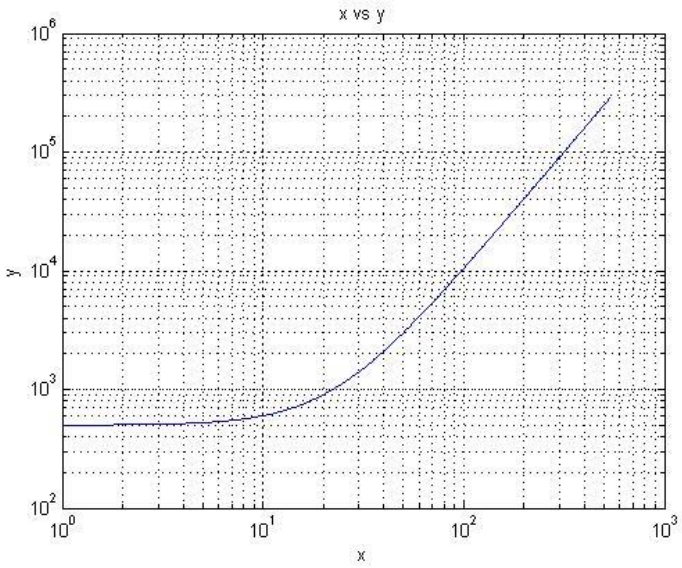
```
t = 0:900;
z1 =100*exp(-0.005*t); z2 = sin(0.005*t);
figure
[HA,HL1,HL2] = plotyy(t,z1,t,z2,'semilogy','plot');
ylabel(HA(1),'Semilog Plot')     % label left y-axis
ylabel(HA(2),'Linear Plot')      % label right y-axis
xlabel(HA(2),'Time') % label x-axis
set(HL1,'LineStyle','–','LineWidth',2);
set(HL2,'LineWidth',2);
grid(HA(1),'on');
```



7. Logarithmic Plots in MATLAB

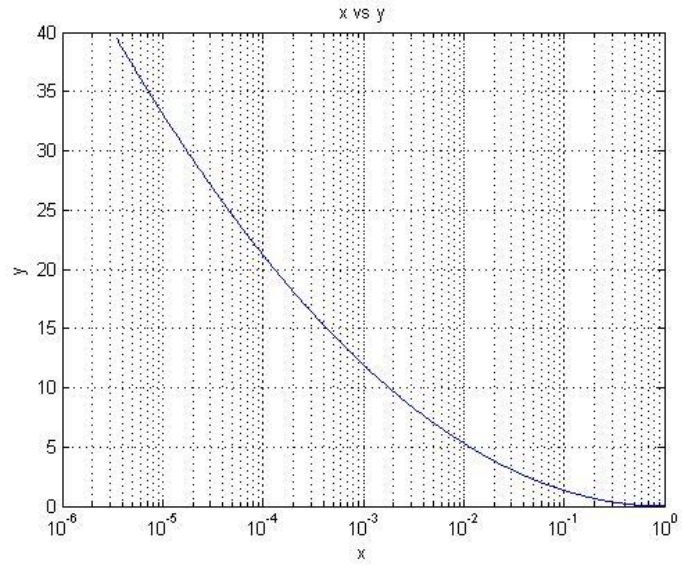
%% loglog plot 1 : Draw the logarithmic plot of x=exp(t), y=500

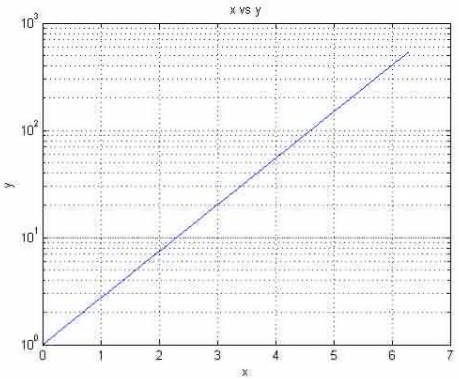
```
t=0:0.01:2*pi ;
x = exp (t ) ;
y=500 + exp (2*t ) ;
loglog(x , y),
grid
xlabel('x')
ylabel('y')
title('x vs y')
```



%% semilogx plot 2: Draw the logarithmic plot of $x=e^{-2t}$, $y= t^2$

```
t=0:0.01:2*pi ;
x = exp(-2*t);
y = t.^2 ;
semilogx (x , y) ,   %% (or) semilogy (t,exp(t) )
grid
xlabel('x')
ylabel('y')
title('x vs y')
```

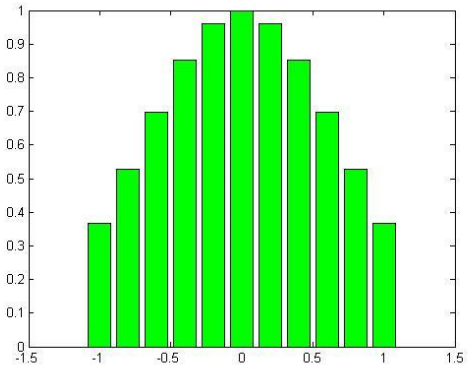




➢8. Bar Plot:

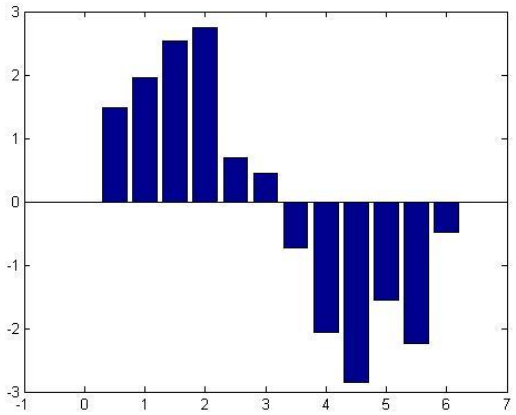
Bar plot ex 1: Draw the bar plot of $x = e^{-t^2}$,

```
t = -1:0.2:1;
bar(t,exp(-t.^2),'g')
```



