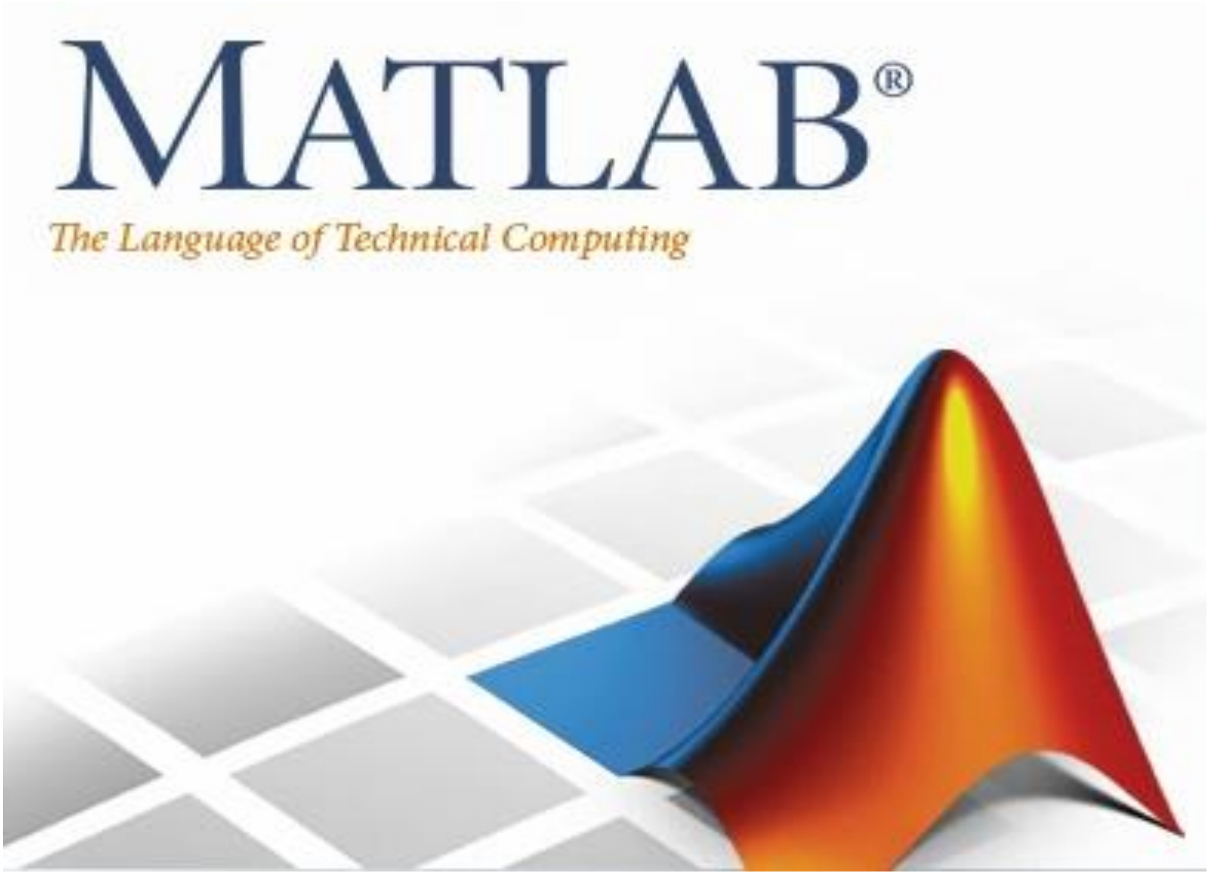


Lecture 1

Introduction to MATLAB & Matrices

By: Abidulkarim K. I. Yasari

Millions of engineers and scientists worldwide use MATLAB to analyze and design the systems and products transforming our world. MATLAB is in automobile active safety systems, spacecraft, health monitoring devices, smart power grids, and LTE cellular networks. It is used for machine learning, signal processing, image processing, computer vision, communications, computational finance, control design, robotics, and much more.

The MATLAB platform is optimized for solving engineering and scientific problems. The matrix-based MATLAB language is the world’s most natural way to express computational mathematics. Built-in graphics make it easy to visualize and gain insights from data. A vast library of pre-built toolboxes lets you get started right away with algorithms essential to your domain. The desktop environment invites experimentation, exploration, and discovery. These MATLAB tools and capabilities are all rigorously tested and designed to work together.

MATLAB helps you take your ideas beyond the desktop. You can run your analyses on larger data sets, and scale up to clusters and clouds. MATLAB code can be integrated with other languages, enabling you to deploy algorithms and applications within web, enterprise, and production systems.

MATLAB Key Features:

- High-level language for scientific and engineering computing.
- Desktop environment tuned for iterative exploration, design, and problem-solving.
- Graphics for visualizing data and tools for creating custom plots.
- Apps for curve fitting, data classification, signal analysis, control system tuning, and many other tasks.
- Add-on toolboxes for a wide range of engineering and scientific applications.
- Tools for building applications with custom user interfaces.
- Interfaces to C/C++, Java, .NET, Python, SQL, Hadoop, and Microsoft Excel.
- Royalty-free deployment options for sharing MATLAB programs with end users

MATLAB is a software package for mathematical calculations. It is a very powerful package but is also very simple to use. One of the attractions of MATLAB is its versatility. You can use it interactively or use it as a programming language. It can handle everything from a simple expression to a set of complex mathematical calculations on large sets of data. There is a massive number of predefined functions to choose from. There is also a large selection of simple to use graphics functions to plot and display data to the screen.
The fundamental unit of MATLAB is a matrix. In fact, MATLAB is short for MATrix LABoratory. However, its use is not restricted to matrix mathematics. Matrices in MATLAB can also be regarded as arrays of numbers. You can see matrices as a convenient way of handling groups of numbers. A matrix in MATLAB can have one, two or more dimensions or be empty. A matrix with a single element is a special case. They are treated as a single number. A matrix element can be an integer, a real or a complex number. You do not have to worry about element types. MATLAB will set the element type to what is required. Matrices that contain a single row or column are called vectors.

- MATLAB treats all variables as matrices. For our purposes a matrix can be thought of as an array, in fact, that is how it is stored.

- Vectors are special forms of matrices and contain only one row OR one column.

- Scalars are matrices with only one row AND one column.

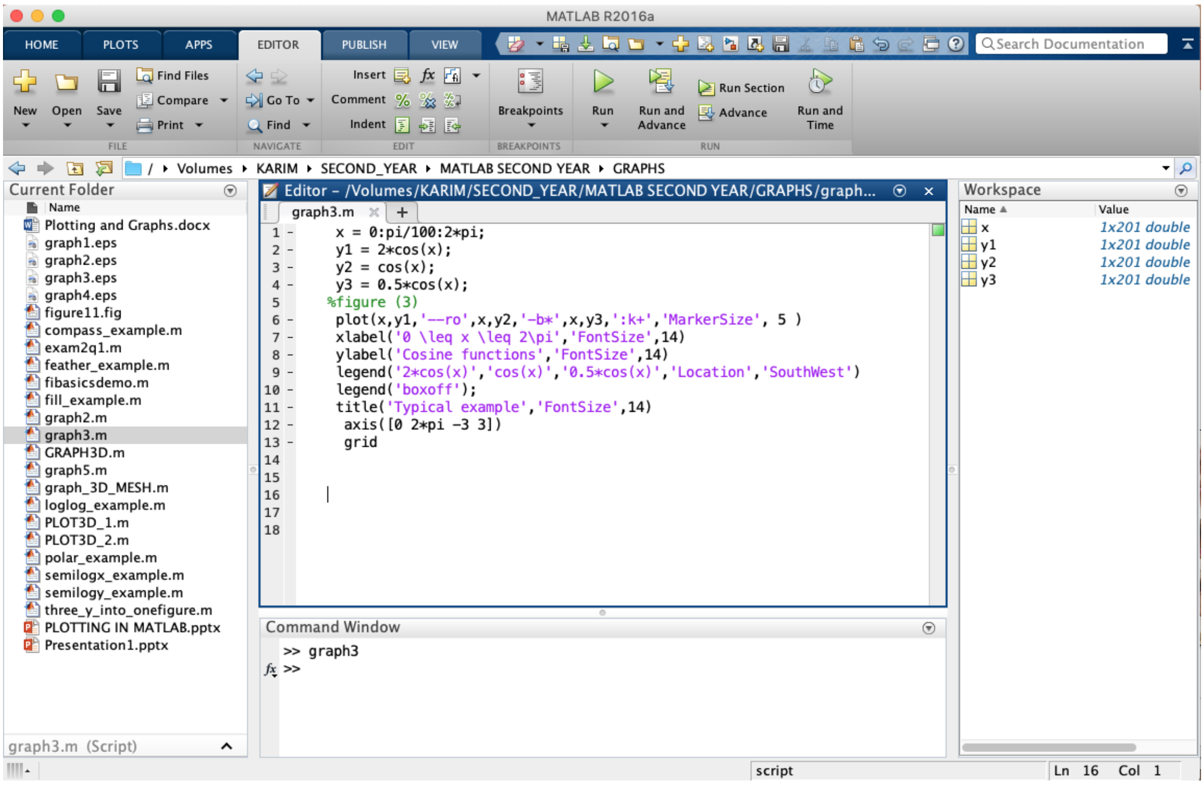
- When specifying the position of an element or the size of a matrix, the convention is to specify the row before the column.

Starting MATLAB:

Normally when MATLAB is installed a MATLAB icon is installed onto the desktop. Double click on the icon to start MATLAB.
If there is no MATLAB icon on the desktop, then MATLAB can be started in other ways. On a PC, select the following in turn:-

Start
All Programs
MATLAB R201X

By now you should have a MATLAB window open in front of you:



```
   x = 0:pi/100:2*pi;
   y1 = 2*cos(x);
   y2 = cos(x);
   y3 = 0.5*cos(x);
  %figure (3)
   plot(x,y1,'--ro',x,y2,'-b*',x,y3,':k+','MarkerSize', 5 )
   xlabel('0 \leq x \leq 2\pi','FontSize',14)
   ylabel('Cosine functions','FontSize',14)
   legend('2*cos(x)','cos(x)','0.5*cos(x)','Location','SouthWest')
   legend('boxoff');
   title('Typical example','FontSize',14)
    axis([0 2*pi -3 3])
    grid
```

```
>> graph3
>>
```

Name	Value
x	1x201 double
y1	1x201 double
y2	1x201 double
y3	1x201 double

- The MATLAB environment is command-oriented; a prompt appears on the screen, and a MATLAB statement can be entered. When the <ENTER> key is pressed, the statement is executed, and another prompt appears.

- If a statement is terminated with a semicolon (;), no results will be displayed. Otherwise, results will appear before the next prompt.

Other MATLAB symbols:

>>		prompt
. . .		continue statement on the next line
,		separates statements and data
%		start comment which ends at the end of the line
;	(1)	suppress output
	(2)	used as a row separator in a matrix
:		specify range

MATLAB Mathematical & Assignment Operators:

Power	^	or	.^	for example	a ^ b	or	a .^ b
Multiplication	*	or	.*	for example	a * b	or	a .* b
Division	/	or	. /	for example	a / b	or	a . / b
or	\	or	.\	for example	b \ a	or	b .\ a

NOTE: 56 / 8 = 8 \ 56

- (unary minus)	+ (unary plus)		uminus and uplus
Addition	+	a + b	
Subtraction	-	a - b	
Assignment	=	a = b	(assign b to a)

The order of precedence is that powers are evaluated first, followed by multiplication and division, with addition and subtraction last. Round brackets can be used to change the order of precedence.
Calculations within brackets take precedence over everything else.

If you do not assign an expression to a variable, it is automatically assigned to a variable called “ans”, which can be used in subsequent expressions.

When there are two items with the same level of precedence, the order is from left to right.

```
>> 12 / 3 * 4
```

This means

$\frac{12}{3} *4$ and not $\frac{12}{3*4}$

because the division operation is to the left of the multiplication operation.

MATLAB Relational Operators:

- MATLAB supports six relational operators.

Less Than	<
Less Than or Equal	<=
Greater Than	>
Greater Than or Equal	>=
Equal To	==
Not Equal To	~=

MATLAB Logical Operators:

- MATLAB supports three logical operators.

not	~	% highest precedence
and	&	% equal precedence with or
or	|	% equal precedence with and

Truth Table for Logical Operations:

The following reference table shows the results of applying the binary logical operators to a series of logical 1 (true) and logical 0 (false) scalar pairs. To calculate NAND, NOR or

XNOR logical operations, simply apply the logical NOT operator to the result of a logical AND, OR, or XOR operation, respectively.

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

MATLAB Variable Names:

- Variable names ARE case sensitive
- Variable names can contain up to 63 characters (as of MATLAB 6.5 and newer)
- Variable names must start with a letter followed by letters, digits, or underscores.

MATLAB Special Variables:

ans	Default variable name for results
pi	Value of π
eps	Smallest incremental number
inf	Infinity
NaN	Not a number e.g. 0/0
i and j	i = j = square root of -1
realmin	The smallest usable positive real number
realmax	The largest usable positive real number

Some Useful MATLAB commands:

who	List known variables (>> who)
whos	List known variables plus their size (>> whos)
help	>> help sqrt Help on using sqrt
lookfor	>> lookfor sqrt Search for keyword sqrt in m-files
what	>> what foldername List MATLAB files in folder
clear	Clear all variables from work space
clear x y	Clear variables x and y from work space
clc	Clear the command window
dir	List all files in current directory (all types)
ls	Same as dir
type test	Display test.m in command window
delete test	Delete test.m
cd foldername	Change directory to foldername
chdir a	Same as cd a

- pwd	Show current directory
- which test	Display current directory path to test.m

A Useless, But Interesting, MATLAB command:

- why	In case you ever needed a reason

Mathematical Built-in Functions:

There are hundreds of functions in MATLAB. To get started considering the standard mathematical functions that you would expect to find on a calculator.

- abs(x)	The absolute value or complex magnitude.
- sqrt(x)	The square root.
- round(x)	The value rounded to the nearest integer.
- rem(x,b)	The remainder of x divided by b.
- sin(x)	The sine.
- cos(x)	The cosine.
- tan(x)	The tangent.
- asin(x)	The arcsine. The inverse of sin(x)
- acos(x)	The arccosine.
- atan(x)	The arctangent.
- exp(x)	The exponential base e.
- log(x)	The natural logarithm. ie to the base e
- log10(x)	The log base 10.

MATLAB Logical Functions:

- MATLAB also supports some logical functions.

xor (exclusive or) Ex: xor (a, b)

Where a and b are logical expressions. The xor operator evaluates to true if and only if one expression is true and the other is false. True is returned as 1, false as 0.

any (x)	returns 1 if any element of x is nonzero
all (x)	returns 1 if all elements of x are nonzero
isnan (x)	returns 1 at each NaN in x
isinf (x)	returns 1 at each infinity in x
isfinite (x)	returns 1 at each finite value in x

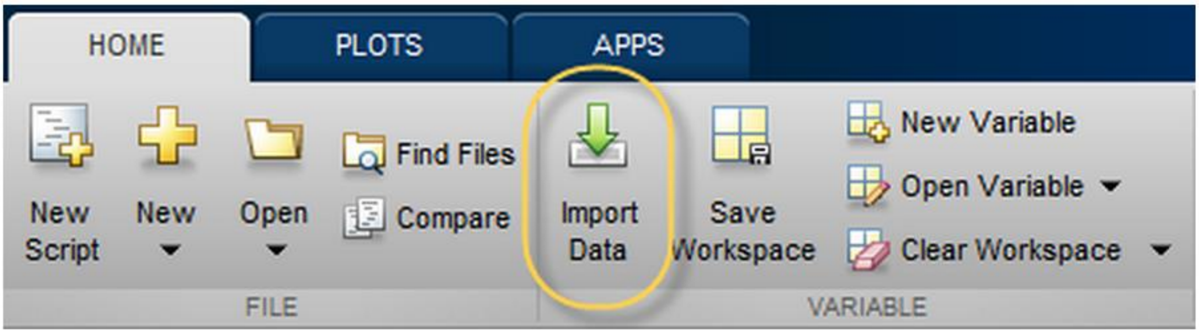
MATLAB Display formats:

- MATLAB supports 8 formats for outputting numerical results.

format long	16 digits
format short e	5 digits plus exponent
format long e	16 digits plus exponent
format hex	hexadecimal
format bank	two decimal digits
format +	positive, negative or zero
format rat	rational number (215/6)
format short	default display

MATLAB Files:

Text, spreadsheets, images, scientific data, audio and video, XML documents Standard file format functions read data from popular file formats, such as Microsoft Excel spreadsheets, text, images, audio and video, and scientific data formats. You can read many of these formats by selecting **Import Data** on the Home tab.



MATLAB can load and save data in several different file formats. It has its own binary data format that can only really be understood by MATLAB. Files of this type have a ".mat" extension.

In MATLAB, programs may be written and saved in Files with a suffix .m called M-Files. There are two types of M-File programs: **functions** and **scripts**.

MATLAB can also read in other file formats. For more information, enter:

```
doc 'Supported File Formats'
```

in the command window.

To Get Started:

To get started, type these commands in the command window:

```
>> a=10;                    % press enter
>> b=10/2                   % press enter
b =
    5
>>
```

After each execution, the prompt will appear at the following line in the command window.

MATLAB Matrices:

To enter the contents of a matrix:

- Enclose the contents in square brackets [ ].
- Separate each element by white space or a comma “,”.
- Separate each row by a semicolon “;” or a new line.

You can increase the size of a matrix by adding rows or columns.
For Example:

```
>> x=[1,2,3]
x =
    1     2     3
>> x = [ x ; 10 11 12]
x =
    1     2     3
   10    11    12
```

When adding a row to matrix, the number of elements in the additional row must be the same as the number of elements in each row of the matrix. The same is true for columns.
When you enter a matrix, each of the elements can be entered as an expression.

```
 >> a = [exp(0), sqrt(4), 1+2]
a =
        1        2        3
```

- A matrix with only one row AND one column is a scalar. A scalar can be created in MATLAB as follows:

```
» a_value=23                    % a_value is a scalar
a_value =
   23
```

- A matrix with only one row is called a row vector. A row vector can be created in MATLAB as follows (note the commas):
-

```
» rowvec = [1 ,  2 , 3 , 4]              % row vector
   1    2    3   4
```

OR

```
» rowvec = [1   2  3  4]
rowvec =
   1    2    3   4
```

OR

```
>> a = 1:4                      % : range from 1 to 4
a =
     1     2     3     4
```

OR

```
>> a = [1:4]
a =
     1   2   3   4
```

- A matrix with only one column is called a column vector. A column vector can be created in MATLAB as follows (note the semicolons):

```
» colvec = [4 ; 3 ; -2 ; 1]
```

```
colvec =
    4
    3
    -2
    1
```

- A matrix can be created in MATLAB as follows (note the commas AND semicolons):

```
» matrix = [1 , 2 , 3 ; 4 , 5  , 6 ; 7 , 8 , 9]

matrix =

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

- A portion of a matrix can be extracted and stored in a smaller matrix by specifying the names of both matrices and the rows and columns to extract. The syntax is:

```
sub_matrix = matrix ( r1 : r2 , c1 : c2 ) ;
```

where r1 and r2 specify the beginning and ending rows and c1 and c2 specify the beginning and ending columns to be extracted to make the new matrix.

A column vector can be extracted from a matrix. As an example, we create a matrix below:

```
» matrix = [1,2,3;4,5,6;7,8,9]

matrix =

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

Here we extract column 2 of the matrix and make a column vector:

```
» col_two = matrix( : , 2)                % here : means all rows, 2 is the second column

col_two =

     2
     5
     8
```

A row vector can be extracted from a matrix. As an example, we create a matrix below:

```
» matrix = [1,2,3;4,5,6;7,8,9]

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

Here we extract row 2 of the matrix and make a row vector. Note that the 2:2 specifies the second row and the 1:3 specifies which columns of the row.