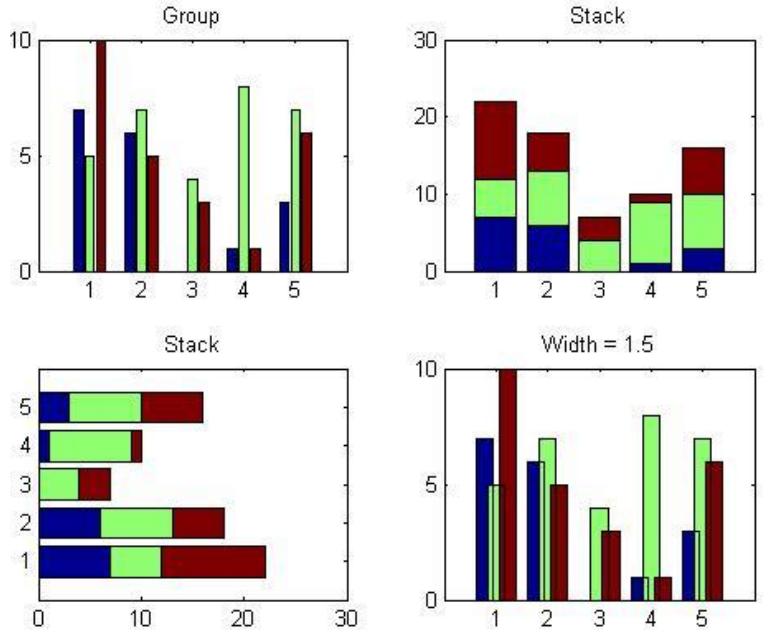
Bar plot ex 2: Draw the bar plot of r2=10∗sin(t) **and** y=r∗si

```
t=0:0.5:2*pi ;
r = sqrt(abs(10*sin(10*t))) ;
y =r.*sin(t) ;
% group
bar (t,y)
```



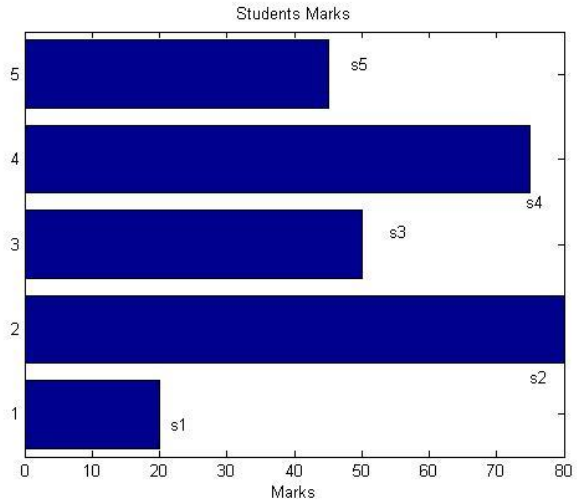
Bar plot ex 3: Draw the bar plot

```
y = round(rand(5,3)*10);
subplot(2 , 2, 1);
bar(y, 'group');
title ('Group');
subplot(2, 2, 2);
bar(y, 'stack');
title ('Stack');
subplot(2, 2, 3);
barh(y, 'stack');
title ( 'Stack');
subplot(2, 2, 4);
bar(y,1.5);
title ('Width = 1.5');
```



Bar plot ex 4: Draw the bar plot for students marks in a class

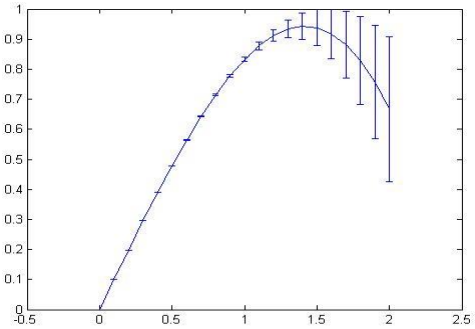
```
cont = char ( 's1 ' , 's2' , ' s3 ',' s4' , ' s5 ' ) ;
mar = [20 ; 80 ; 50 ; 75 ; 45] ;
barh (mar)
for i =1:5,
gtext (cont(i,:));
end
xlabel('Marks' )
title( 'Students Marks')
```



➢9. Error bar:

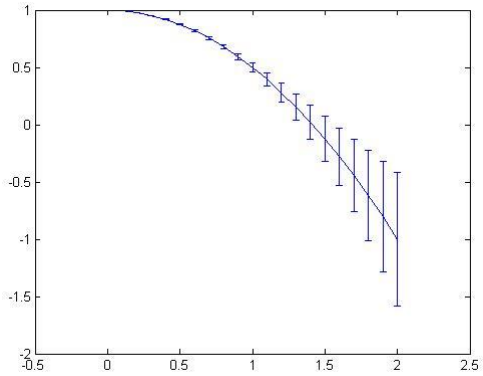
Errorbar plot ex 1: Draw the error plot of sin(x) with its approximation

```
 x=0:0.1 : 2 ;
aprx2= x – x.^3/6 ;
er = aprx2 – sin (x ) ;
errorbar (x , aprx2 , er )
```



Errorbar plot ex 2: Draw the error plot of cos(x) with its approximation

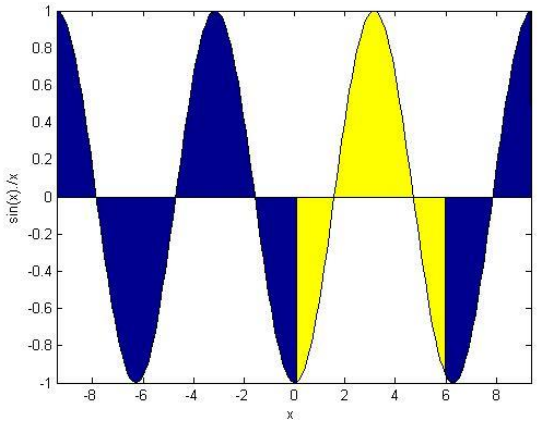
```
x=0:0.1 : 2 ;
aprx2= 1 – x.^2/2 ;
er = aprx2 – cos(x ) ;
errorbar (x , aprx2 , er )
```



➢10. Area:

Area plot ex 1

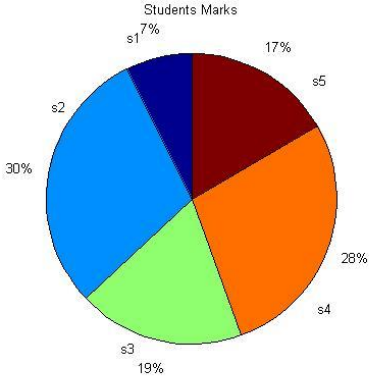
```
x=-3*pi:0.1:3*pi;
%y =-sin (x)./x ;
y =-cos(x);
area( x , y)
xlabel( ‘x ‘ ), ylabel (‘ sin(x)./x ‘ )
hold on
x1= x (96: 155 ) ; y1 = y( 96: 155) ;
area( x1,y1,’facecolor’ ,’y ‘ )
```



➢11. pie:

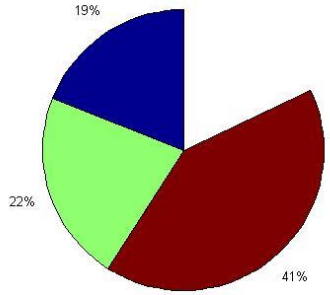
pie plot ex 1

```
cont = char ( ‘s1 ‘ , ‘s2’ , ‘ s3 ‘,‘ s4’ , ‘ s5 ‘ ) ;
mar = [20 ; 80 ; 50 ; 75 ; 45] ;
pie(mar)
for i =1:5,
gtext (cont(i,:));
end
xlabel(‘Marks’ )
title( ‘Students Marks’)
```



pie plot ex 2

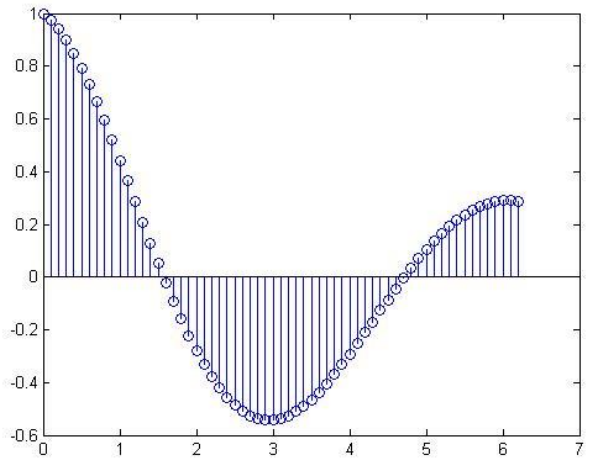
```
x = [.19 .22 .41];
pie(x)
```



➢12. Stem:

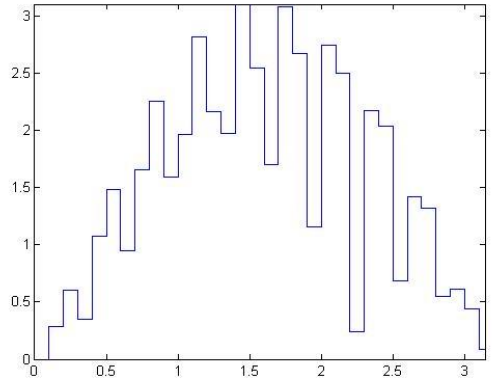
Stem plot ex 1

```
t=0:0.1:2*pi ;
y = exp(-.2*t).*cos(t) ;
stem (t , y)
```



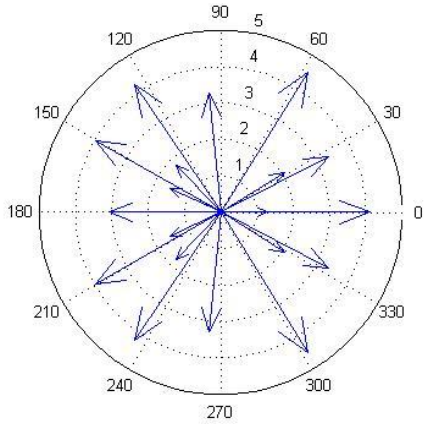
➢13. Stairs:

Stairs Plot ex 1t=0:0.1:2*pi ;
% t=linspace(0,2*pi , 200) ;
r=sqrt(abs(10*sin(10 *t))) ;
y=r.*sin(t) ;
stairs (t,y)
axis([0 pi 0 inf]) ;



➢ 14. Compass:

%The compass function shows vectors emanating from the origin of a graph.
%The function takes Cartesian coordinates and plots them on a circular grid.
Compass plot ex 1
x = randn(20,20);
y = eig(x);
compass(y)