```
» rowvec = matrix(2 : 2 , 1 : 3)

rowvec =

    4     5     6
```

If:

```
>> a = [1 2 3 4; 5 6 7 8 ;9 10 11 12; 13 14 15 16]
           a =
                    1      2      3      4
                    5      6      7      8
                    9      10     11     12
                    13     14     15     16
```

Then b:

```
>> b = a(:,3)                        % the 3rd column is extracted from matrix a

          b =
                    3
                    7
                    11
                    15
```

or b:

```
>> b = a(2,:)                        % the 2nd  row is extracted from matrix a

          b =
                    5   6   7   8
```

or b:

```
>> b = a(1:3,1:3)                    % extracts rows from 1 to 3, and columns from 1 to 3
          b =
                    1     2      3
                    5     6      7
                    9     10     11



>> b(2,3) = 0                        % change the value of the 3rd element in the second row to zero
          b =
                    1     2      3
                    5     6      0
                    9     10     11
>> b(3,:) = 0                        % change the value of all elements oft he third row to zero
          b =
                    1     2      3
```

```
                  5     6      0
                  0     0      0

>> b(4,:) = [4 6 7]                      % add row 4 to the matrix b
b =
             1     2     3
             5     6     0
             0     0     0
             4     6     7

>> b(:,4) = [7;8;9;5]                    % add column 4 to the matrix b
b =
             1     2     3     7
             5     6     0     8
             0     0     0     9
             4     6     7     5

>> H = reshape(b,1,16)                   % change matrix b dimantions to 1X16
ans =
         1    5     0     4     2     6     0     6     3     0     0     7     7     8     9    5

>> N = reshape(H,2,8)                    % change matrix b dimantions to 2X8
N =
          1     0     2     0     3     0     7     9
          5     4     6     6     0     7     8     5

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

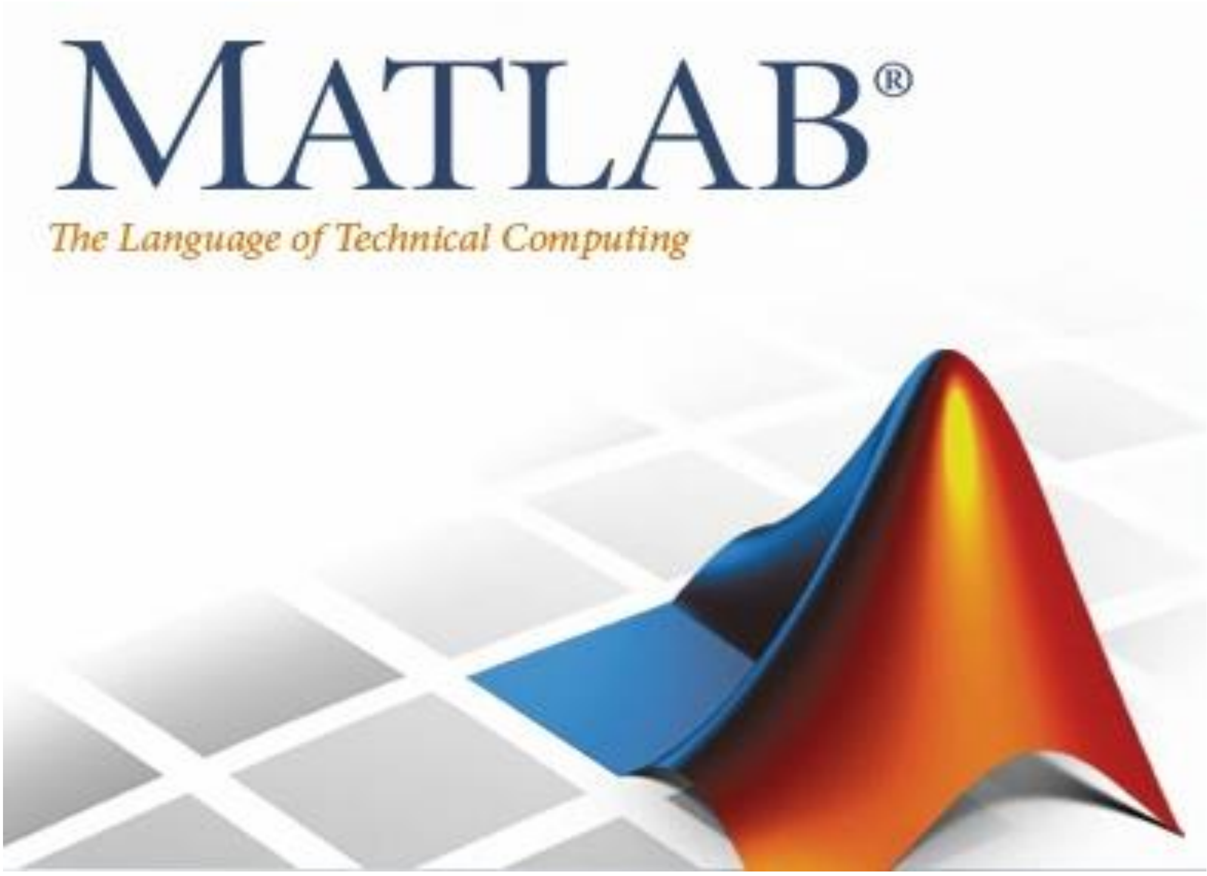
magic(N) is an N-by-N matrix constructed from the integers 1 through N^2 with equal row, column, and diagonal sums. Produces valid magic squares for all N > 0 except N = 2.

```
>> eye(4)
ans =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

eye : Identity matrix. eye(N) is the N-by-N identity matrix.
x = eye(2,3)

```
>> randi([2,6],[4,8])        % 2 is the min element value, 6 is the max element value
```

```
ans =                          % 4 number of rows, 8 number of columns
     3     5     5     4     2     5     5     2
     5     4     5     2     4     4     2     6
     2     4     6     3     2     4     2     6
     2     6     6     6     6     2     4     5

>> b(2:3,:) = 1             % Change the values of the 2nd and the 3rd row in all columns to 1

>> b(1:end,1) = 0              % Change all first column values to 0
>> save('karim')               % to save the workspace as karim.mat file
>> clear all                   % to clear workspace
>> load('karim.mat')           % to load karim.mat file
```



Lecture 2

Mathematical Operations

By: Abidulkarim K. I. Yasari

Matrix addition:

Use + to add two matrices or to add a scalar to an array (matrix)

```
» a=3;

» b=[1, 2, 3;4, 5, 6]
b =
    1    2    3
    4    5    6

» c= b+a                          % Add a to each element of b
c =
    4    5    6
    7    8    9

>> A = [ 1  2 ; 3  4];
>> B = [ 5  6 ; 7  8];
>> A+5                                %This adds 5 to each element of A.

[ 1  2 ; 3  4] + 5 = [1+5, 2+5; 3+5, 4+5]

ans =
         6    7
         8    9
```

```
 >> A + B                  %This adds each element of A to the corresponding element in B.

         A + B = [ 1  2 ; 3  4] + [ 5  6 ; 7  8] = [ 1+5  2 +6; 3+7  4+8] = [ 6 8; 10 12]

ans =
         6    8
         10    12
```

When adding two arrays *A* and *B*, MATLAB adds the corresponding elements, i.e.,

- It adds the element in the first row and first column of *A* to the element in the first row and column of *B*
- It adds the element in the first row and second column of *A* to the element in the first row and second column of *B*, etc.

This called *elementwise addition*

EXAMPLE:

For c a scalar and $A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$

$$A + c = \begin{bmatrix} A_{11} + c & A_{12} + c & A_{13} + c \\ A_{21} + c & A_{22} + c & A_{23} + c \end{bmatrix}$$

For $A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$ and $B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix}$

$$A + B = \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} & A_{13} + B_{13} \\ A_{21} + B_{21} & A_{22} + B_{22} & A_{23} + B_{23} \end{bmatrix}$$

Matrix Subtraction:

Use – to subtract one array from another or to subtract a scalar from an array

```
» a=3;

» b=[1, 2, 3;4, 5, 6]
b =
    1    2    3
    4    5    6
```

```
» c = b - a                    %Subtract a from each element of b
c =
   -2   -1    0
    1    2    3
```

```
>> A – B        %This subtracts each element of B from the corresponding element in A.
```

A - B = [1 2 ; 3 4] - [5 6 ; 7 8] = [1-5 2-6; 3-7 4-8] = [-4 -4; -4 -4]

```
ans  =
         -4      -4
         -4      -4
```

EXAMPLE:

For c a scalar and $A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$

$$A - c = \begin{bmatrix} A_{11} - c & A_{12} - c & A_{13} - c \\ A_{21} - c & A_{22} - c & A_{23} - c \end{bmatrix}$$

For $A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$ and $B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix}$

$$A - B = \begin{bmatrix} A_{11} - B_{11} & A_{12} - B_{12} & A_{13} - B_{13} \\ A_{21} - B_{21} & A_{22} - B_{22} & A_{23} - B_{23} \end{bmatrix}$$

- When subtracting two arrays *A* and *B*, MATLAB performs an elementwise subtraction

Matrix multiplication:

There are two ways of multiplying matrices – matrix multiplication and elementwise multiplication.

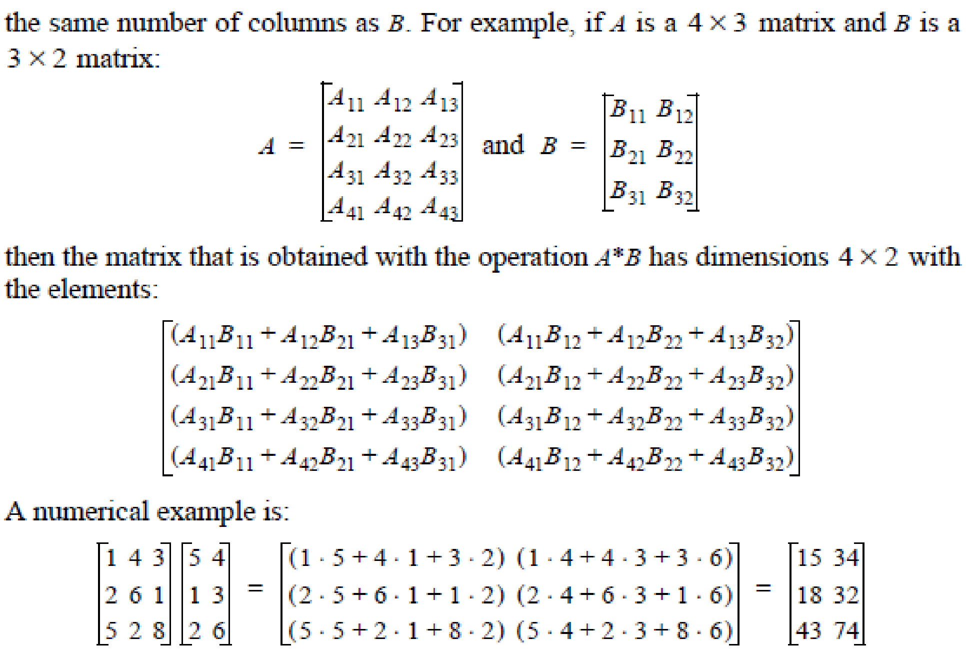
Matrix multiplication:

- Type used in linear algebra
- MATLAB denotes this with asterisk (*)
- Number of columns in left matrix must be same as number of rows in right matrix

```
>> A * 5                       %This multiplies each element in A by 5

 >> A * B                      % Multiply matrix A by B
```

A * B = [1 2 ; 3 4] * [5 6 ; 7 8] = [1*5+2*7, 1*6 + 2*8; 3*5 + 4+7, 3*6 + 4*8]

```
ans  =
          19       22
          43       50
```



the same number of columns as B. For example, if A is a 4×3 matrix and B is a 3×2 matrix:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{bmatrix} \text{ and } B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix}$$

then the matrix that is obtained with the operation $A*B$ has dimensions 4×2 with the elements:

$$\begin{bmatrix} (A_{11}B_{11}+A_{12}B_{21}+A_{13}B_{31}) & (A_{11}B_{12}+A_{12}B_{22}+A_{13}B_{32}) \\ (A_{21}B_{11}+A_{22}B_{21}+A_{23}B_{31}) & (A_{21}B_{12}+A_{22}B_{22}+A_{23}B_{32}) \\ (A_{31}B_{11}+A_{32}B_{21}+A_{33}B_{31}) & (A_{31}B_{12}+A_{32}B_{22}+A_{33}B_{32}) \\ (A_{41}B_{11}+A_{42}B_{21}+A_{43}B_{31}) & (A_{41}B_{12}+A_{42}B_{22}+A_{43}B_{32}) \end{bmatrix}$$

A numerical example is:

$$\begin{bmatrix} 1 & 4 & 3 \\ 2 & 6 & 1 \\ 5 & 2 & 8 \end{bmatrix} \begin{bmatrix} 5 & 4 \\ 1 & 3 \\ 2 & 6 \end{bmatrix} = \begin{bmatrix} (1\cdot 5+4\cdot 1+3\cdot 2) & (1\cdot 4+4\cdot 3+3\cdot 6) \\ (2\cdot 5+6\cdot 1+1\cdot 2) & (2\cdot 4+6\cdot 3+1\cdot 6) \\ (5\cdot 5+2\cdot 1+8\cdot 2) & (5\cdot 4+2\cdot 3+8\cdot 6) \end{bmatrix} = \begin{bmatrix} 15 & 34 \\ 18 & 32 \\ 43 & 74 \end{bmatrix}$$

```
» a=3;

» b=[1, 2, 3; 4, 5, 6]
b =
     1     2     3
     4     5     6

» c = a * b                          % Multiply each element of b by a
c =
     3     6     9
    12    15    18
```

Note: When using two arrays, they must both have the same dimensions (number of rows and number of columns)

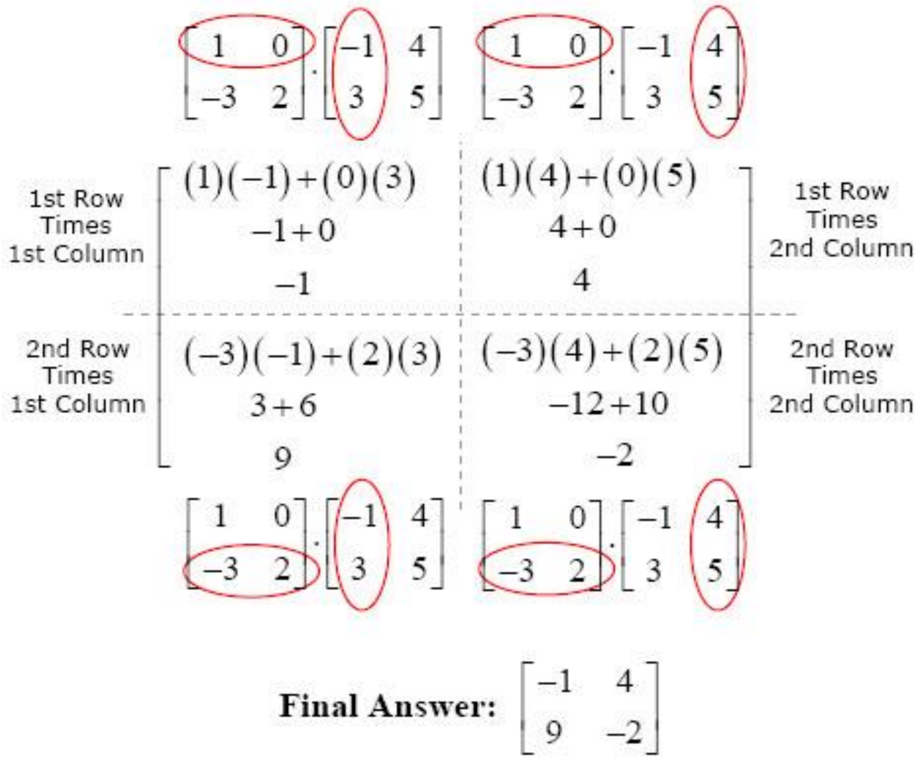
```
>> X = [2   4   1]
X =
          2     4     1


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

```
>> Z = [3 -4 2; 2 3 1; 1 4 0]
Z =
          3    -4     2
          2     3     1
          1     4     0

>> C = X * Y
C =
        45    37    35

>> C = Y * Z
C =
        19     2    10
        43    14    22
        34    11    19

>> C = Y .* Z
C =
        12    -8     6
        16    18     7
         5    36     0
```

When performing matrix multiplication on two square matrices

- They must both have the same dimensions
- The result is a matrix of the same dimension
- In general, the product is not commutative, i.e.,

$$A*B \neq B*A$$



```
>> A = randi(3,3)
A =
     3     3     1
     3     2     2
     1     1     3

>> B=randi(3,3)
B =
     3     3     1
     1     2     2
     3     3     3

>> AB = A*B
AB =
    15    18    12
    17    19    13
    13    14    12

>> BA = B*A
BA =
    19    16    12
    11     9    11
    21    18    18

>> AB == BA
ans =
     0     0     1
     0     0     0
     0     0     0
```

When performing matrix multiplication on two vectors

- They must both be the same size
- One must be a row vector and the other a column vector
- If the row vector is on the left, the product is a scalar
- If the row vector is on the right, the product is a square matrix whose size is the same size as the vectors

```
>> h = [2  4  6]
h =
     2     4     6

>> v = [-1  0  1]'
v =
    -1
     0
     1

>> h * v
ans =
     4
```

```
>> v * h
ans =
    -2    -4    -6
     0     0     0
     2     4     6
```

Elementwise Multiplication:

- Use .* to get elementwise multiplication (notice period before asterisk)
- Both matrices must have the same dimensions

```
>> A = [1  2; 3  4];
>> B = [0  1/2; 1  -1/2];

>> C = A .* B
    C =
           0          1
           3         -2
```

If matrices not same dimension in elementwise multiplication, MATLAB gives error

```
>> A = [ 1  2; 3  4];
>> B = [1  0]';
>> A .* B                                  % Meant matrix multiplication!
                                           ??? Error using ==> times
                                           Matrix dimensions must agree.
>> A * B                                   % this works
ans =
           1
           3
```

Be careful – when multiplying square matrices:

- Both types of multiplication always work

- If you specify the wrong operator, MATLAB will do the wrong computation and there will be no error!