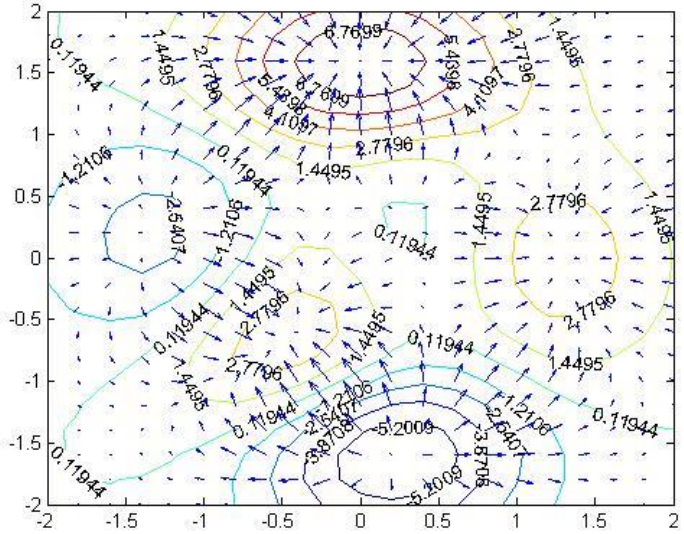


➢15. Comet:

Comet plot ex 1t=0:0.001:4*pi ;
y = t.*sin (t) ;
% y = sin (t) ; % Use this for 2nd exmple
comet(t,y)

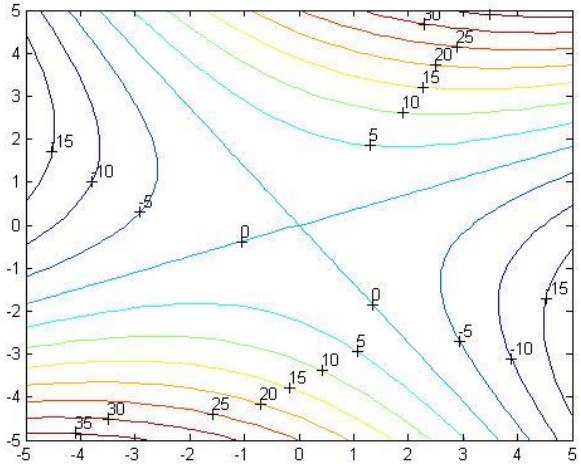
➢ 16. Contour:

contour plot ex 1n = -2.0:.2:2.0;
[X,Y,Z] =peaks(n);
[C, h] = contour(X,Y,Z,10); % Displays 2-D isolines generated from values given by a matrix Z.
colormap('default');
clabel(C,h) % Generates labels using the contour matrix and displays the labels in the current figure.
[U,V] = gradient(Z,.2);
hold on
quiver(X,Y,U,V)
hold off



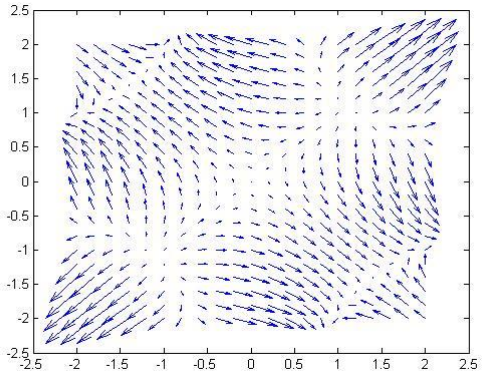
➢17. Meshgrid:

meshgrid ex 1r = -5:.2:5 ;
[X,Y] = meshgrid (r,r) ;
Z =-0.5*X.^2+ X.*Y +Y.^2 ;
cs =contour (X , Y , Z) ;
clabel(cs)



➢ 18. Quiver:

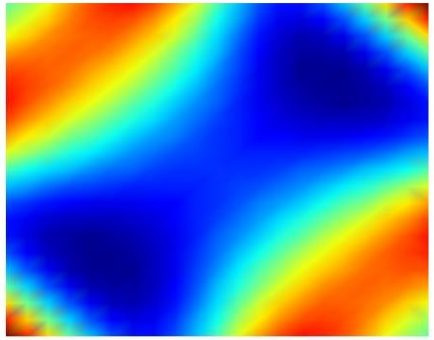
quiver ex 1[X , Y] = meshgrid(r,r) ;
Z = X.^2-5*sin(X.*Y)+ Y.^ 2 ;
[dx,dy]=gradient(Z,.2 ,.2) ;
quiver(X,Y,dx,dy,2) ;



➢ 19. Pcolor:

%pcolor(C) draws a pseudo color plot

pcolor ex 1r = -2:.2:2 ;
[X,Y] =meshgrid(r,r) ;
Z=X.^2-5*sin(X.*Y)+ Y.^2 ;
pcolor(Z), axis('off')
shading interp



➢ 20. Feather:

% feather plot ex 1
th=-pi:pi/10:pi ;
x=cos (th) ;
y=sin (th) ;
feather(x+i*y) % z=complex(x,y) and then feather(z)

➢ 21. Rose:

% rose plot ex 1t=0:pi/100:2*pi ;
rose(t)

2D-3D functions:

Line Plots

plot	2-D line plot
plot3	3-D point or line plot
stairs	Stairstep graph
errorbar	Line plot with error bars
area	Filled area 2-D plot
stackedplot	Stacked plot of several variables with common x-axis
loglog	Log-log scale plot
semilogx	Semilog plot (x-axis has log scale)
semilogy	Semilog plot (y-axis has log scale)
fplot	Plot expression or function
fimplicit	Plot implicit function
fplot3	3-D parametric curve plotter
LineSpec (Line Specification)	Line specification
ColorSpec (Color Specification)	Color specification

Data Distribution Plots

histogram	Histogram plot
histogram2	Bivariate histogram plot
morebins	Increase number of histogram bins
fewerbins	Decrease number of histogram bins
histcounts	Histogram bin counts
histcounts2	Bivariate histogram bin counts
boxchart	Create box chart (box plot)
scatter	Scatter plot
scatter3	3-D scatter plot
binscatter	Binned scatter plot
scatterhistogram	Create scatter plot with histograms
spy	Visualize sparsity pattern of matrix
plotmatrix	Scatter plot matrix
parallelplot	Create parallel coordinates plot
pie	Pie chart
pie3	3-D pie chart
heatmap	Create heatmap chart
sortx	Sort elements in heatmap row
sorty	Sort elements in heatmap column
wordcloud	Create word cloud chart from text data

Discrete Data Plots

bar	Bar graph
barh	Horizontal bar graph
bar3	Plot 3-D bar graph
bar3h	Plot horizontal 3-D bar graph
pareto	Pareto chart
stem	Plot discrete sequence data
stem3	Plot 3-D discrete sequence data
scatter	Scatter plot
scatter3	3-D scatter plot
stairs	Stairstep graph

Geographic Plots

geoplot	Plot line in geographic coordinates
geoscatter	Scatter chart in geographic coordinates
geobubble	Visualize data values at specific geographic locations
geodensityplot	Geographic density plot
geobasemap	Set or query basemap
geolimits	Set or query geographic limits
geoaxes	Create geographic axes
geotickformat	Set or query geographic tick label format

Polar Plots

`polarplot`	Plot line in polar coordinates
`polarscatter`	Scatter chart in polar coordinates
`polarhistogram`	Histogram chart in polar coordinates
`compass`	Plot arrows emanating from origin
`ezpolar`	Easy-to-use polar coordinate plotter
`rlim`	Set or query r-axis limits for polar axes
`thetalim`	Set or query theta-axis limits for polar axes
`rticks`	Set or query r-axis tick values
`thetaticks`	Set or query theta-axis tick values
`rticklabels`	Set or query r-axis tick labels
`thetaticklabels`	Set or query theta-axis tick labels
`rtickformat`	Specify r-axis tick label format
`thetatickformat`	Specify theta-axis tick label format
`rtickangle`	Rotate r-axis tick labels
`polaraxes`	Create polar axes

Regression

- Regression models describe the relationship between a response (output) variable, and one or more predictor (input) variables.

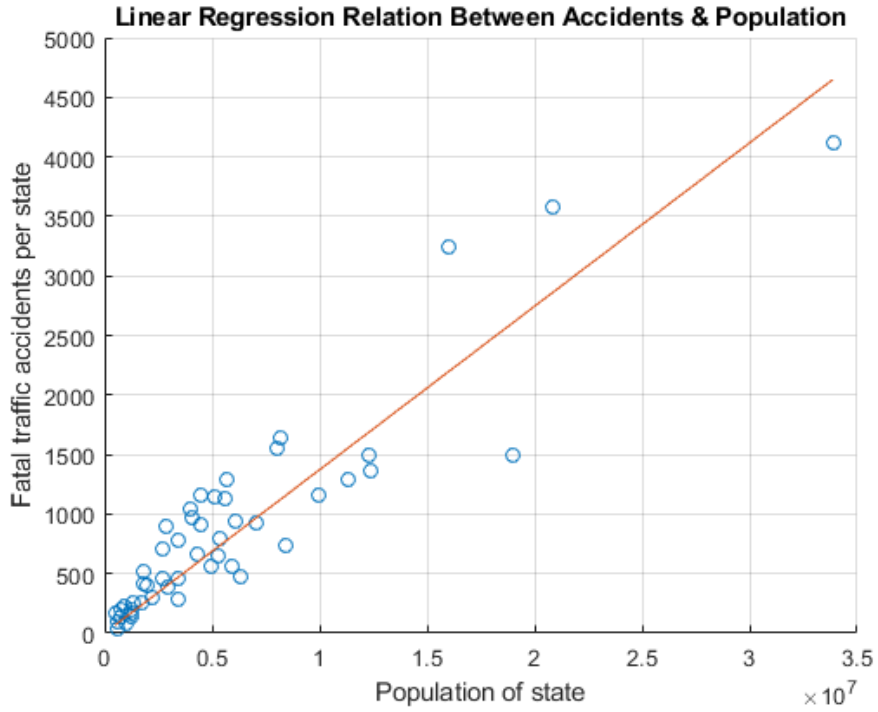
- The most common type of linear regression is a *least-squares fit*, which can fit both lines and polynomials, among other linear models.

- This topic explains how to:
  - Perform simple linear regression using the \ operator.
  - Use correlation analysis to determine whether two quantities are related to justify fitting the data.
  - Fit a linear model to the data.
  - Evaluate the goodness of fit by plotting residuals and looking for patterns.
  - Calculate measures of goodness of fit R^2 and adjusted R^2

- Find the linear regression relation between the accidents in a state and the population of a state using the \ operator. The \ operator performs a least-squares regression
- Calculate the accidents per state yCalc from x using the relation. Visualize the regression by plotting the actual values y and the calculated values yCalc.