- Difficult to find this kind of mistake

EXAMPLE:

```
>> A = [1  2; 3   4];
>> B = [0   1/2; 1   -1/2];

>> A .* B
 ans
           0    1
           3   -2
```

```
>> A * B
ans =
          2.0000     -0.5000
          4.0000     -0.5000
```

Matrix Division:

```
» a=3;
» b=[1, 2, 3; 4, 5, 6]
b =
    1     2     3
    4     5     6

» c = b / a                          % Divide each element of b by a
c =
   0.3333    0.6667    1.0000
   1.3333    1.6667    2.0000
```

Left division \:

Left division is one of MATLAB's two kinds of array division

- Used to solve the matrix equation $AX=B$
  - A is a square matrix, X, B are column vectors
  - Solution is $X = A_{-1}B$

In MATLAB, solve by using left division operator (\), i.e.,

```
>> X = A \ B
```

Right division /:

Right division is the other kind of MATLAB's array division

- Used to solve the matrix equation $XC=D$
  - C is a square matrix, X, D are row vectors
  - Solution is $X = D \cdot C_{-1}$

In MATLAB, solve by using right division operator (/), i.e.,

```
>> X = D / C
```

Mathematical Functions:

Trigonometric functions all use radians. However, there are degree-based versions of each of the functions. Their name is the same as the functions above, except for and extra "d" at the end of the name.

```
>> sind(45)
```

Functions can be used in any expression in any place you could put a number or a variable.

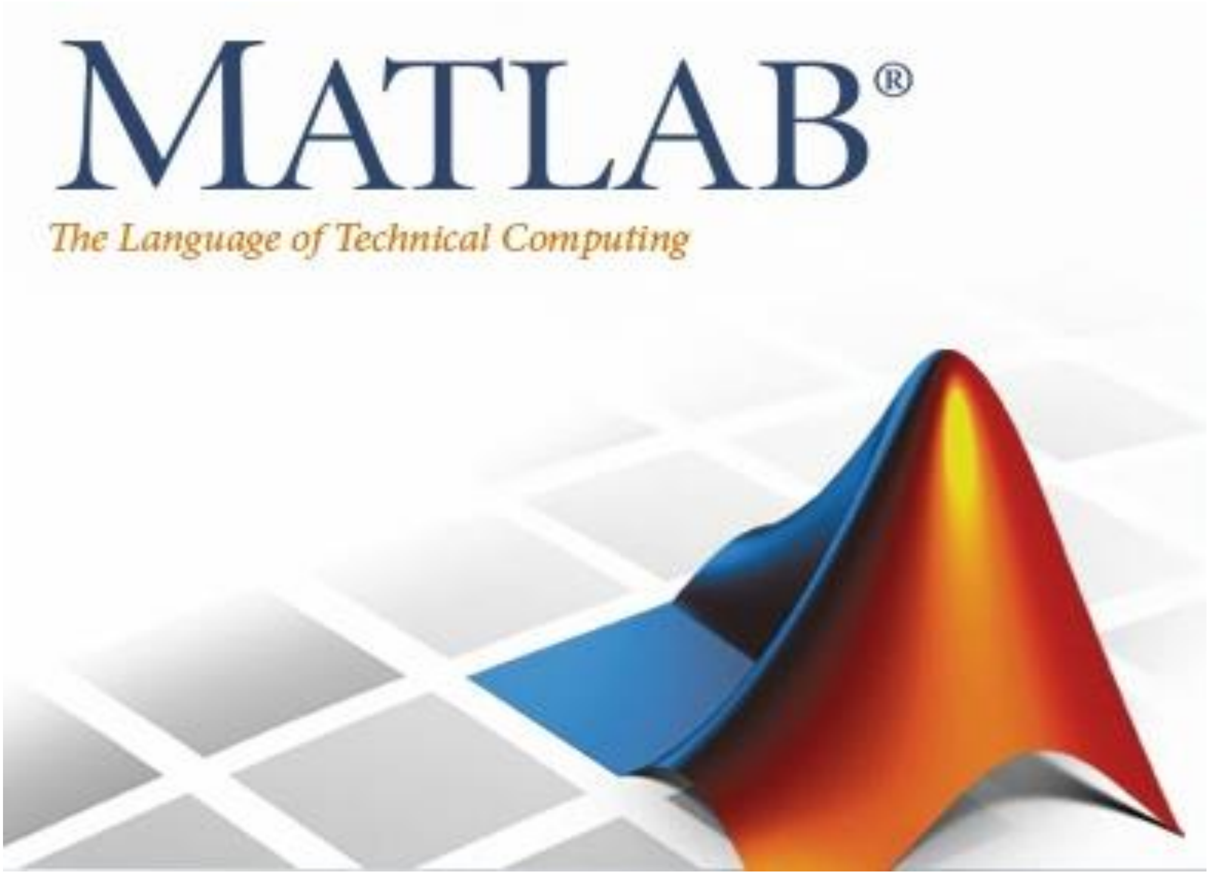
```
>> sin(x)^2 + cos(x)^2

>> 10^(log10(3) + log10(4))
```

The argument of a function (x and b in the table above) can also be an expression.

```
>> r = sqrt(3^2+4^2)
```

The argument can also contain calls to other functions.

```
>> exp(2*log(3) + 3*log(2))
```



Lecture 3

File Usage in MATLAB & Some Built-in Functions

By: Abidulkarim K. I. Yasari

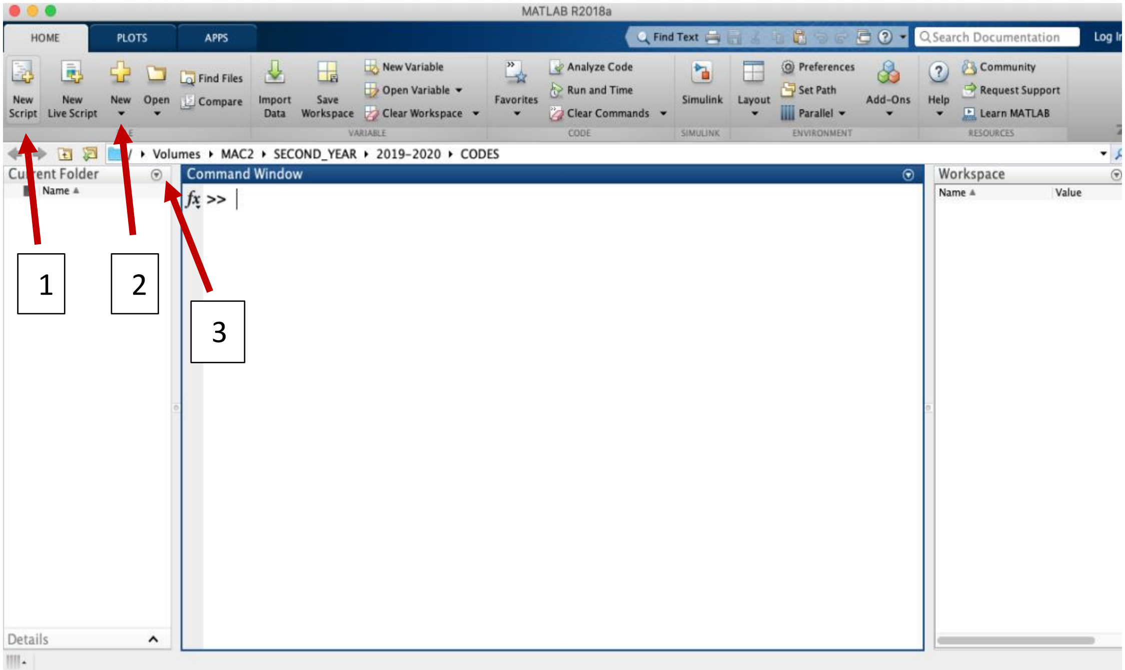
File Usage in MATLAB:

Now we will learn how to create a new MATLAB script [also we call it (.m) file], then how to edit, save, and run this file. And next how to open all (.m) files which already created in MATLAB.

There are five ways to create a new script, or (.m) file:

1. Click the Home tab then click the (New Script) button, which is located at the top-left at the home tab. A new script (.m) file will be opened in the editor window, its name by default is untitled.m

2. Click the new button at the home tab then choose (Script).

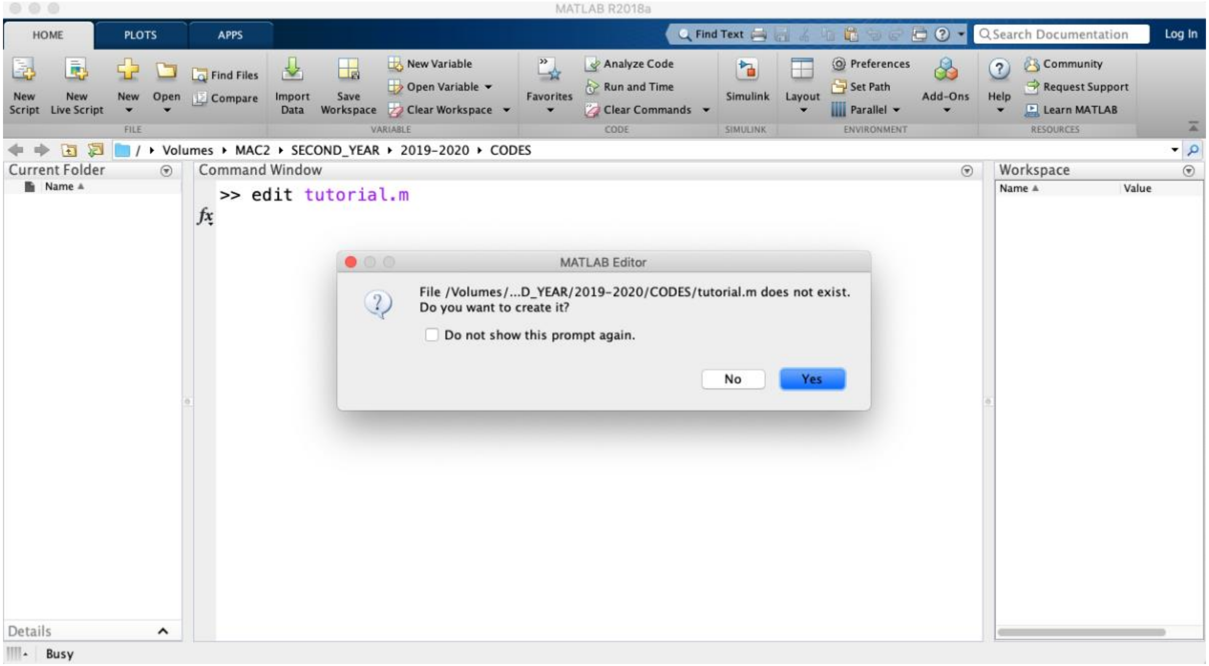


3. Click the drop-down menu at the current folder tab, then choose (New Folder).
4. If you know in advance what to name the new script file, type in the command window the following command if the file name for example is tutorial.m

```
>>edit tutorial.m
```

Then click enter, a new pop-up window will appear and there is a message there says: File/…../tutorial.m does not exist. Do you want to create it? By clicking Yes, a new file will be created and opened in the editor window under the name tutorial.m.

As shown in the following Figure.



```
>> edit tutorial.m
```

File /Volumes/...D_YEAR/2019-2020/CODES/tutorial.m does not exist.
Do you want to create it?

□ Do not show this prompt again.

5. If the editor window is already opened and there is one or more script files already opened, click the (+) button next to the last tab of opened scripts.

After closing the file: how to open it in the editor window?

First, check that the path of the current folder is showing that the tutorial.m file is in this current folder. To open this file:

Either to go to the current folder and double click on tutorial.m file, then it will be opened in the editor window. Or to type in the command window the following command:

```
>> edit tutorial.m
```

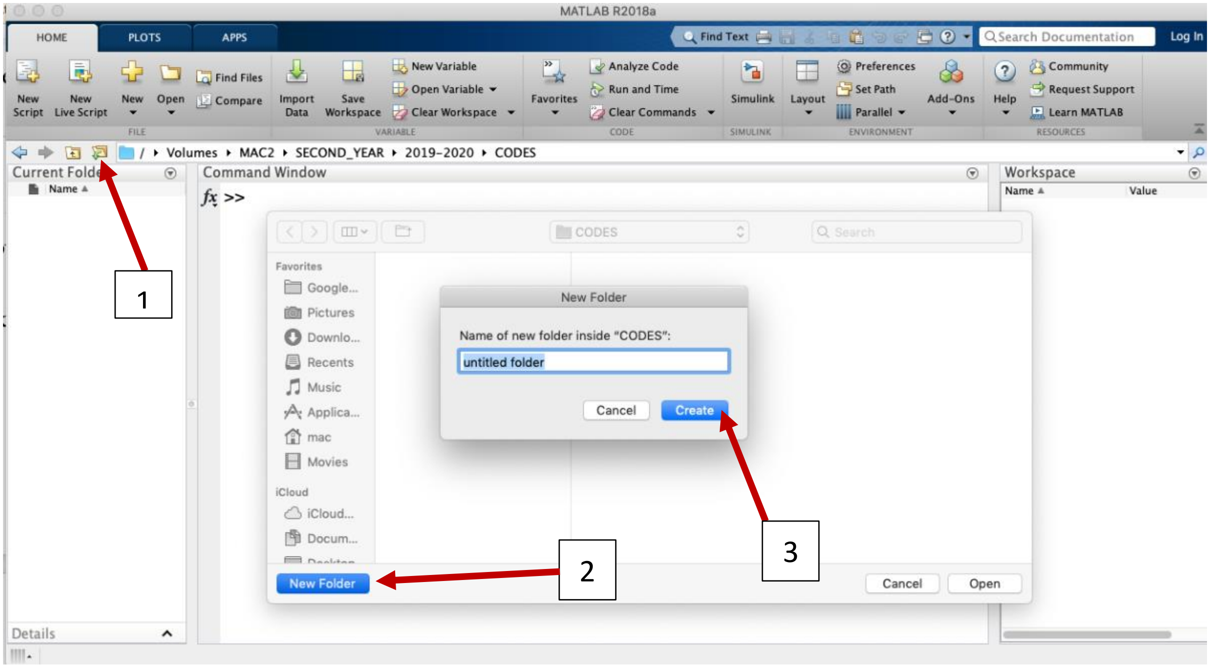
Then click enter. The tutorial.m file will be opened in the editor window.

So there are two uses of the function edit:

1. To create a completely new script file.
2. To open the already saved script file.

How to create a new folder:

To create a new folder in the current folder, go to the current folder window and click the drop-down menu at the current folder and choose (New Folder). Or click the folder button to the left of the path bar, a new window will open, then click the (New Folder) button.



Name of new folder inside "CODES":

untitled folder

How to delete files:

There are two ways to delete files (any MATLAB file):

1. Right-click on the file in the current folder, then click delete.
2. Type in the command window (>> delete filename.m)

How to delete folders:

1. Right-click on the folder in the Current Folder window, then click delete.
2. Type in the command window (>> rmdir foldername).

How to save data and variable which are existed in the workspace as (.mat) file:

1. Type in the command window (>> save ‘filename.mat’).
2. Go to the workspace window and click the drop-down menu, then click on (save), a new window will appear, give a file name and then click save button. A new (.mat) file will be created in the current window.

How to open (.mat) files:

1. Double click on the .mat file in the current folder.
2. Type in the command window or any script file (>> load('filename.mat').

How to run the script file:

1. Click the run button at the Editor tab.
2. Type the filename (without the file extension .m) in the command window.
3. Call the file name (without the file extension .m) from another script file.

Some Built-in Math & Logical Functions:

There are plenty of Math functions in MATLAB, herein some of common Math functions:

First create a matrix a;

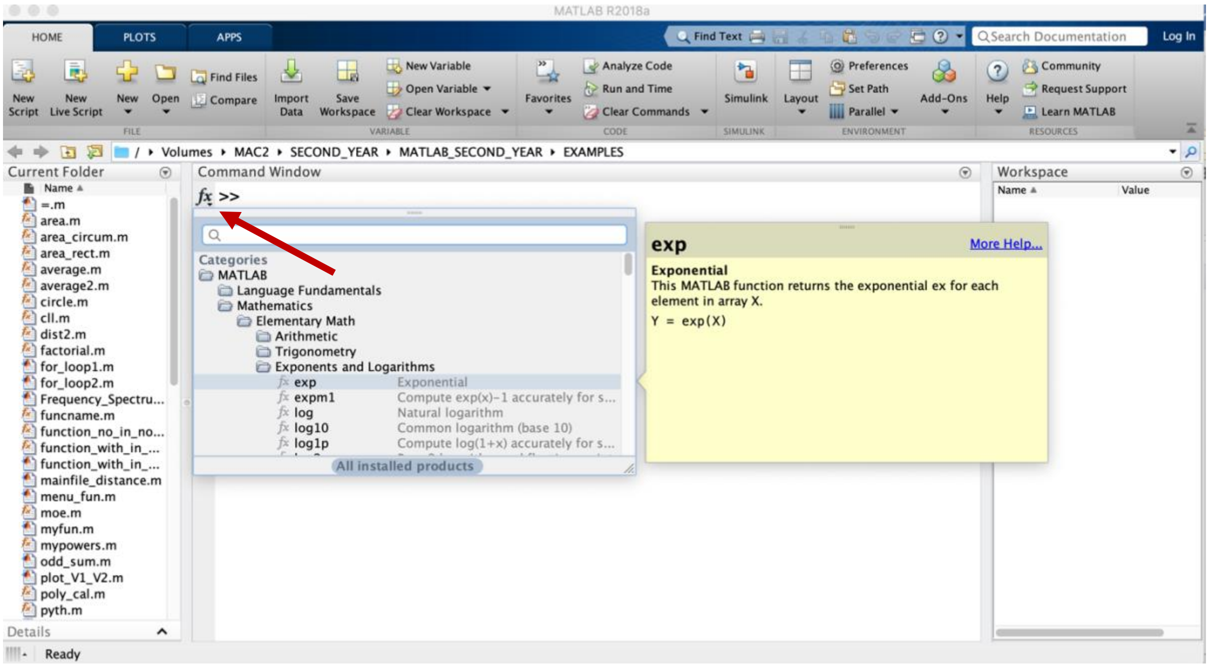
```
>> a=randi([1 9],[3 4])

a =

    8    2    2    4

    6    4    6    6

    4    5    3    3

>> transpose(a)

ans =

    8    6    4

    2    4    5

    2    6    3

    4    6    3

>> a'

ans =

    8    6    4

    2    4    5

    2    6    3
```

```
     4     6     3

>> a=randi([1 9],[3 3])

a =

     8     9     3

     9     6     5

     2     1     9

>> inv(a)                        % inv(a) computes the inverse of matrix a.

ans =

   -0.1914    0.3047   -0.1055

    0.2773   -0.2578    0.0508

    0.0117   -0.0391    0.1289

>> rand                                   % creates one random number only

ans =

    0.2936

>> a=rand(3,4,'double')

a =

    0.9446    0.1375    0.3977    0.3478

    0.5866    0.1393    0.1654    0.7508

    0.9034    0.8074    0.9275    0.7260

>> a=randi(3,4,'double')

a =

     3     1     2     2

     2     3     2     2

     3     2     1     2

     2     3     3     1
```

```
>> a=randi([3 7])           % generates random integer number between 3 & 7
a =
     6
>> randi(5)                 % generates random integer number between 1 & 5
ans =
     5
>> X=(1==2)
X =
  logical
   0
>> class(X)
ans =
    'logical'
>> (1==1)+(3==3)
ans =
     2
>> 1|1
ans =
  logical
   1
>> or(1,1)
ans =
  logical
   1
```

```
>> and(1,0)

ans =

  logical

   0

>> a=NaN(2,3)              % creates a NaN matrix with 2 * 3 dimensions

a =

   NaN   NaN   NaN

   NaN   NaN   NaN
```