Program:

```
b1 = x\y
yCalc1 = b1*x;
scatter(x,y)
hold on
plot(x,yCalc1)
xlabel('Population of state')
ylabel('Fatal traffic accidents per state')
title('Linear Regression Relation Between
        Accidents & Population')
grid on
```



UNIT-V

Linear systems of Algebraic equations

Introduction

- In MATLAB, solving a set of linear algebraic equations is very easy.
- To solve a system of equations or scientific calculations on a calculator on a computer is a big task.
- There are several conventional methods to solve linear algebraic equations such as
  - ✓Elementary solution methods(or) Left division method
  - ✓Gaussian elimination method
  - ✓Eigen values and eigen vector method
  - ✓solve method
  - ✓Cramer’s method

✓ Elementary solution methods(or) Left division method

- In MATLAB, to solve the linear equations by using left division method is left divider symbol('\').
- Example:

considering linear algebraic equations

$$2x+y+z=2$$

$$y-x=z+3$$

$$x+2y+3z+10=0$$

- Steps to solve the linear equations as follows:
  - Step 1: Re arrange the given system of linear algebraic equations as

$$2x+y+z=2$$

$$-x+y-z=3$$

$$x+2y+3z=10$$

Unknown values on Left hand side

known values on Right hand side

- Step 2: Represent re –arranged system of equations in matrix form as

$$[A]\,[x] = [B]$$

where [x]-is a vector of unknown

$$\begin{bmatrix} 2 & 1 & 1 \\ -1 & 1 & -1 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ -10 \end{bmatrix}$$

- Step 3: Solve vector x with

x=A\B

MATLAB program:

```
Clc;
Clear all;
Close all;
A=[2 1 1;-1 1 -1;1 2 3];
B=[2; 3; -10]
X=A\B;
```

Simulation:

```
X←enter
X=      3
        1
        -5
```

From X, x=3; y=1; z=-5

✓ Gaussian Elimination Method:

- Gaussian elimination method, to solve the linear equations by using the MATLAB's inbuilt command 'rref'.

- The command R = rref(A) produces the reduced row echelon form of A using Gauss Jordan elimination with partial pivoting.

- Example:

considering linear algebraic equations

$$2x+y+z=2$$
$$y-x=z+3$$
$$x+2y+3z+10=0$$

- Steps to solve the linear equations as follows:
  - Step 1: Re arrange the given system of linear algebraic equations as

$$2x+y+z=2$$
$$-x+y-z=3$$
$$x+2y+3z=10$$

Unknown values on Left hand side

known values on Right hand side

- Step 2: Represent re –arranged system of equations in matrix form as

  where [x]-is a vector of unknown

$$\begin{bmatrix} 2 & 1 & 1 \\ -1 & 1 & -1 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ -10 \end{bmatrix}$$

- Step 3: construct an argument matrix and reduce the argument matrix into echelon form by using the command ‘rref’.
- Here rref form is called the solution vector [x]

  [R,jb] = rref(A) also returns a vector jb such that:

  - r = length(jb) is this algorithm's idea of the rank of A.
  - x(jb) are the pivot variables in a linear system Ax = b.
  - A(:,jb) is a basis for the range of A.
  - R(1:r,jb) is the r-by-r identity matrix

MATLAB program:

```
Clc;
Clear all;
Close all;
A=[2 1 1;-1 1 -1;1 2 3];
B=[2; 3; -10]
Aug_mat=[A,B];
Crref=rref(B)
X= Crref(:,4);  %range of X.
```

Simulation:

X←enter

X= 3
    1
    -5

From X, x=3; y=1; z=-5

Example

$$3y + 2x = 6$$
$$5y - 2x = 10$$

We can eliminate the x-variable by addition of the two equations.

$$3y + 2x = 6$$
$$+\ 5y - 2x = 10$$
$$= 8y \quad = 16$$
$$y \quad = 2$$

The value of y can now be substituted into either of the original equations to find the value of x

$$3y + 2x = 6$$
$$3 \cdot 2 + 2x = 6$$
$$6 + 2x = 6$$
$$x = 0$$

The solution of the linear system is (0, 2).

✓ Finding eigen values and eigen vector

- An *eigenvalue* and *eigenvector* of a square matrix *A* are, respectively, a scalar *λ* and a nonzero vector *υ* that satisfy

$$A\upsilon = \lambda\upsilon \qquad (1)$$

Where *λ is a scalar*

- *The solution is usually obtained by first solving for the 'n' eigen values from the determinant equation |A-λI|=0, and then solving for the 'n' eigen vectors by substituting the corresponding eigen values in equation(1)*
- Steps to solve the linear equations as follows:
  - Step 1: enter the matrix 'A' and type

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

- Step 2: in the above step the output v is an n by m matrix whose column are eigen vectors and D is an n by n diagonal matrix that has the eigen values of A on its diagonal
- let

$$A=\begin{bmatrix} 0 & -6 & -14 \\ 6 & 2 & -16 \\ -5 & 20 & -10 \end{bmatrix}$$

- Performing [V,D]=eig(A)

produces

V =

1.0000	1.0000	-0.5571
0	0.0000	0.7428
0	0	0.3714

D =

1	0	0
0	1	0
0	0	3

MATLAB program:

```
Clc;
Clear all;
Close all;
A=[0 -6 14;6 2 -16;-5 20 -10];
[V,D]=eig(A);
Disp(X); X
```

✓ Solve Method:

- This method is also called as matrix method.
- To solve a system of linear equations using the symbolic math toolbox are
  - ✓Solve system of linear equations using 'linsolve'
  - ✓Solve system of linear equations using 'solve'

Linsolve Method:

A system of linear equations

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m$$

can be represented as the matrix equation $A \cdot \vec{x} = \vec{b}$, where A is the coefficient matrix,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

and $\vec{b}$ is the vector containing the right sides of equations,

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

- Example:

considering linear algebraic equations

$$2x+y+z=2$$

$$y-x=z+3$$

$$x+2y+3z+10=0$$

- Steps to solve the linear equations as follows:
  - Step 1: Re arrange the given system of linear algebraic equations as

$$2x+y+z=2$$

$$-x+y-z=3$$

$$x+2y+3z=10$$

Unknown values on Left hand side

known values on Right hand side

  - Step 2: In case the system of equations are not in the matrix form (Ax=B), use equationsTo Matrix – to convert the equations into matrix form

- Declare the system of equations:

```
Syms x y z
Eqn1=2*x+y+z==2;
Eqn2=-x+y-z==3;
Eqn3=x+2y+3z==10;
```

- Use equationsToMatrix to convert the equations into the form AX=B

```
[A,B]= equationsToMatrix ([eqn1, eqn2, eqn3],[x,y,z])
```

- Use linsolve to solve Ax=B for the vector of unknowns X.

```
X=linsolve(a,b);
X=
        3
        1
        -5
From X, x=3; y=1; z=-5
```

- Step 3: write a MATLAB program to solve linear algebraic equations

Program:

```
Clc;
Clear all;
Close all;
Syms x y z
Eqn1=2*x+y+z==2;
Eqn2=-x+y-z==3;
Eqn3=x+2y+3z==10;
[A,B]= equationsToMatrix ([eqn1, eqn2, eqn3],[x,y,z])
X=linsolve(A,B);
```

Simulaton:

```
X←enter
X=      3
        1
        -5
```

From X, x=3; y=1; z=-5

✓ Solve Method….

➢solve Method:

A system of linear equations

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2$$
$$\ldots$$
$$a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m$$

can be represented as the matrix equation $A \cdot \vec{x} = \acute{b}$, where A is the coefficient matrix,

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}$$

and $\vec{b}$ is the vector containing the right sides of equations,

$$\vec{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

- Example:

considering linear algebraic equations

$$2x+y+z=2$$

$$y-x=z+3$$

$$x+2y+3z+10=0$$

- Steps to solve the linear equations as follows:
  - Step 1: Re arrange the given system of linear algebraic equations as

$$2x+y+z=2$$

$$-x+y-z=3$$

$$x+2y+3z=10$$

Unknown values on Left hand side

known values on Right hand side

  - Step 2: In case the system of equations are not in the matrix form (Ax=B), use equationsTo Matrix – to convert the equations into matrix form

- Declare the system of equations:

```
Syms x y z
Eqn1=2*x+y+z==2;
Eqn2=-x+y-z==3;
Eqn3=x+2y+3z==10;
```

- Solve the system of equations using solve. The input to solve are a vector of equations and a vector of variables to solve the equations for

```
sol= solve ([eqn1, eqn2, eqn3],[x,y,z]);
xSol=sol.x
ySol=sol.y
zSol=sol.z
```

- The simulation results

```
xSol ←                  xsol=   3
ySol ←                  ysol=   1
zSol ←                  zsol=   -5
From X, x=3; y=1; z=-5
```

- Step 3: write a MATLAB program to solve linear algebraic equations

Program:

```
Clc;
Clear all;
Close all;
Syms x y z
Eqn1=2*x+y+z==2;
Eqn2=-x+y-z==3;
Eqn3=x+2y+3z==10;
sol= solve ([eqn1, eqn2, eqn3],[x,y,z]);
xSol=sol.x
ySol=sol.y
zSol=sol.z
```

Simulaton:

```
xSol ←                  xsol=   3
ySol ←                  ysol=   1
zSol ←                  zsol=   -5
```

From X, x=3; y=1; z=-5

✓ Cramer's Method

- The method of **solution of linear equations** by determinants is called **Cramer's Rule**. This rule for linear equations in 3 unknowns is a method of solving -by determinants- the following equations for x, y, z

$$a_1x + b_1y + c_1z = d_1$$
$$a_2x + b_2y + c_2z = d_2$$
$$a_3x + b_3y + c_3z = d_3$$

- Represent re –arranged system of equations in matrix form as

$$AX=B$$

$$\begin{bmatrix} a1 & b1 & c1 \\ a2 & b2 & c2 \\ a3 & b3 & c3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} d1 \\ d2 \\ d3 \end{bmatrix}$$

If $\Delta = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$ is the **determinant of coefficients** of x, y, z and is

assumed not equal to zero, then we may re-write the values as

$$x = \frac{\begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix}}{\Delta}, \quad y = \frac{\begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix}}{\Delta}, \quad \text{and} \quad z = \frac{\begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix}}{\Delta}$$

Solve this system using Cramer's Rule

$2x + 4y - 2z = -6$
$6x + 2y + 2z = 8$
$2x - 2y + 4z = 12$

$$\Delta = \begin{vmatrix} 2 & 4 & -2 \\ 6 & 2 & 2 \\ 2 & -2 & 4 \end{vmatrix} = 16 + 16 + 24 + 8 - 96 + 8 = -24$$

For x, take the determinant above and replace the first column by the constants on the right of the system. Then, divide this by the determinant:

$$x = \frac{\begin{vmatrix} -6 & 4 & -2 \\ 8 & 2 & 2 \\ 12 & -2 & 4 \end{vmatrix}}{\Delta} = \frac{-24}{-24} = 1$$

For y, replace the second column by the constants on the right of the system. Then, divide it by the determinant:

$$y = \frac{\begin{vmatrix} 2 & -6 & -2 \\ 6 & 8 & 2 \\ 2 & 12 & 4 \end{vmatrix}}{\Delta} = \frac{24}{-24} = -1$$