For more examples for Math & Logical Built-in functions, please explore the *fx* in the command window as in the following Figure.



exp — Exponential

expm1 — Compute exp(x)−1 accurately for s...

log — Natural logarithm

log10 — Common logarithm (base 10)

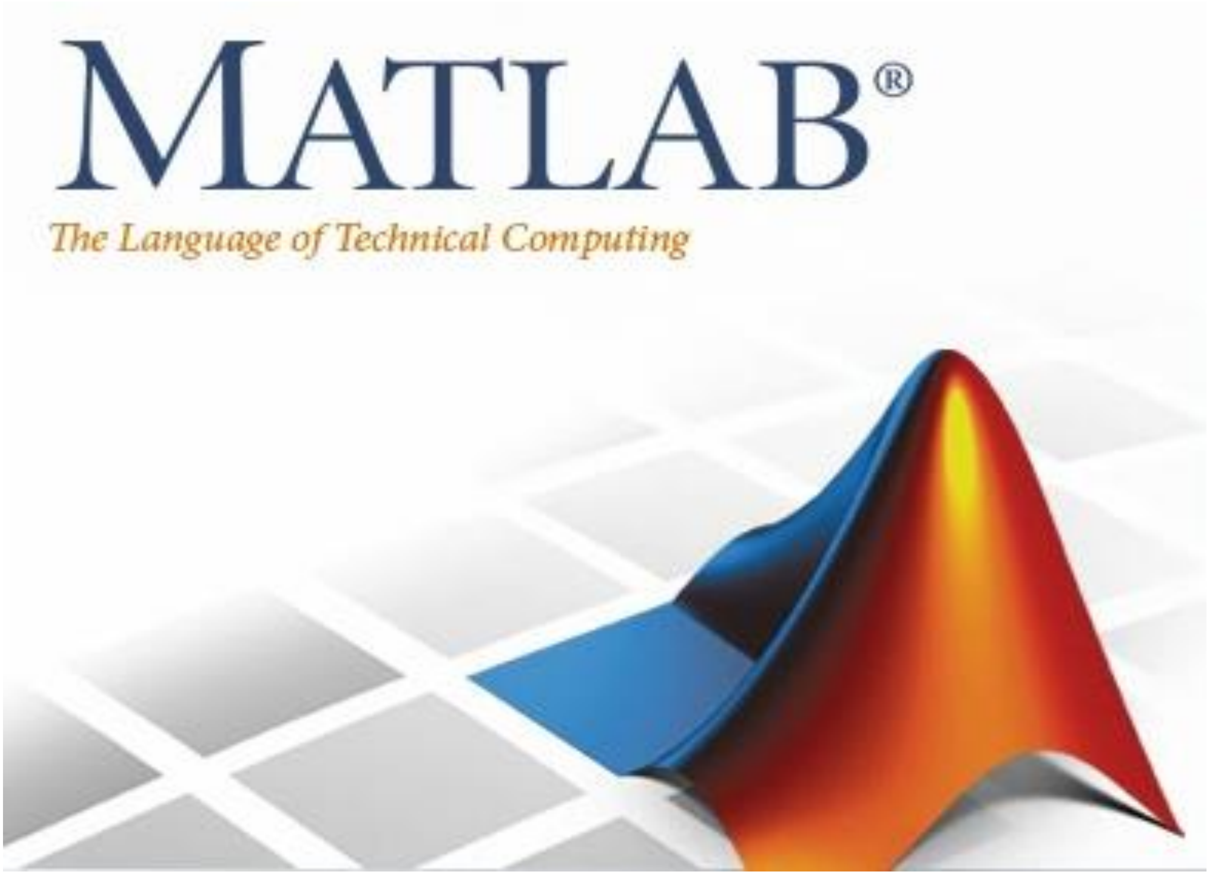
log1p — Compute log(1+x) accurately for s...

exp

Exponential

This MATLAB function returns the exponential ex for each element in array X.

```
Y = exp(X)
```



Lecture 4

User Defined Functions in MATLAB

By: Abidulkarim K. I. Yasari

User Defined Functions in MATLAB

Introduction:

- User-defined functions are similar to the MATLAB pre-defined functions.
- A function is a MATLAB program that can accept inputs and produce outputs.
- A function can be called or executed by another program or function.
- Code for a function is done in an Editor window or any text editor same way as script and saved as m-file.
- The m-file of user-defined function must have the same name as the function name.
- The convention for naming functions is the same as for variables.

- It is important that you give meaningful variable names to variables inside a function that you write, so that you and others can understand what the function does.
- The .m file of the function must be saved in your current directory (otherwise it must be in the path)

Why use user-defined functions?

- Functions provide reusable code.
- Use same code in more than one place in program without rewriting code.
- Reuse code by calling in different programs.
- Make debugging easier.

Before you start …

- Identify the function.
- Decide the function name.
- Decide the input variables.
- Decide the output variables.
- File name must be the same as the function name.

Function Syntax:

```
function[a, b, c]= funcname(x,y)
% where
% function      Declaration Statement.
% [a, b, c]         Output Arguments.
% funcname      Name of the user-defined function.
% (x,y)             Input Arguments.
% Comments about the function
%funcname is a Basic Mathematical function.
%funcname(x,y)is a sample MATLAB function to perform
%basic mathematical operations on input variables x &
%y. Outputs in the following example of the function
%are sum, difference, and product of input arguments.
a = x + y;% executable code to calculate sum.
b = x - y;%executable code to calculate difference.
c = x * y;% executable code to calculate the product.
end
```

- The declaration statement function is used to define the file as a function.
- It must be typed in lower case letters.
- Input arguments
    - Typed inside the parentheses ( )
    - Used to transfer data into function from calling program.
    - Can be zero or more input arguments.
- Output arguments
    - Typed inside the square brackets [ ]
    - Used to transfer data out of function to calling program.
    - Can be zero or more output arguments.
- Give a meaningful variable name.
    - Rules for giving function name is same as the rules for variable names.

More on Function Arguments:

- Functions have private workspaces.
- Variables that are created inside of a user-defined function are referred to as local variables.
- They can only be accessed from inside of that function.
- After the function completes its operations, the local variables are deleted from memory.
- The only variable that appears in the workspace is the output of the function.
- Conversely, functions cannot access variables from the workspace. (with the exception of any input parameters they might have or “global variables”---see MATLAB help on this matter).

Function Definition:

- function [a, b, c] = funcname (x,y)
- The first statement in a function must be function definition.
- Components of a function is discussed previously.
- Basically, a function accepts an input vector, perform the operation, and returns a result.
- The sample function given has two input variables and three output variables.
- But we can also have functions without input or/and output.

Functions with no Input and Output:

- Under some special occasions we may need some functions without any input or output.
- An example is a function used to clear workspace while writing code. It clears command window, workspace from all variables and closes all figures.
- Just a simple code, but useful.

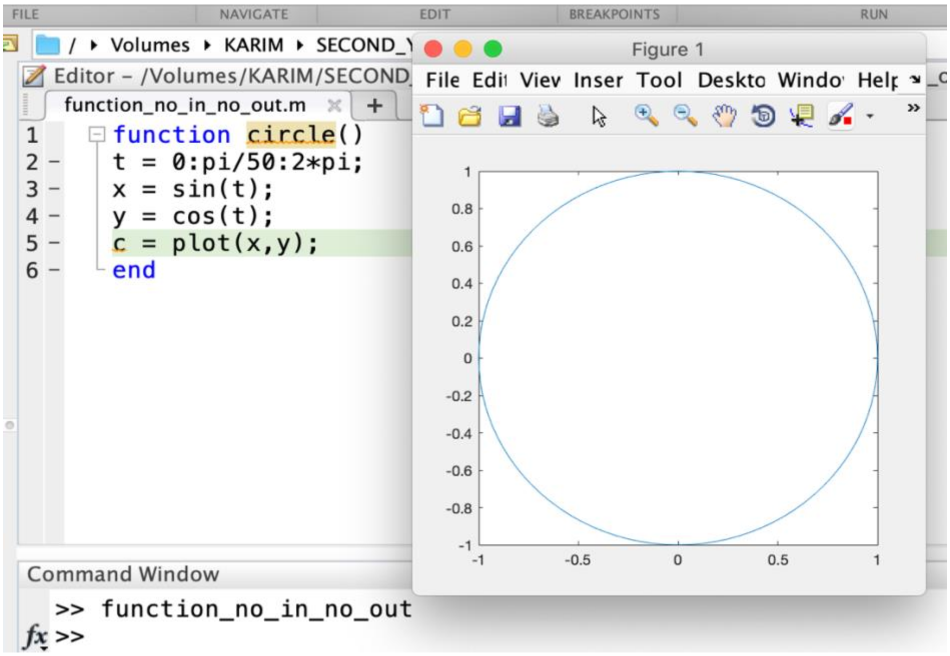
Example 1:

```
function cll()
clc
evalin ('base', 'clear all') % Execute MATLAB expression (clear all) in specified workspace(base in this example).
close all
end
```

Example 2:

- Another example is a function to draw a unit circle.

```
function circle()
t = 0:pi/50:2*pi;
x = sin(t);
y = cos(t);
c = plot(x,y);
end
```



```
function circle()
t = 0:pi/50:2*pi;
x = sin(t);
y = cos(t);
c = plot(x,y);
end
```

```
>> function_no_in_no_out
>>
```

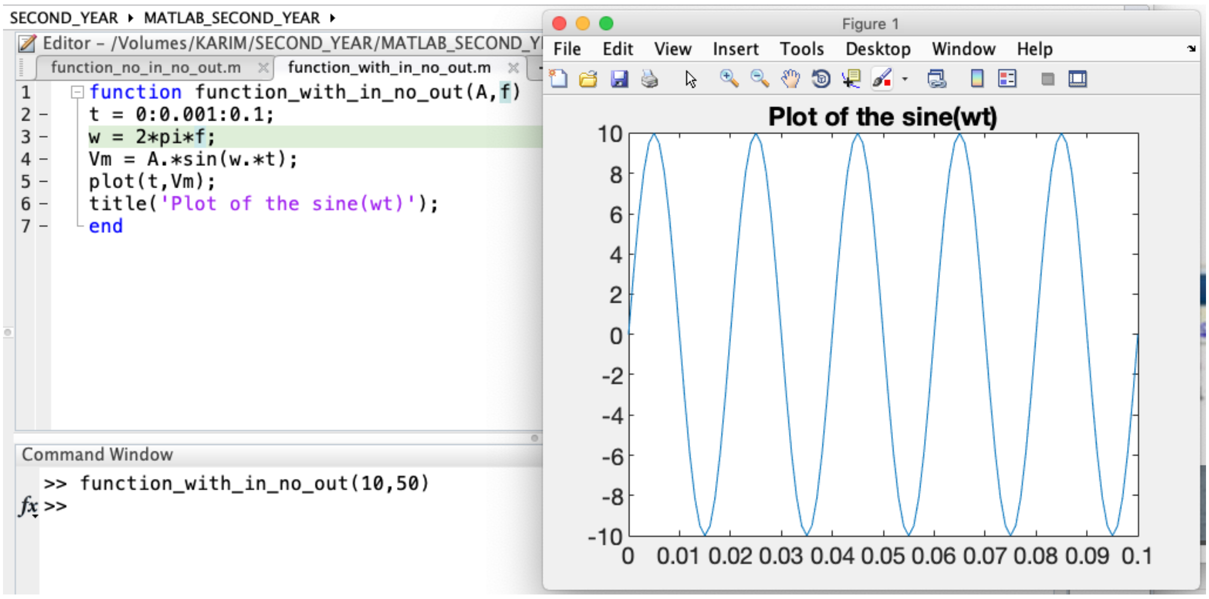
Functions with only Inputs (No Output):

- Some functions will take some input arguments but will not return any output arguments.
- Functions to plot some shapes or curves and functions to display some message are examples.

Example 1:

```
function function_with_in_no_out(A,f)
t = 0:0.001:0.1;
w = 2*pi*f;
Vm = A.*sin(w.*t);
plot(t,Vm);
title('Plot of the sine(wt)');
end
```

You can see after run the function there is no ans variable or any other variables in the workspace.



```
function function_with_in_no_out(A,f)
t = 0:0.001:0.1;
w = 2*pi*f;
Vm = A.*sin(w.*t);
plot(t,Vm);
title('Plot of the sine(wt)');
end
```

```
>> function_with_in_no_out(10,50)
fx >>
```

Example 2:

The built-in MATLAB function tic has no output. It starts a timer going for use with another built-in MATLAB function, toc.

Measure time to generate two random matrices and compute element-by-element multiplication of their transposes.

```
tic

A = rand(12000, 4400);

B = rand(12000, 4400);

toc

C = A'.*B';

toc
```

Functions with only Outputs (No Intput):

- If you need to access some constant value a number of times, instead of hard-coding the constant value every time it is needed, we can hard-code it once into a function that has no input.
- When the constant is needed, its value is called via the function.
- As an example, if we needed, for whatever reason, the mass of the earth (in kg) multiple times, we could write a function to store it as follows:

```
function mass_of_earth = moe()
mass_of_earth = 5.976E24;
end
```

or

```
function today = show_date()
today=date;
end
```

Function with one Output:

The function has only one output you don’t need to put it inside the square brackets [ ].

Example 1:

The function in a file named average.m that accepts an input vector, calculates the average of the vector values, and returns a single result.

```
function y = average(x)
     if isscalar(x)
          error('Input must be a vector')
     end
     y = sum(x)/length(x);
end
```

Example 2:

The function factorial(n) will find the factorial of input n, i.e. the product of all the integers from 1 to n.

```
function f = factorial(n)
if (length(n)~=1) | (fix(n) ~= n) | (n < 0) % fix (round toward zero)
    error('n must be a positive integer');
end
f = prod(1:n); % (prod) Product of array elements
end
```

Functions with more Outputs:

- When there are more than one output arguments put them in a square bracket.

Example:

```
function [a, b, c] = basicmath (x, y)
% basicmath is the name of user-defined
%function(Basic Mathematical function)
% to perform basic mathematical operations of input
%variables x and y.
% outputs of the function are sum, difference and
product of the input arguments.
```

```
a = x+y;
b = x-y;
c = x*y;
end
```

Calling a user-defined function:

- A function can be called from the command window or inside a script or function.
- To run a function in command window, just type the name of the function with proper input and output arguments
- For example, consider the function basicmath

```
>> [a,b,c] = basicmath (2,3)

a = 5

b = -1

c = 6
```

Example 2:

```
% The function to be called in a script can be
% included in the same script.
r = input('Enter radius of the sphere: ');
vol = volume_sphere(r); % Compute the volume.
fprintf('The volume of sphere is %f\n',vol);
function y = volume_sphere(r)
y = (4/3)*pi.*(r^3);
end
```

Example 3:

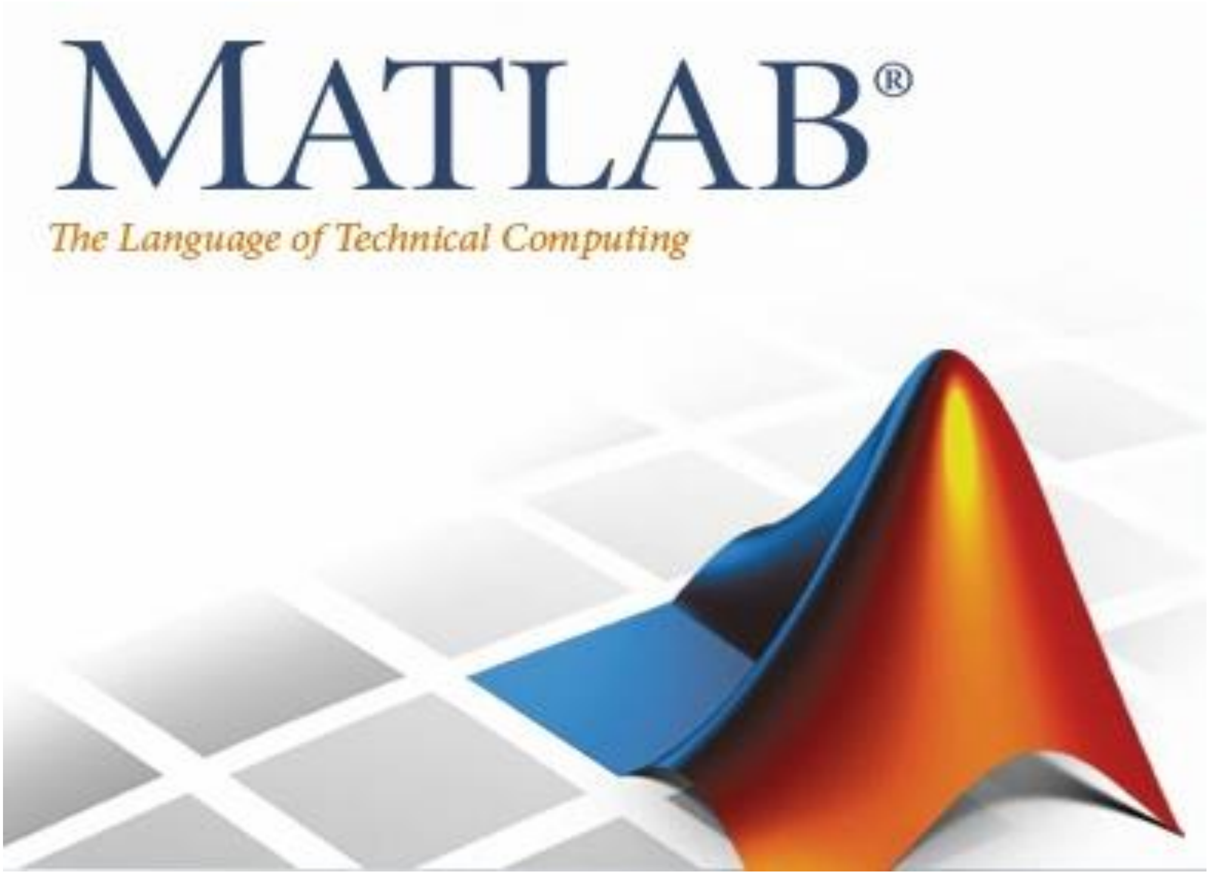
Consider the function to calculate distance between two points. First create a user defined function to calculate distance between two points and give it the file name dist2.m

```
function distance = dist2(x1, y1, x2, y2)
distance = sqrt((x2-x1).^2 + (y2-y1).^2);
 end
```

This function can be called in another script. To do so, create a script file as in following and give it the name calling_dist2.m as in following:

```
 % Get input data.
```

```
ax = input('Enter x value of point a:     ');
ay = input('Enter y value of point a:     ');
bx = input('Enter x value of point b:     ');
by = input('Enter y value of point b:     ');

% Evaluate function

result = dist2 (ax, ay, bx, by);

% Write out result.

fprintf('The distance between points a and b is …
%f\n',result);
```



Lecture 5

Branch and Loop Constructs in MATLAB

By: Abidulkarim K. I. Yasari

Branches and Loops:

- Branches (decision making) and loops are two of the most important constructs in computer programming.
- Branches allow code to optionally execute different code (or none at all) depending on the values of variables and other criteria.
- Loops make code repeat execution, possibly millions of times per second.