For z, replace the third column by the constants on the right of the system. Then, divide it by the determinant:

$$z = \frac{\begin{vmatrix} 2 & 4 & -6 \\ 6 & 2 & 8 \\ 2 & -2 & 12 \end{vmatrix}}{\Delta} = \frac{-48}{-24} = 2$$

- Steps to solve linear equations:
- Example:

considering linear algebraic equations

$$2x+y+z=2$$
$$y-x=z+3$$
$$x+2y+3z+10=0$$

- Steps to solve the linear equations as follows:
  - Step 1: Re arrange the given system of linear algebraic equations as

$$2x+y+z=2$$
$$-x+y-z=3$$
$$x+2y+3z=10$$

  - Step 2: Represent re –arranged system of equations in matrix form as

$$AX=B$$

Cramer’s rule produces the solution of a system **A x = B** as follows:

- Cramer's rule produces the solution of a system **A x = B** as follows:

The first unknown is given by **X = det(Ax) / det(A)**

The second by **Y = det(Ay) / det(A),**

The third by **Z = det(Az) / det(A),**

- Write a MATLAB program:

MATLAB program:

```
Clc;
Clear all;
Close all;
A=[2 1 1;-1 1 -1;1 2 3];
B=[2; 3; -10]
% construct auxiliary matrix
Ax=[2 1 1;3 1 -1; -10 2 3];
Ay=[2 2 1;-1 3 -1; 1 -10 3];
Az=[2 1 2; -1 1 3; 1 2 -10];
X=det(Ax)/det(A);
Y=det(Ay)/det(A);
Z=det(Az)/det(A);
```

Simulation:

X←enter x=3

Y←enter y=1

Z←enter z=-5

Matrix decomposition

- MATLAB provides built in functions for several matrix factorizations (decompositions)
- A **matrix decomposition** is a factorization of a matrix into some canonical form.
- A **canonical** form (often called **normal or standard** form) of an object is a standard way of presenting that object.
- There are several decompositions they are:
  - ✓ **LU decomposition**
  - ✓ **QR factorization**
  - ✓ **Cholesky decomposition**
  - ✓ **Singular value decomposition (svd)**

LU decomposition:

- The name of the built-in function is '**lu**'. To get the LU factorization of a square matrix ***M***,
- Type the command **[L,U] = lu(M)**.
- Matlab returns a **lower triangular matrix *L*** and an **upper triangular matrix *U*** such that L*U = M.
- It is also possible to det the permutation matrix as an output.
- Example: A = [10 -7 0; -3 2 6; 5 -1 5];

[L,U] = lu(A)

L = *3×3*

1.0000	0	0
-0.3000	-0.0400	1.0000
0.5000	1.0000	0

U = *3×3*

10.0000	-7.0000	0
0	2.5000	5.0000
0	0	6.2000

QR factorization

- The name of the appropriate built-in function for this purpose is '**qr**'.
- Typing the command **[Q, R] = qr(M)**
- returns an **orthogonal matrix** ($A^TA=I$ or $A^{-1}=A^T$)***Q*** and an **upper triangular matrix *R*** such that Q*R = M.

Cholesky decomposition

- If you have a **positive definite matrix *A***, you can factorize the matrix with the built-in function '**chol**'.
- The command **R = chol(A);**
- produces an **upper triangular matrix *R*** such that R'*R = A for a positive definite ***A***

Singular value decomposition (svd)

- This is another type of matrix decomposition and the name of the built-in function is ’**svd’**.
- Typing **[U,S,V] = svd(M);**
- produces a diagonal matrix ***S***, of the same dimension as ***M*** and with nonnegative diagonal elements in decreasing order, and unitary matrices(inverse =conjugate(A^{-1}=A*)) ***U*** and ***V*** so that M = U*S*V'.

Classification of Systems

- Depending upon the solution, the linear systems are classified into three types they are
  - ✓ Determined system (or)critically determined systems
  - ✓ Under determined systems
  - ✓ Over determined systems

Determined system (or)critically determined systems:

- These are the linear systems in which the number of unknown variables are equal to the number of equations.
- For determined (or) critically determined system, an unique and perfect solutions will exist.
- For example:

$$5x-3y+2z=10$$
$$-3x+8y+4z=20$$
$$2x+4y-9z=9$$

Note: for determined system also an unique and perfect solution exist if and only if the determinant of the coefficient matrix is non-zero

Under determined systems:

- These are the systems of liner algebraic equations in which the number of unknown variables are greater than the number of equations [(or) number of equations ate less than the number of unknown variables.
- For these systems an unique and perfect solutions will not exists.
- An infinite number of solutions will exist and also it is required to represent one unknown variable as a function of another unknown variable
- We can solve the under determined sstems by using any one of the conventional method.

$$5x-3y+2z=10$$

$$-3x+8y+4z=20$$

Over determined systems:

- These are the system of linear algebraic equations in which the number of equations are greater than the number of unknown variables.
- For these type of systems an unique and perfect solutions will exist, but along with the solutions some additional zeros will be generated.
- For example

$$5x-3y+2z=10$$

$$-3x+8y+4z=20$$

$$x+4y-9z=9$$

$$-4x+6y+10z=8$$

- we can solve these systems also by using any one of the existing solution method

Order of the systems

- All the linear systems are the first order systems, where as the higher order systems will have the higher order powers of the independent variables.
- If the order of the independent variable is n, then that system have ‘n’ solutions for that independent variable.
- Using MATLAB we can solve the quadratic (order-2) systems and higher order systems easily.
- We can solve the ordinary differential equations and parial differential equations of any order and any degree by using the MATLAB’s simple and straight forward inbuilt functions.
- Example: solution for the quadratic equation

$$x^2+5x+6=0 \quad \text{----->} \quad x=-2;x=-3$$