MATLAB – Loop types

There may be a situation when you need to execute a block of code several times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. MATLAB provides various types of loops to handle looping requirements including while loops, for loops, and nested loops. If you are trying to declare or write your own loops, you need to make sure that the loops are written as scripts and not directly in the Command Window.
To start a new script, locate the button in the upper left corner of the window labelled ***New Script***.

for Loop:

- **for executes a group of statements in a loop for a specified number of times. *Values* has one of the following forms:**
- ***initVal*:*endVal* — Increment the *index* variable from *initVal* to *endVal* by 1, and repeat execution of *statements* until *index* is greater than *endVal*.**
- ***initVal*:*step*:*endVal* — Increment *index* by the value *step* on each iteration, or decrements *index* when *step* is negative.**

Syntax of the for loop is shown below:

```
for index = array of values
    statements
end
```

The commands between the `for` and `end` statements are executed for all values stored in the array.
Suppose that one-need values of the sine function at eleven evenly spaced points $\pi n/10$, for $n = 0, 1, \ldots, 10$. To generate the numbers in question one can use the for loop.

```
for n=0:10
    x(n+1) = sin(pi*n/10);
end
x
```

save the script file as for_loop1, and run the program:

```
>> for_loop1

x =

       0   0.3090   0.5878   0.8090   0.9511   1.0000   0.9511   0.8090   0.5878   0.3090   0.0000
```

The for loops can be nested:

Long loops are more memory efficient when the colon (:) expression appears in the for statement since the index vector is never created.

```
H = zeros(5);
for k=1:5
    for l=1:5
        H(k,l) = 1/(k+l-1);
    end
end
H
```

Save the script file as nested_for_loop, and run it:

```
>> nested_for_loop

H =
   1.0000   0.5000   0.3333   0.2500   0.2000
   0.5000   0.3333   0.2500   0.2000   0.1667
   0.3333   0.2500   0.2000   0.1667   0.1429
   0.2500   0.2000   0.1667   0.1429   0.1250
   0.2000   0.1667   0.1429   0.1250   0.1111
```

Matrix H created here is called the Hilbert matrix. First command assigns a space in computer's memory for the matrix to be generated. This is added here to reduce the overhead that is required by loops in MATLAB.
The for loop should be used only when other methods cannot be applied. Consider the following problem. Generate a 10-by-10 matrix $A = [a_{kl}]$, where $a_{kl} = \sin(k)\cos(l)$. Using nested loops one can compute entries of the matrix A using the following code:

```
>> nested_for_loop2

A =

   0.4546  -0.3502  -0.8330  -0.5500   0.2387   0.8080   0.6344  -0.1224  -0.7667  -0.7061

   0.4913  -0.3784  -0.9002  -0.5944   0.2579   0.8731   0.6855  -0.1323  -0.8285  -0.7630

   0.0762  -0.0587  -0.1397  -0.0922   0.0400   0.1355   0.1064  -0.0205  -0.1286  -0.1184

  -0.4089   0.3149   0.7492   0.4947  -0.2147  -0.7267  -0.5706   0.1101   0.6895   0.6350

  -0.5181   0.3991   0.9493   0.6268  -0.2720  -0.9207  -0.7229   0.1395   0.8737   0.8046

  -0.1510   0.1163   0.2766   0.1826  -0.0793  -0.2683  -0.2107   0.0407   0.2546   0.2344

   0.3550  -0.2734  -0.6504  -0.4294   0.1864   0.6308   0.4953  -0.0956  -0.5986  -0.5513

   0.5346  -0.4117  -0.9795  -0.6467   0.2806   0.9500   0.7459  -0.1440  -0.9014  -0.8301

   0.2227  -0.1715  -0.4080  -0.2694   0.1169   0.3957   0.3107  -0.0600  -0.3755  -0.3458

  -0.2939   0.2264   0.5386   0.3556  -0.1543  -0.5224  -0.4101   0.0792   0.4957   0.4565
```

A loop free version might look like this:

```
k = 1:10;
A = sin(k)'*cos(k);
A
```

```
>> loop_free_version

A =

   0.4546  -0.3502  -0.8330  -0.5500   0.2387   0.8080   0.6344  -0.1224  -0.7667  -0.7061

   0.4913  -0.3784  -0.9002  -0.5944   0.2579   0.8731   0.6855  -0.1323  -0.8285  -0.7630

   0.0762  -0.0587  -0.1397  -0.0922   0.0400   0.1355   0.1064  -0.0205  -0.1286  -0.1184

  -0.4089   0.3149   0.7492   0.4947  -0.2147  -0.7267  -0.5706   0.1101   0.6895   0.6350

  -0.5181   0.3991   0.9493   0.6268  -0.2720  -0.9207  -0.7229   0.1395   0.8737   0.8046

  -0.1510   0.1163   0.2766   0.1826  -0.0793  -0.2683  -0.2107   0.0407   0.2546   0.2344

   0.3550  -0.2734  -0.6504  -0.4294   0.1864   0.6308   0.4953  -0.0956  -0.5986  -0.5513

   0.5346  -0.4117  -0.9795  -0.6467   0.2806   0.9500   0.7459  -0.1440  -0.9014  -0.8301
   0.2227  -0.1715  -0.4080  -0.2694   0.1169   0.3957   0.3107  -0.0600  -0.3755  -0.3458
  -0.2939   0.2264   0.5386   0.3556  -0.1543  -0.5224  -0.4101   0.0792   0.4957   0.4565
```

First command generates a row array k consisting of integers 1, 2, … , 10. The command sin(k)' creates a column vector while cos(k) is the row vector. Components of both vectors are the values of the two trig functions evaluated at k. Code presented above illustrates a powerful feature of MATLAB called vectorization. This technique should be used whenever it is possible.

Useful Tips:

- Avoid assigning a value to the *index* variable within the loop statements. The for statement overrides any changes made to *index* within the loop.
- To iterate over the values of a single column vector, first transpose it to create a row vector.

While Loop:

The while loop repeatedly executes statements while a specified condition is true. The syntax of a while loop in MATLAB is as following:

```
while <expression>   condition
    <statements>
end
```

The while loop repeatedly executes a program statement(s) as long as the expression remains true. An expression is true when the result is nonempty and contains all nonzero elements (logical or real numeric). Otherwise, the expression is false.

Example:

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

Save the script file as while_loop1.m. When the code above is executed, the result will be:

```
>> while_loop1
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Example:

```
%Repeat Statements Until Expression Is False
%Use a while loop to calculate factorial(10).
n = 10;
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
```

The output will be:

```
>> while_loop2
n! = 3628800
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. The scope defines where the variables can be valid in MATLAB, typically a scope within a loop body is from the beginning of conditional code to the end of conditional code. It tells MATLAB what to do when the conditional code fails in the loop. MATLAB supports both break statement and continue statement.

Break statement:

The break statement terminates execution of for or while loops. Statements in the loop that appear after the break statement are not executed. In nested loops, break exits only from the loop in which it occurs. Control passes to the statement following the end of that loop.

Example:

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

The output is:

```
>> break_example
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Continue Statement:

The continue statement is used for passing control to the next iteration of a for or while loop. The continue statement in MATLAB works somewhat like the break statement. Instead of forcing termination, however, 'continue' forces the next iteration of the loop to take place, skipping any code in between.

Example:

```
a = 10;
%while loop execution

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

The output is:

```
>> continue_example

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

Switch – case commands:

In this section, we will discuss more loop controls, which includes switch-case statement.

- There is no conditional statement.
- It chooses code to execute based on value of scalar, string or a cell array of scalars or strings, not just true/false.
- Different codes can be executed based on the value of the scalar.
- It is easier than if-elseif-end commands if the conditions are more.

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

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

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

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

The syntax of switch statement in MATLAB is:

```
switch variable
  case exp1
    action(s)
  case exp2
    action(s)
  case exp3
    action(s)
  otherwise
    action(s)
end
```

Example:

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

The output is:

```
>> switch_case_example
Excellent!
```

- The expression after switch command is evaluated first.
- If the obtained value is equal to ‘A’ – then commands below case ‘A’ is executed up to next case or otherwise or end statement.
- Similarly if the obtained value is equal to ‘B’ – then then commands below case ‘B’ is executed.
- If switch-expression not equal to any of the values (A, B, C, D, or F) in case statement, commands after otherwise executed.
- If otherwise command is not present, no commands are executed

Example 2:

```
id = input(‘enter your id:   ‘)
switch id
    case 321456
        disp(‘You name is Martin’)
    case 320423
        disp(‘You name is Allen’)
    case 332458
        disp(‘You name is Mohan’)
    case 321564
        disp(‘You name is Vicky’)
    otherwise
        disp(‘You are not a member of the class’)
end
```

The output is:

```
>> switch_case_example2
enter your id:  320423
id =
    320423
You name is Allen
```

The if Statement:

MATLAB provides an “if” statement

- Nearly all programming languages have something similar.
- Can be entered in the command window.
- More commonly used in scripts/functions.
- The “if” keyword (“if” is a reserved word in MATLAB).

A logical or relational condition Warning: Only works for scalars.

Actions to be performed if condition(s) is (are) TRUE. Note: The actions should be indented (easier to read).

The “end” keyword that ends the “if” statement.

The if statement structure is:

```
if condition
    action(s)
end
```

Example:

```
x = 20;
if x > 10
    disp('x is greater than ten!\n')
end
```

The output is:

```
>> if_example
x is greater than ten!\n
```

Example 2:

```
x=10;
if rem(x,2)==0
    fprintf('%d is divisible by 2\n',x);
end
if rem(x,3)==0
    fprintf('%d is divisible by 3\n',x);
end
if rem(x,4)==0
    fprintf('%d is divisible by 4\n',x);
end
if rem(x,5)==0
    fprintf('%d is divisible by 5\n',x);
end
if(rem(x,2)~=0)&&(rem(x,3)~=0)&&(rem(x,4)~=0)&&(rem(x,5)~=0)
    fprintf('%d not divisible by 2,3,4,5\n',x);
end
```

The output is:

```
>> if_example2
10 is divisible by 2
10 is divisible by 5
```

The If-Else Statement:

What if more than one condition needs to be tested?

- Use a nested if-else statement.
- Can execute different actions depending on if condition is met.
- More efficient than separate if statements.
- Actions only executed if condition is TRUE
- Actions only executed if condition is FALSE

The syntax of If-Else statement is:

```
if condition
    action(s)
else
    action(s)
end
```

Example:

```
x = 2;
if x > 10
    disp('x is greater than ten!\n')
else
    disp('x is less than ten!\n')
end
```

The output is:

```
>> if_example
x is less than ten!\n
```

The If-Elseif-Else Statement:

What if multiple conditions need to be tested?

- Each one results in different actions

  ⇒ Use a nested if-elseif-else statement

- Much more efficient than separate if statements
- Can have as many elseif statements as needed

The syntax of if-elsef-else statement is:

```
if condition1
    action(s)
elseif condition2
    action(s)
else
    action(s)
end
```

`action(s)` **Only executed if condition1 is true**

`action(s)` **Only executed if condition1 is FALSE and condition2 is TRUE**

`action(s)` **Only executed if condition1 and condition2 are BOTH FALSE**

Example:

```
x = 7;
if x > 10
    disp('x is greater than 10!\n')
elseif x >= 5 && x <= 10
    disp('x is between 5 and 10!\n')
```

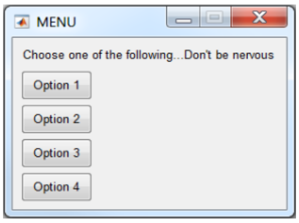
```
elseif x >= 0 && x < 5
    disp('x is between 0 and 5!\n')
else
    disp('x is a negative number!\n')
end
```

The output is:

```
>> if_elseif_else
x is between 5 and 10!
```

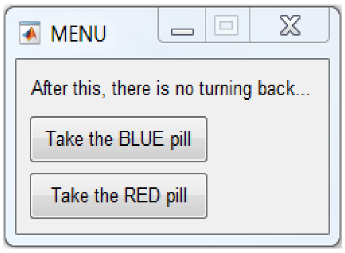
The menu function:

- If you are writing code that may be used by novice users, often having graphical buttons is useful
  - ⇒ Not good for expert users
  - ⇒ Cannot easily automate mouse clicks
  - ⇒ Clicking buttons is very slow



Choose one of the following...Don't be nervous

- It is good to understand that buttons in software are executing source code
- MATLAB provides the “menu” function
  - ⇒ Makes simple graphical buttons
  - ⇒ Pops up in a new window
  - ⇒ Pairs up well with switch or nested if statements

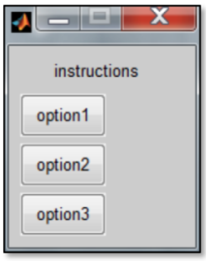


After this, there is no turning back...

The Menu Function

- The menu function is straightforward and easy to use

```
choice = menu('instructions','option1','option2','option3')
```



🟥 Where the returned value (i.e. the choice) is stored

🟦 Text/Instructions to put in the menu box above the buttons

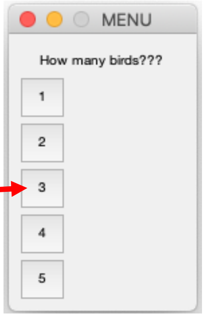
🟨 Text to put on top button
If clicked, returns 1

🟩 Text to put on second from top button
If clicked, returns 2

🟪 Text to put on third from top button
If clicked, returns 3

Example:

```
clc,clear
x = menu('How many birds???','1','2','3','4','5');
switch x
    case 1
        disp('You need 3 more birds')
    case 2
       disp('You have a perfect pair of birds')
    case 3
       disp('You have a trio of birds')
    case 4
        disp('You have two-squared birds')
    case 5
        disp('Seriously???')
    otherwise
        disp('You have more than enough of birds')
end
```



Click Here

The output is:

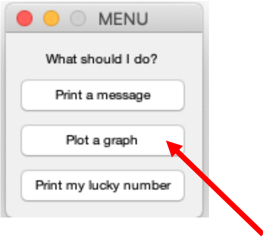
You have a trio of birds

Example 2:

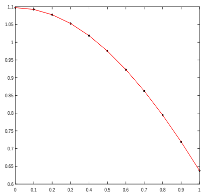
```
% The first script using the menu function
% menu_example.m
clc,clear
pick = menu('What should I do?','Print a message','Plot a graph','Print my
lucky number');

switch pick
case 1
    fprintf('You rock at menus\n');
case 2
    x = 0:0.1:1;
    y = cos(x) + rand;
 plot(x,y,'ro','MarkerFaceColor','r','MarkerEdgeColor’,'k','MarkerSize',3);
    fprintf('Here is your graph\n');
case 3
    num = round(rand*100);
    fprintf('Your lucky number is: %d\n',num);
otherwise
    fprintf('You didn''t select anything!\n');
    fprintf('That makes me sad\n');
    fprintf(':(\n');
end
```

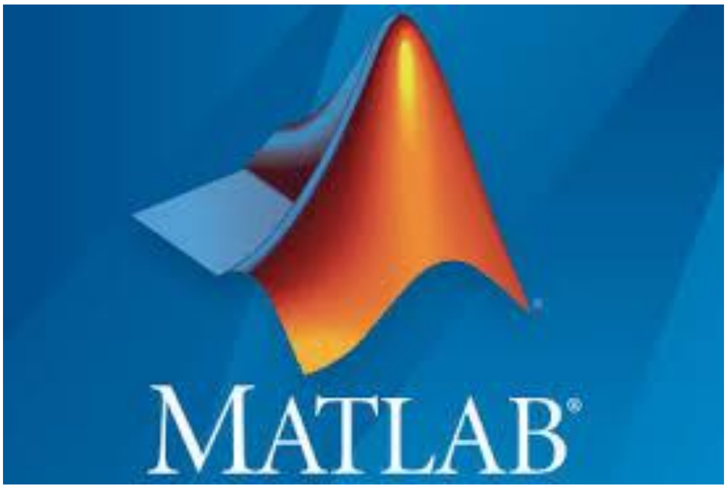
The output is:

Here is your graph



Click Here





Lecture 6

Plotting & Graphs

By: Abidulkarim K. I. Yasari

MATLAB has an excellent set of graphic tools. Plotting a given data set or the results of computation is possible with very few commands. You are highly encouraged to plot mathematical functions and results of analysis as often as possible. Trying to understand mathematical equations with graphics is an enjoyable and very efficient way of learning mathematics. Being able to plot mathematical functions and data freely is the most important step, and this section is written to assist you to do just that.

The basic MATLAB graphing procedure, for example in 2D, is to take a vector of x-coordinates,
x = (x1,...,xN), and a vector of y-coordinates,
y = (y1,...,yN), locate the points (xi, yi), with
i = 1, 2, . . . , n and then join them by straight lines. You need to prepare x and y in an identical array form; namely, x and y are both row arrays or column arrays of the same length.

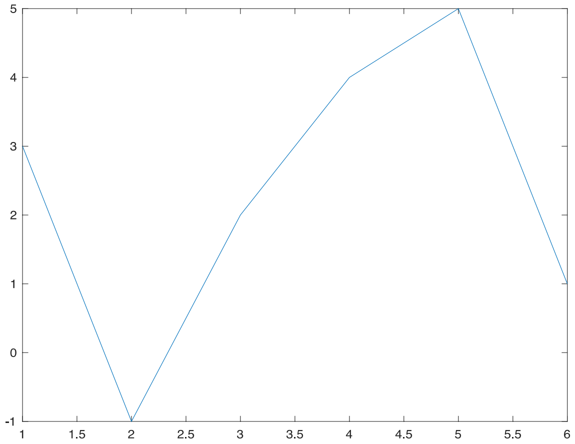
The MATLAB command to plot a graph is
plot(x,y).

The vectors
x = [1, 2, 3, 4, 5, 6] and
y = [3, −1, 2, 4, 5, 1]
produce the picture shown in the following Figure .

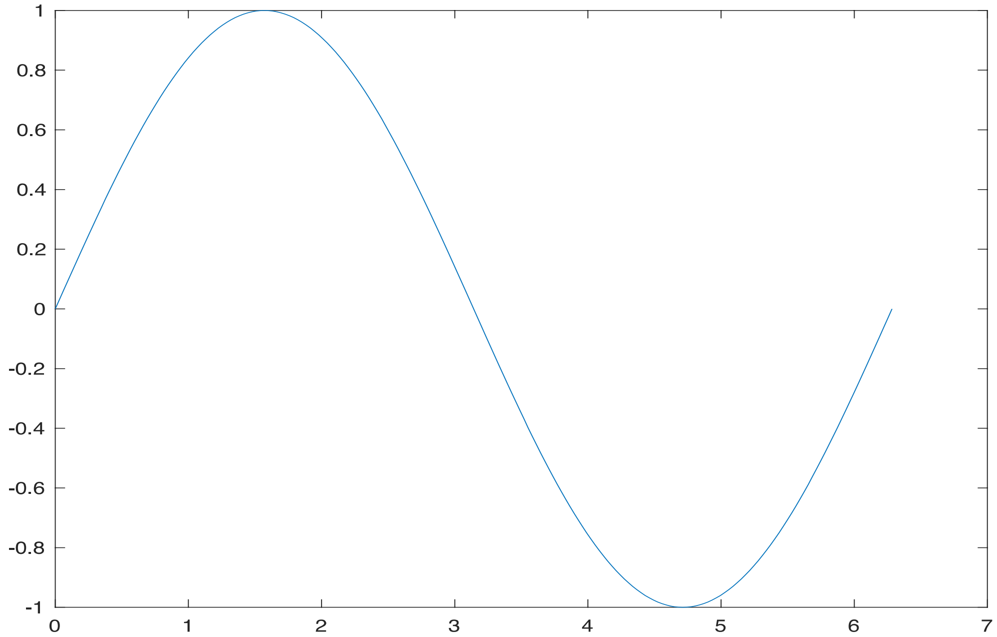
```
>> x = [1 2 3 4 5 6];
>> y = [3 -1 2 4 5 1];
>> plot(x,y)
```

•Note: The plot function has different forms depending on the input arguments. If y is a vector plot(y)produces a piecewise linear graph of the elements of y versus the index of the elements of y. If we specify two vectors, as mentioned above, plot(x,y) produces a graph of y versus x.

•For example, to plot the function sin(x) on the interval [0,2π], we first create a vector of x values ranging from 0 to 2π, then compute the sine of these values, and finally plot the result:



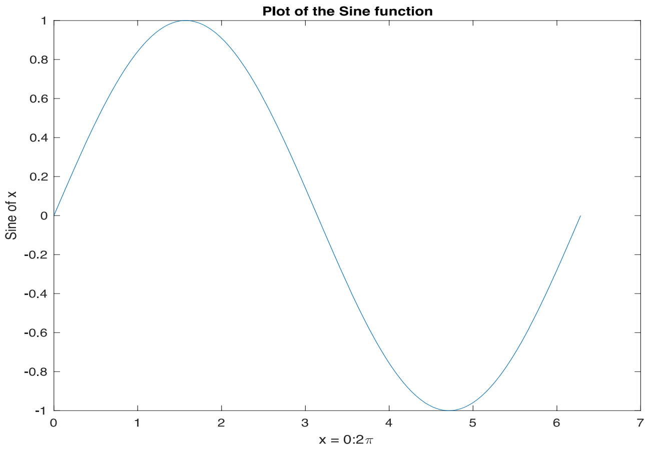
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)



MATLAB enables you to add axis labels and titles. For example, using the graph from the previous example, add an x- and y-axis labels.
Now label the axes and add a title. The character \pi creates the symbol π. An example of 2D plot is shown in the following Figure.
The color of a single curve is by default blue, but other colors are possible. The desired color is indicated by a third argument. For example, red is selected by plot(x,y,'r'). Note the single quotes, ' ', around r.

x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)

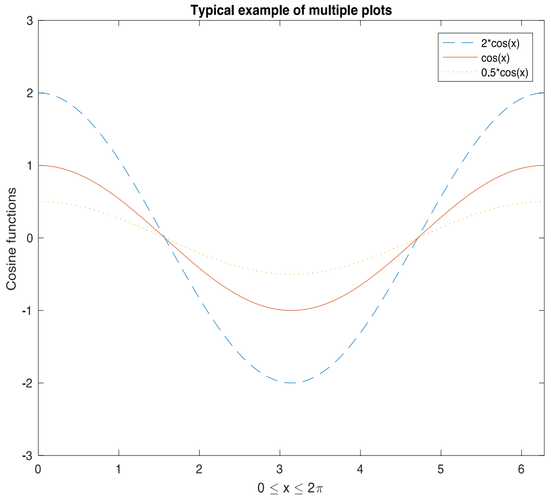
```
xlabel(’x = 0:2\pi’)
ylabel(’Sine of x’)
title(’Plot of the Sine function’)
```



Multiple data sets in one plot:

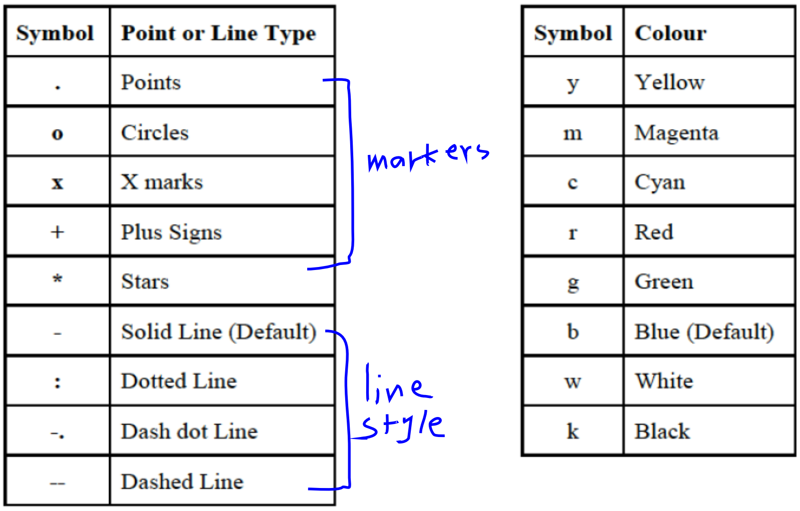
Multiple (x, y) pairs arguments create multiple graphs with a single call to plot. For example, these statements plot three related functions of x: y1 = 2cos(x), y2 = cos(x), and y3 = 0.5 ∗ cos(x), in the interval 0 ≤ x ≤ 2π.

```
x = 0:pi/100:2*pi;
 y1 = 2*cos(x);
 y2 = cos(x);
 y3 = 0.5*cos(x);
 plot(x,y1,'--',x,y2,'-',x,y3,':')
 xlabel('0 \leq x \leq 2\pi')
 ylabel('Cosine functions')
 legend('2*cos(x)','cos(x)','0.5*cos(x)')
```



Specifying line styles and colors:

It is possible to specify line styles, colors, and markers (e.g., circles, plus signs, . . .) using the plot command:
plot(x,y,’style_color_marker’) where style_color_marker is a triplet of values from the following Table.


markers
line style

Symbol	Point or Line Type
.	Points
o	Circles
x	X marks
+	Plus Signs
*	Stars
-	Solid Line (Default)
:	Dotted Line
-.	Dash dot Line
--	Dashed Line

Symbol	Colour
y	Yellow
m	Magenta
c	Cyan
r	Red
g	Green
b	Blue (Default)
w	White
k	Black

2D Graph Commands:

The plot command produces a linear x-y plot. Listed below is a selection of plot commands.
plot(y) %Plot y against the element number.
plot(x,y) %Plot y against x.
plot(x1,y1,x2,y2) % Plot y1 against x1 and y2 against x2.
You can use a third argument to define various options.
plot(x,y,'r+’) %Plot y against x using red plus signs.
plot(x1,y1,'r+',x2,y2, 'go') %Use red plus signs for x1 and y1and %green circles for x2 y2.
The plot options can be one, two or three characters long. For example
‘-rx' will plot x marks and a solid line, both in red.

In addition to the plot command, there are other commands to produce other types of 2D graphs.

On the next page are various examples of the graphs produced by the commands fill(we have to specify the colour as a third argument), bar, stairs, loglog, semilogx, semilogy, polar, compass, feather etc.

The arguments and options of these commands are similar to those of plot.

Reading Data from files

•MATLAB supports reading an entire file and creating a matrix of the data with one statement.
•
>> load mydata.dat; % *load*s file into matrix.
% The matrix may be a scalar, a vector, or a
% matrix with multiple rows and columns. The
% matrix will be named mydata.
>> size (mydata) % *size* will return the number

```
  % of rows and number of
  % columns in the matrix
>> length (myvector)  % length will return the total
  % no. of elements in myvector
```

Examples:

```
>> load('filename.mat')        % loading data from .mat files
>>xlsread('filename.xlsx')  % loading data from .xlsx files
```

•MATLAB will plot one vector vs. another. The first one will be treated as the abscissa (or x) vector and the second as the ordinate (or y) vector. The vectors have to be the same length.

•

•MATLAB will also plot a vector vs. its own index. The index will be treated as the abscissa vector. Given a vector “time” and a vector “dist” we could say:

```
 >> plot (time, dist)     % plotting versus time
 >> plot (dist)  % plotting versus index
```

•There are commands in MATLAB to "annotate" a plot to put on axis labels, titles, and legends. For example:

```
 >> % To put a label on the axes we would use:
 >> xlabel ('X-axis label')
 >> ylabel ('Y-axis label')
 >> % To put a title on the plot, we would use:
 >> title ('Title of my plot')
```

•Vectors may be extracted from matrices. Normally, we wish to plot one column vs. another. If we have a matrix “mydata” with two columns, we can obtain the columns as vectors with the assignments as follows:

```
>> first_vector = mydata ( : , 1) ;        % First column
>> second_vector = mydata ( : , 2) ;   % Second one
```

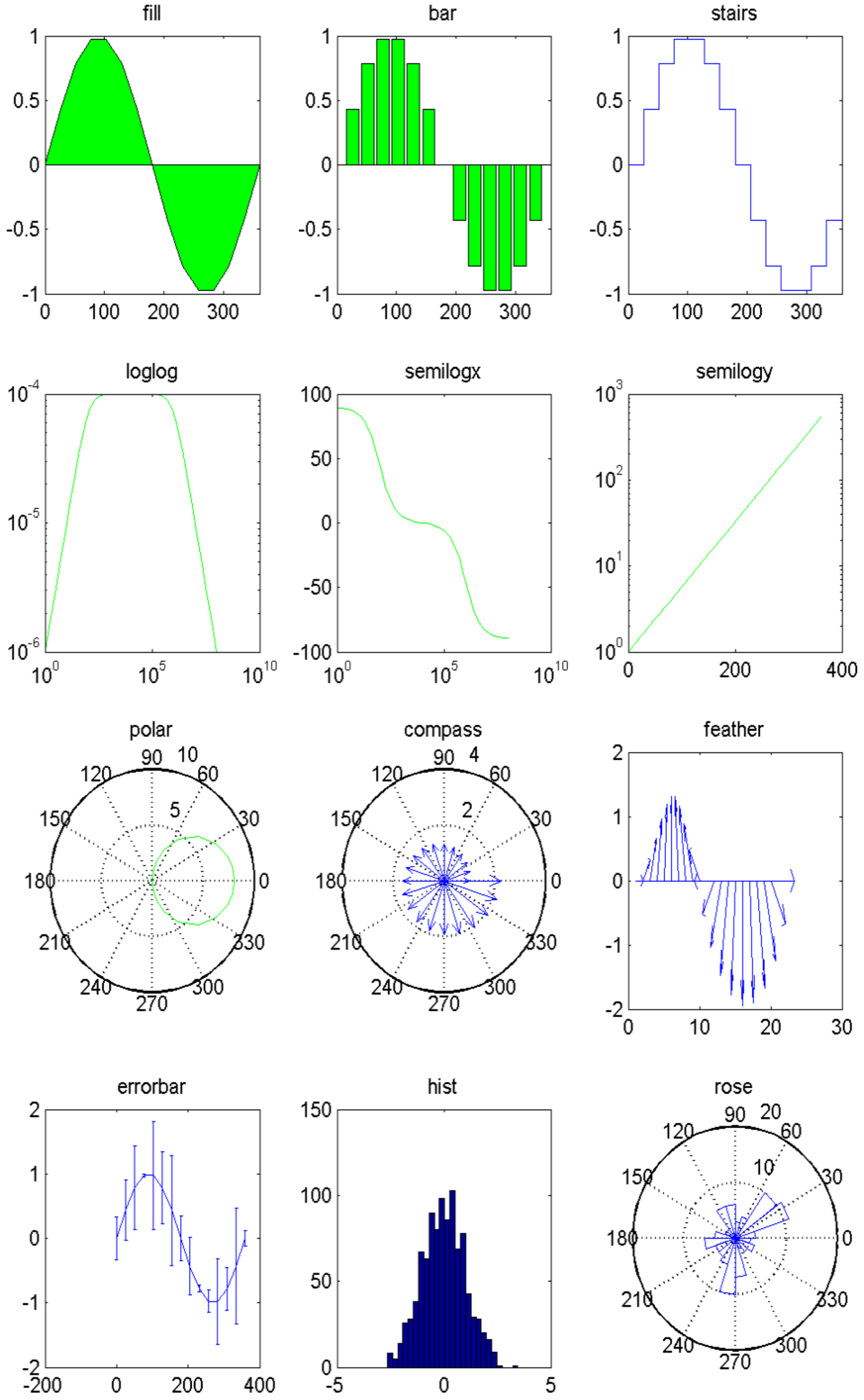
and we can plot the data

```
>> plot ( first_vector , second_vector )
```

Graph Commands

This section describes how you can produce graphs in MATLAB. The script below shows how to plot and annotate a simple graph.

```
% Matlab Script to plot a Sin Wave
% Create the x and y vectors to plot
x = 0:6:360 ;                 % The values of x in degrees
rads = 2*pi*x./360;           % The above convert to radians
y = sin(rads);                         % Calculate the values of y
plot(x,y);                             % Plot the graph
% Annotate the graph
title('Sine Wave’);           % Add a Title
xlabel('degrees’);            % Label the x axis
ylabel('value’);              % Label the y axis
grid;                         % Draw a grid on the graph
axis([0 360 -1 1]);           % Set the limits of the x and y axis
        x    y
```


fill
1
0.5
0
-0.5
-1
0
100
200
300
bar
1
0.5
0
-0.5
-1
0
100
200
300
stairs
1
0.5
0
-0.5
-1
0
100
200
300
loglog
10^{-4}
10^{-5}
10^{-6}
10^{0}
10^{5}
10^{10}
semilogx
100
50
0
-50
-100
10^{0}
10^{5}
10^{10}
semilogy
10^{3}
10^{2}
10^{1}
10^{0}
0
200
400
polar
90
10
120
60
5
150
30
180
0
210
330
240
300
270
compass
90
4
120
60
2
150
30
180
0
210
330
240
300
270
feather
2
1
0
-1
-2
0
10
20
30
errorbar
2
1
0
-1
-2
-200
0
200
400
hist
150
100
50
0
-5
0
5
rose
90
20
120
60
10
150
30
180
0
210
330
240
300
270

Subplots:

Subplot enables you to split the graphics window into a number of smaller subgraphs. As an example, consider how the graphs where plotted on the previous page.

```
subplot(4,3,1);                          % 4 by 3 graphs, 1st graph
fill(degrees,sin(rad),'g’);      % Plot a Fill graph
```

The subplot command has three arguments. The first digit specifies the number of rows of graphs you want and the second the number of columns. In this case we are going to have 4 rows of graphs in 3 columns. The third number specifies which of the 12 subgraphs you are going to use next. The subgraphs are numbered from left to right, top to bottom. So the Fill plot will be in the top left had corner. Next to it is the bar plot.

```
subplot(4,3,2);                          % 4 by 3 graphs, 2nd graph
bar(degrees,sin(rad),'g’);       % Plot a Bar Graph
```

and so on.

```
subplot(4,3,3);                          % 4 by 3 graphs, 3rd graph
stairs(degrees,sin(rad));        % Plot a Stair Graph
subplot(4,3,4);                          % 4 by 3 graphs, 4th graph
```

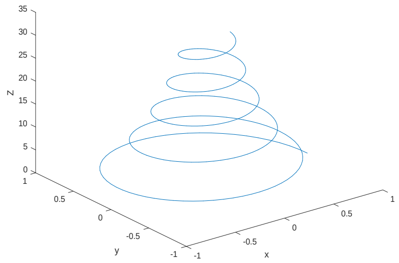
etc

PLOTTING IN THREE DIMENSIONS:

Elementary 3-D Plotting:

we discussed how matrix data could be visualized by plotting with the plot command. Not all data is intuitively represented with a 2-D plot. We live in a three-dimensional world and much of our information is best revealed with 3-D techniques. Fortunately, MATLAB provides you with a cornucopia of graphics functions that let you make quick 3-D plots and visualizations of your data. The plot3 function is used in almost the same way that plot is used, except that an additional variable, z, is used to provide the data for the third dimension. For example, let’s make use of the form plot3 (x,y,z) by typing:

```
t = 0:0.1:10*pi;
x = exp(-t/20).*cos(t);
y = exp(-t/20).*sin(t);
z = t;
plot3(x,y,z);
xlabel('x');
ylabel('y');
zlabel('z’);
%  here we introduce a new labeling command, zlabel
```



You can change the viewing angle of plot by either one of two ways. First, you can select the Rotate 3-D tool from the Figure Window. Doing so will let you to interactively rotate the axes

of the plot by holding down the mouse button and moving the mouse about. The specific values of the azimuth and elevation will be shown in the lower left corner of the figure while you are rotating the axes.
Your second option is to use the view function. The general form of this function is view (az, el) or view ([az,el]) and with it you can specify the exact values of azimuth and elevation by which you wish to rotate the axes, the different views shown in the following figure:

plot3 Plot lines and points in 3-D space.
plot3() is a three-dimensional analogue of PLOT().

plot3(x,y,z), where x, y and z are three vectors of the same length,
plots a line in 3-space through the points whose coordinates are the elements of x, y and z.

plot3(X,Y,Z), where X, Y and Z are three matrices of the same size,
plots several lines obtained from the columns of X, Y and Z.

Various line types, plot symbols and colors may be obtained with
plot3(X,Y,Z,s) where s is a 1, 2 or 3 character string made from
the characters listed under the PLOT command.

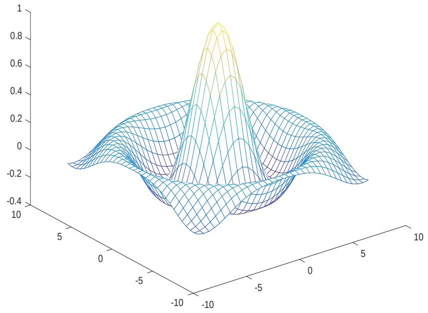
plot3(x1,y1,z1,s1,x2,y2,z2,s2,x3,y3,z3,s3,...)
combines the plots defined by the (x,y,z,s) fourtuples, where the x's, y's and z's are vectors or matrices and the s's are strings.

Example: A helix:

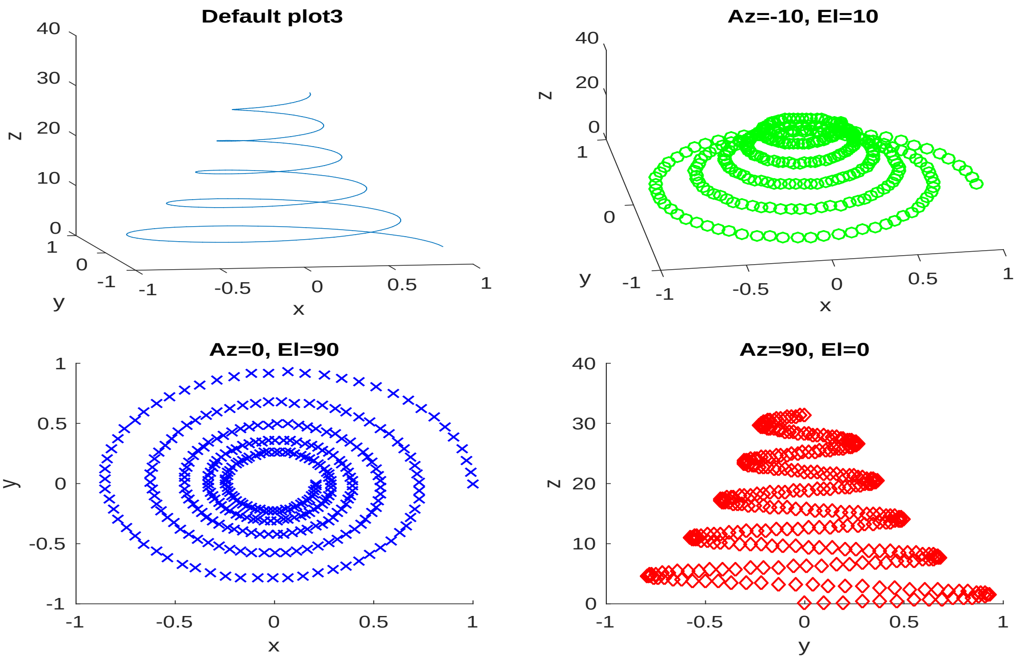
```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t);
```

plot3 returns a column vector of handles to line series objects, one handle per line. The X,Y,Z triples, or X,Y,Z,S quads, can be followed by parameter/value pairs to specify additional properties of the lines.

```
% plot z = sin(r)/r with r = sqrt(x^2 + y^2)
% -8 <= x <= 8
% -8 <= y <= 8
x = -8:0.5:8;
y = x;
[X,Y] = meshgrid(x,y);
R =sqrt(X.^2 + Y.^2) + eps;   % add eps to prevent R = 0
Z = sin(R)./R;
mesh(x,y,Z)
```



```
subplot(2,2,1);plot3(x,y,z);
xlabel('x');
ylabel('y');
zlabel('z');
view(-10,10);
title('Default plot3');
subplot(2,2,2);plot3(x,y,z,'og');
xlabel('x');
ylabel('y');
zlabel('z’);
view(-9,56);
title('Az=-10, El=10');
subplot(2,2,3);plot3(x,y,z,'xb');
xlabel('x');
ylabel('y');
zlabel('z');
view(0,90);
title('Az=0, El=90');
subplot(2,2,4);plot3(x,y,z,'dr');
xlabel('x');
ylabel('y');
zlabel('z');
view(90,0);
title('Az=90, El=0');
```

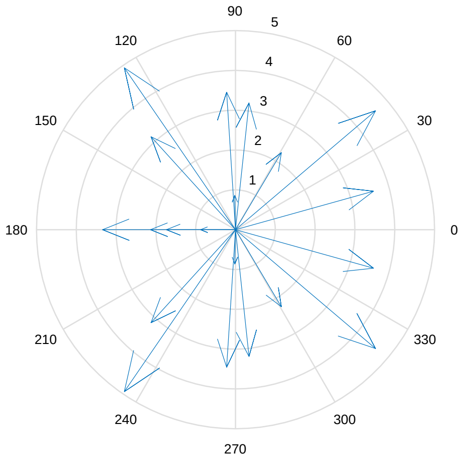


Examples:

How to plot three figures into one figure window?
How to plot more than one figure?
How to plot multiple values of y axis vs x axis?

Examples:

```
%% Create Compass Graph
% Create a compass graph of the eigenvalues of a random
matrix.

rng(0,'twister') % initialize random number generator
M = randn(20,20);
Z = eig(M);  %Z = eig(M) produces a column vector Z containing
the eigenvalues of a square matrix M.

figure
compass(Z)
```

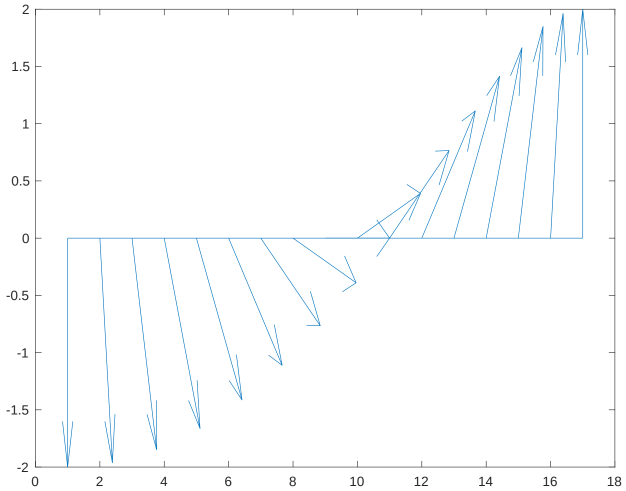


```
%% Create Feather Plot
%  feather(U,V) plots the velocity vectors with components U
and V as arrows emanating from equally spaced points along a
horizontal axis.
%  feather is useful for displaying direction and magnitude
data that is collected along a path.
```

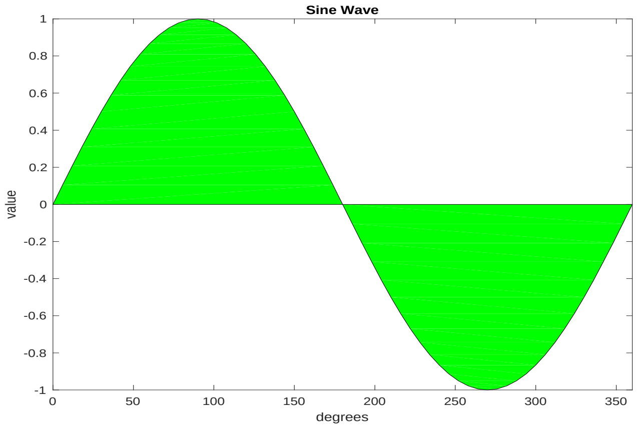
Example:

```
% Define |theta| as values between $-2\pi$ and $2\pi$. Define
|r| as a vector the same size as |theta|.

theta = -pi/2:pi/16:pi/2;
r = 2*ones(size(theta));

% Create a feather plot showing the direction of |theta|. Since
|feather|uses Cartesian coordinates, convert |theta| and |r| to
Cartesian coordinates using |pol2cart|.

[u,v] = pol2cart(theta,r);

% pol2cart Transform polar to Cartesian coordinates.
%     [X,Y] = pol2cart(TH,R) transforms corresponding elements
of data stored in polar coordinates (angle TH, radius R) to
```

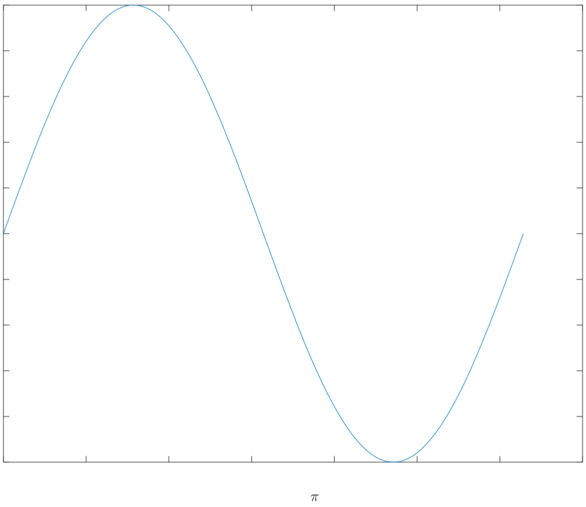
```
Cartesian coordinates X,Y.  The arrays TH and R must the same
size (or either can be scalar).  TH must be in radians.
feather(u,v)
```



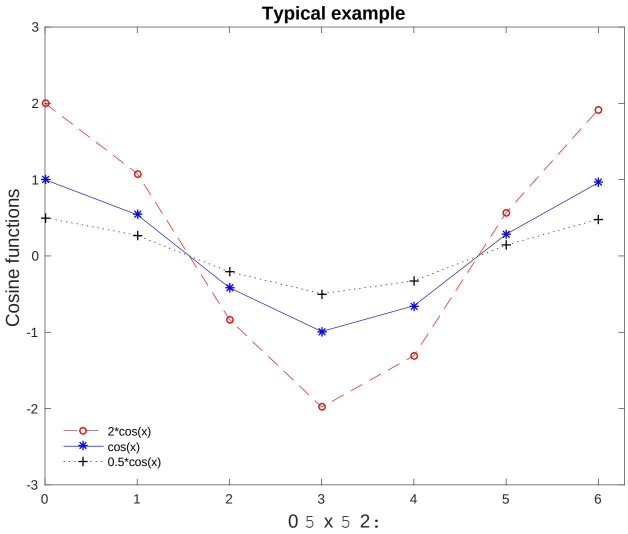
```
x = 0:6:360 ; % The values of x in degrees
rads = 2*pi*x./360; % The above convert to radians
y = sin(rads); % Calculate the values of y
fill(x,y,'g'); % Plot the graph
title('Sine Wave'); % Add a Title
xlabel('degrees'); % Label the x axis
ylabel('value'); % Label the y axis
grid; % Draw a grid on the graph
axis([0 360 -1 1]);
```



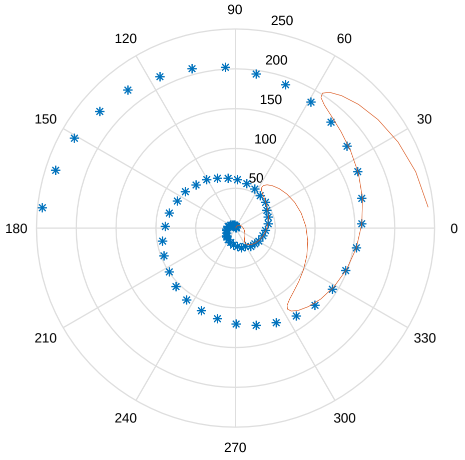
```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
%plot(x,y,'r')
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine function')
```



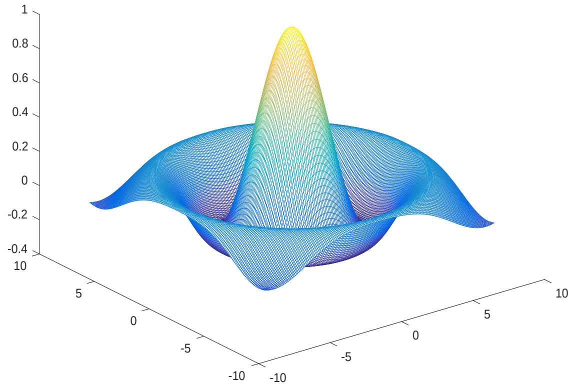
```
x = 0:2*pi;
y1 = 2*cos(x);
y2 = cos(x);
y3 = 0.5*cos(x);
%figure (3)
plot(x,y1,'--ro',x,y2,'-b*',x,y3,':k+','MarkerSize', 5 )
xlabel('0 \leq x \leq 2\pi','FontSize',14)
ylabel('Cosine functions','FontSize',14)
legend('2*cos(x)','cos(x)','0.5*cos(x)','Location','SouthWest)
legend('boxoff');
title('Typical example','FontSize',14)
axis([0 2*pi -3 3])
grid
```



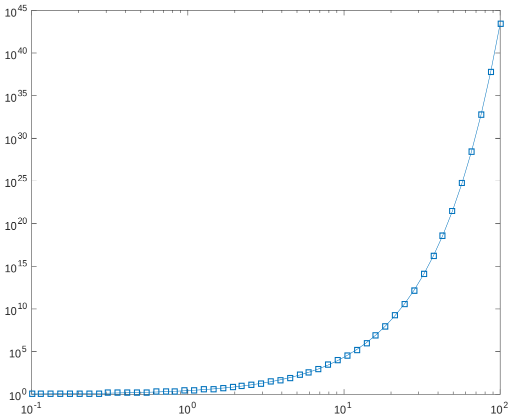
```
theta=0:0.2:5*pi;
rho=theta.^2;
polar(theta,rho,'*')
hold on
polar(sin(theta),rho)
hold off
```



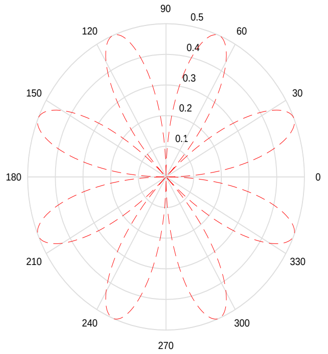
```
% plot z = sin(r)/r with r = sqrt(x^2 + y^2)
% -8 <= x <= 8
% -8 <= y <= 8
x = -8:0.1:8;
y = x;
[X,Y] = meshgrid(x,y);
R =sqrt(X.^2 + Y.^2) + eps; % add eps to prevent R = 0
Z = sin(R)./R;
mesh(x,y,Z)
```



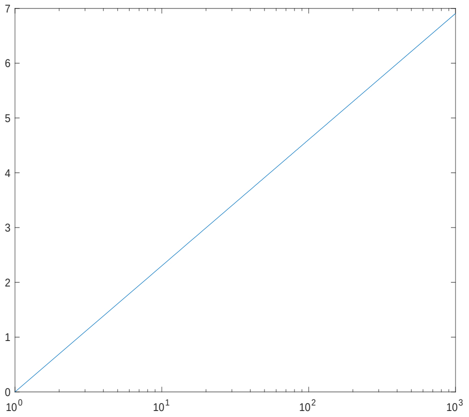
```
%% Logarithmic Scale for Both Axes
% Create a plot using a logarithmic scale for both the x-axis
and the
% y-axis. Set the |LineSpec| string so that |loglog| plots
using a line with square markers.
% Display the grid.
x = logspace(-1,2);
y = exp(x);
figure
loglog(x,y,'-s')
grid on
```



```
t = 0 : .01 : 2 * pi;
polar(t, sin(2 * t) .* cos(2 * t), '--r');
```

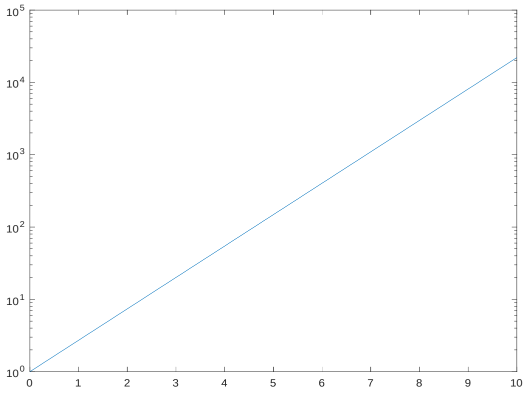


```
%semilogx Semi-log scale plot.
%semilogx(...) is the same as PLOT(...), except a
%logarithmic (base 10) scale is used for the X-axis.

 x=1:1000;
 y=log(x);
 figure
 semilogx(x,y)
```



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

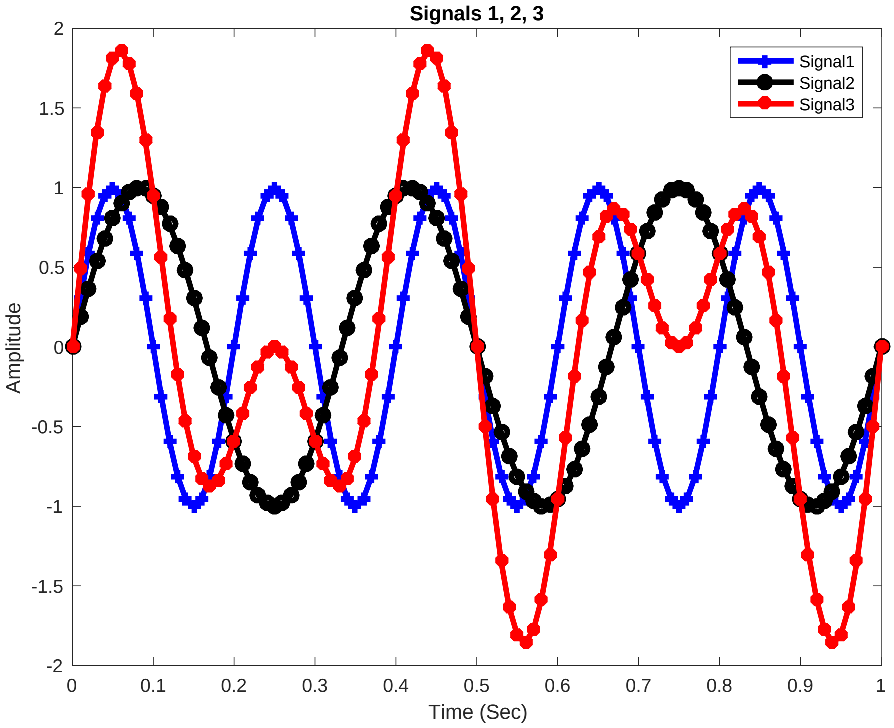
```
x = 0:0.1:10;
y = exp(x);
figure
semilogy(x,y)
```

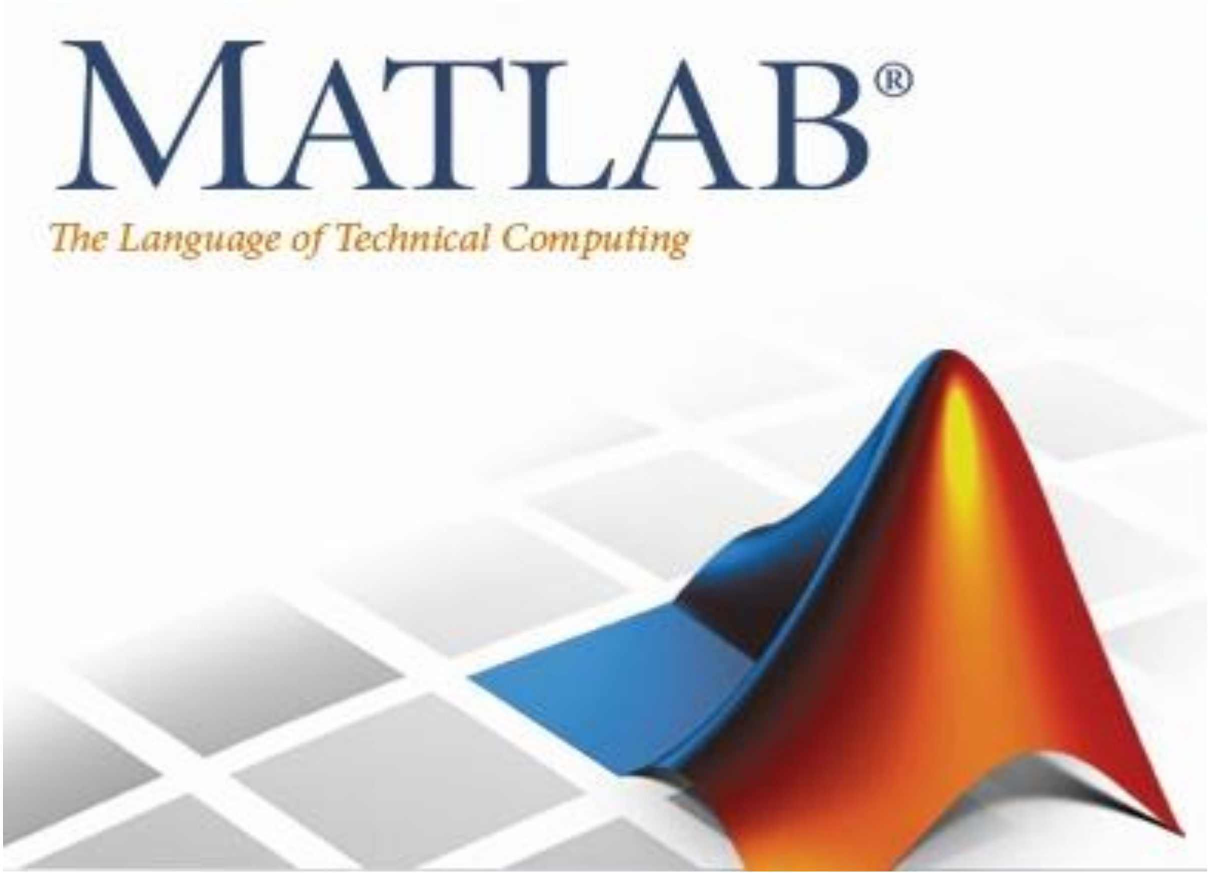


Example:

```
clear all;
close all;
clc;
%% Initialization
f1=5;           % Frequncy 1
f2=3;           %Frequency 2
fs=100;         % Sampling frequency
TimeAxis=0:1/fs:1;   % Time axis (one second)
Signal1=sin(2*pi*f1*TimeAxis);  % Signal 1
Signal2=sin(2*pi*f2*TimeAxis);  % Signal 2
Signal3=Signal1+Signal2;        % Third Signal
%% Plotting
figure
```

```
plot(TimeAxis,Signal1,'-b+','LineWidth',3,'MarkerSize',6)
hold on
plot(TimeAxis,Signal2,'-ko','LineWidth',3,'MarkerSize',6)
Signal3=Signal1+Signal2;            % Third Signal
hold on
plot(TimeAxis,Signal3,'-r*','LineWidth',3,'MarkerSize',6)
xlabel('Time (Sec)')
ylabel('Amplitude')
title('Signals 1, 2, 3')
legend('Signal1','Signal2','Signal3')
grid
```





Lecture 7

xlsread & xlswrite Functions in MATLAB

By: Abidulkarim K. I. Yasari

xlsread & xlswrite Functions in MATLAB

Xlsread:

Read Microsoft Excel spreadsheet file

Syntax:

- **num = xlsread('filename')**
- **num = xlsread('filename',sheet)**
- **num = xlsread('filename',xlRange)**
- **num = xlsread('filename',sheet,xlRange)**

Description

num = xlsread('filename') reads the first worksheet in the Microsoft® Excel® spreadsheet workbook named filename and returns the numeric data in a matrix.

num = xlsread('filename',sheet) reads the specified worksheet.

num = xlsread('filename',xlRange) reads from the specified range of the first worksheet in the workbook. Use Excel range syntax, such as 'A1:C3'.

num = xlsread('filename',sheet,xlRange) reads from the specified worksheet and range.

xlswrite:

Write Microsoft Excel spreadsheet file

Syntax:

xlswrite(filename,A)

xlswrite(filename,A,sheet)

xlswrite(filename,A,xlRange)

xlswrite(filename,A,sheet,xlRange)

Description:

xlswrite(filename,A)

writes array A to the first worksheet in Excel file, filename, starting at cell A1.

xlswrite(filename,A,sheet) writes to the specified worksheet.

xlswrite(filename,A,xlRange) writes to the rectangular region specified by xlRange in the first worksheet of the file.

xlswrite(filename,A,sheet,xlRange) writes to the specified sheet and range, xlRange.

Write Data to a Spreadsheet

Write a 7-element vector to an Excel file, testdata.xlsx.

```
filename = 'testdata.xlsx';
A = [12.7, 5.02, -98, 63.9, 0, -.2, 56];
xlswrite(filename,A)
```

Write Data to a Specific Sheet and Range in a Spreadsheet

Write mixed text and numeric data to an Excel file, testdata.xlsx, starting at cell E1 of Sheet2.

```
filename = 'testdata.xlsx';
A = {'Time','Temperature'; 12,98; 13,99; 14,97};
sheet = 2;
xlRange = 'E1';
xlswrite(filename,A,sheet,xlRange)
```

Examples:

Read Data from First Worksheet into Numeric Array

Create an Excel file named myExample.xlsx.

values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8,9};

```
headers = {'First', 'Second', 'Third'};
xlswrite('myExample.xlsx', [headers; values]);
```

Sheet1 of myExample.xlsx contains:

First	Second	Third
1	2	3
4	5	x
7	8	9

Read data from the first worksheet.

```
filename = 'myExample.xlsx';
A = xlsread(filename)
A =
    1    2    3
    4    5  NaN
    7    8    9
```

xlsread returns the numeric data in array A.

Read a Specific Range of Data

Read a specific range of data from the Excel file in the previous example.

```
filename = 'myExample.xlsx';
sheet = 1;
xlRange = 'B2:C3';
subsetA = xlsread(filename, sheet, xlRange)
subsetA =
    2    3
    5  NaN
```

Read a Column of Data

Read the second column of data from the Excel file in the first example.

```
filename = 'myExample.xlsx';

columnB = xlsread(filename,'B:B')
columnB =
          2
          5
          8
```

For better performance, specify the row numbers in the range, as shown in the previous example.

Request Numeric, Text, and Unprocessed Data

Request the numeric data, text, and a copy of the unprocessed (raw) data from the Excel file in the first example.

```
[ndata, text, alldata] = xlsread('myExample.xlsx')
ndata =
     1     2     3
     4     5   NaN
     7     8     9

text =
    'First'    'Second'    'Third'
    ''         ''          ''
    ''         ''          'x'

alldata =
    'First'    'Second'    'Third'
    [    1]    [     2]    [    3]
    [    4]    [     5]        'x'
    [    7]    [     8]    [    9]
```

xlsread returns numeric data in array ndata, text data in cell array text, and unprocessed data in cell array alldata.

More Examples:

Example 1

```
filename = 'myExample.xlsx';
A = xlsread(filename)
```

Example 2

```
filename = 'myExample.xlsx';
columnB = xlsread(filename,'B:B')
```

Example 3

```
filename = 'myExample.xlsx';
sheet = 1;
xlRange = 'B2:C3';
subsetA = xlsread(filename, sheet, xlRange)
```

Example 4

```
[ndata, text, alldata] = xlsread('myExample.xlsx')
```

Example 5

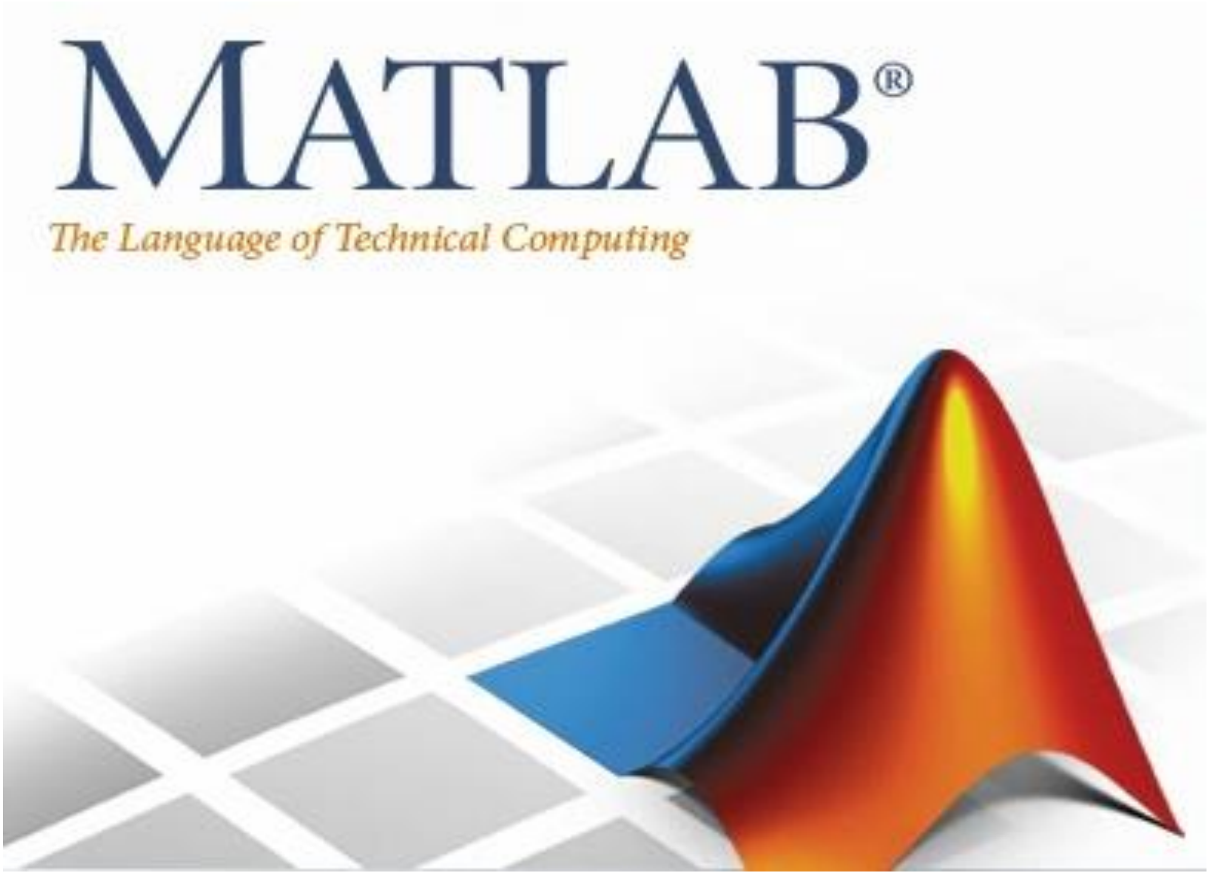
```
values = {1, 2, 3 ; 4, 5, 'x' ; 7, 8, 9};
headers = {'First', 'Second', 'Third'};
xlswrite('myExample.xlsx', [headers; values]);
```

Example 6

```
filename = 'testdata.xlsx';
A = [12.7, 5.02, -98, 63.9, 0, -.2, 56];
xlswrite(filename,A)
```

Example 7

```
filename = 'testdata1.xlsx';
A = {'Time','Temperature'; 12,98; 13,99; 14,97};
sheet = 2;
xlRange = 'j6';
xlswrite(filename,A,sheet,xlRange)
```



Lecture 8

Symbolic Numbers, Variables, and Expressions in MATLAB

By: Abidulkarim K. I. Yasari

The following sections will show how to create symbolic numbers, variables, and expressions.

Create Symbolic Numbers

You can create symbolic numbers by using sym. Symbolic numbers are exact representations, unlike floating-point numbers.

Create a symbolic number by using sym and compare it to the same floating-point number.

```
>>sym(1/3)
ans =
         1/3
```

But if we write the command:

```
>>1/3
ans =
          0.3333
```

The symbolic number is represented in exact rational form, while the floating-point number is a decimal approximation. The symbolic result is not indented, while the standard MATLAB result is indented.

Calculations on symbolic numbers are exact. Demonstrate this exactness by finding sin(pi) symbolically and numerically. The symbolic result is exact, while the numeric result is an approximation.

```
>> sin(sym(pi))
ans =
          0
>> sin(pi)

ans =

        1.224646799147353e-16
```

To learn more about symbolic representation of numbers, see Numeric to Symbolic Conversion.

Create Symbolic Variables:

You can use two ways to create symbolic variables, syms and sym. The syms syntax is a shorthand for sym.

Create symbolic variables x and y using syms and sym respectively.

```
>> syms x
>> y = sym('y')
y =
          y
```

The first command creates a symbolic variable x in the MATLAB workspace with the value x assigned to the variable x. The second command creates a symbolic variable y with value y. Therefore, the commands are equivalent.

With syms, you can create multiple variables in one command. Create the variables a, b, and c.

```
>> syms a b c
```

If you want to create many variables, the syms syntax is inconvenient. Instead of using syms, use sym to create many numbered variables.

Create the variables a1, ..., a20.

```
>> A = sym('a', [1 20])

A =

    [ a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16, a17, a18, a19, a20]
```

The syms command is a convenient shorthand for the sym syntax. Use the sym syntax when you create many variables, when the variable value differs from the variable name, or when you create a symbolic number, such as sym(5).

Create Symbolic Expressions:

Suppose you want to use a symbolic variable to represent the golden ratio:

$$\emptyset = \frac{1 + \sqrt{5}}{2}$$

The command

```
>> phi = (1 + sqrt(sym(5)))/2

phi =

            5^(1/2)/2 + 1/2
```

achieves this goal. Now you can perform various mathematical operations on phi. For example:

```
>> f = phi^2 - phi - 1
f =
          (5^(1/2)/2 + 1/2)^2 - 5^(1/2)/2 - 3/2
```

Now suppose you want to study the quadratic function f = ax2 + bx + c. First, create the symbolic variables a, b, c, and x:

```
syms a b c x
```

Then, assign the expression to f:

```
>> f = a*x^2 + b*x + c

f =
            a*x^2 + b*x + c
```

syms function to create a symbolic expression that is a constant. For example, to create the expression whose value is 5, enter f = sym(5). The command f = 5 does not define f as a symbolic expression.

Reuse Names of Symbolic Objects:

If you set a variable equal to a symbolic expression, and then apply the syms command to the variable, MATLAB software removes the previously defined expression from the variable. For example,

```
>> syms a b

>> f=a+b

f =

         a + b
```

If later you enter

```
>> syms f
```

or

```
>> sym f

ans =

         f
```

then MATLAB removes the value a + b from the expression f:

```
>> f

f =

         f
```

You can use the syms command to clear variables of definitions that you previously assigned to them in your MATLAB session. However, syms does not clear the following assumptions of the variables: complex, real, integer, and positive. These assumptions are stored separately from the symbolic object.

Create Symbolic Functions:

You also can use sym and syms to create symbolic functions. For example, you can create an arbitrary function f(x, y) where x and y are function variables. The simplest way to create an arbitrary symbolic function is to use syms:

```
syms f(x, y)
```

This syntax creates the symbolic function f and symbolic variables x and y. If instead of an arbitrary symbolic function you want to create a function defined by a particular mathematical expression, use this two-step approach. First, create symbolic variables representing the arguments of the function:

```
>>syms x y
```

Then assign a mathematical expression to the function. In this case, the assignment operation also creates the new symbolic function:

```
>>f(x, y) = x^3*y^3
f(x, y) =
          x^3*y^3
```

Note that the body of the function must be a symbolic number, variable, or expression. Assigning a number, such as f(x,y) = 1, causes an error.

After creating a symbolic function, you can differentiate, integrate, or simplify it, substitute its arguments with values, and perform other mathematical operations. For example, find the second derivative on f(x, y) with respect to variable y. The result d2fy is also a symbolic function.

```
>> syms x y

>> f(x, y) = x^3*y^3

f(x, y) =

        x^3*y^3

>> d2fy = diff(f, y, 2)

d2fy(x, y) =

            6*x^3*y
```

Now evaluate f(x, y) for x = y + 1:

```
>> f(y + 1, y)
```

```
ans =

        y^3*(y + 1)^3
```

Create Symbolic Matrices:

Use Existing Symbolic Variables

A circulant matrix has the property that each row is obtained from the previous one by cyclically permuting the entries one step forward. For example, create the symbolic circulant matrix whose elements are a, b, and c, using the commands:

```
>> syms a b c

>> A=[a b c;c a b;b c a]

A =

        [ a, b, c]

        [ c, a, b]

        [ b, c, a]
```

Since matrix A is circulant, the sum of elements over each row and each column is the same. Find the sum of all the elements of the first row:

```
>> sum(A(1,:))

ans =

        a + b + c

>> sum(A(2,:))

ans =

        a + b + c

>> sum(A(3,:))

ans =

        a + b + c

>> sum(A(:,1))
```

```
ans =

        a + b + c
```

or

```
>> sum(A(1:2,1:2))

ans =

        [ a + c, a + b]
```

To check if the sum of the elements of the first row equals the sum of the elements of the second column, use the isAlways function:

```
isAlways(sum(A(1,:)) == sum(A(:,2)))
```

The sums are equal:

```
ans =
    1
```

From this example, you can see that using symbolic objects is very similar to using regular MATLAB numeric objects.

Generate Elements While Creating a Matrix

The sym function also lets you define a symbolic matrix or vector without having to define its elements in advance. In this case, the sym function generates the elements of a symbolic matrix at the same time that it creates a matrix. The function presents all generated elements using the same form: the base (which must be a valid variable name), a row index, and a column index. Use the first argument of sym to specify the base for the names of generated elements. You can use any valid variable name as a base. To check whether the name is a valid variable name, use the isvarname function. By default, sym separates a row index and a column index by underscore. For example, create the 2-by-4 matrix A with the elements X1_1, ..., X2_4:

```
>> A = sym('X', [2 4])

A =

        [ X1_1, X1_2, X1_3, X1_4]

        [ X2_1, X2_2, X2_3, X2_4]
```

To control the format of the generated names of matrix elements, use %d in the first argument:

```
>> A = sym('X%d', [2 4])
A =
[ X11, X12, X13, X14]
[ X21, X22, X23, X24]
```

Create Matrix of Symbolic Numbers

A particularly effective use of sym is to convert a matrix from numeric to symbolic form. The command:

```
>>A = hilb(3)       % generates the 3-by-3 Hilbert matrix

A =
   1.0000    0.5000    0.3333
   0.5000    0.3333    0.2500
   0.3333    0.2500    0.2000
```

By applying sym to A

```
>>A = sym(A)
```

you can obtain the precise symbolic form of the 3-by-3 Hilbert matrix:

```
A =
           [   1, 1/2, 1/3]
           [ 1/2, 1/3, 1/4]
           [ 1/3, 1/4, 1/5]
```

Perform Symbolic Computations:

Differentiate Symbolic Expressions

With the Symbolic Math Toolbox™ software, you can find

- Derivatives of single-variable expressions
- Partial derivatives
- Second and higher order derivatives
- Mixed derivatives

Expressions with One Variable

To differentiate a symbolic expression, use the diff command. The following example illustrates how to take a first derivative of a symbolic expression:

```
>> syms x

>> f=sin(x)^2;

>> diff(f)

ans =

        2*cos(x)*sin(x)
```

Partial Derivatives

For multivariable expressions, you can specify the differentiation variable. If you do not specify any variable, MATLAB® chooses a default variable by its proximity to the letter x:

```
>>syms x y
>>f = sin(x)^2 + cos(y)^2;
>>diff(f)
ans =
           2*cos(x)*sin(x)
```

To differentiate the symbolic expression f with respect to a variable y, enter:

```
>>syms x y
>>f = sin(x)^2 + cos(y)^2;
>>diff(f, y)
ans =
          -2*cos(y)*sin(y)
```

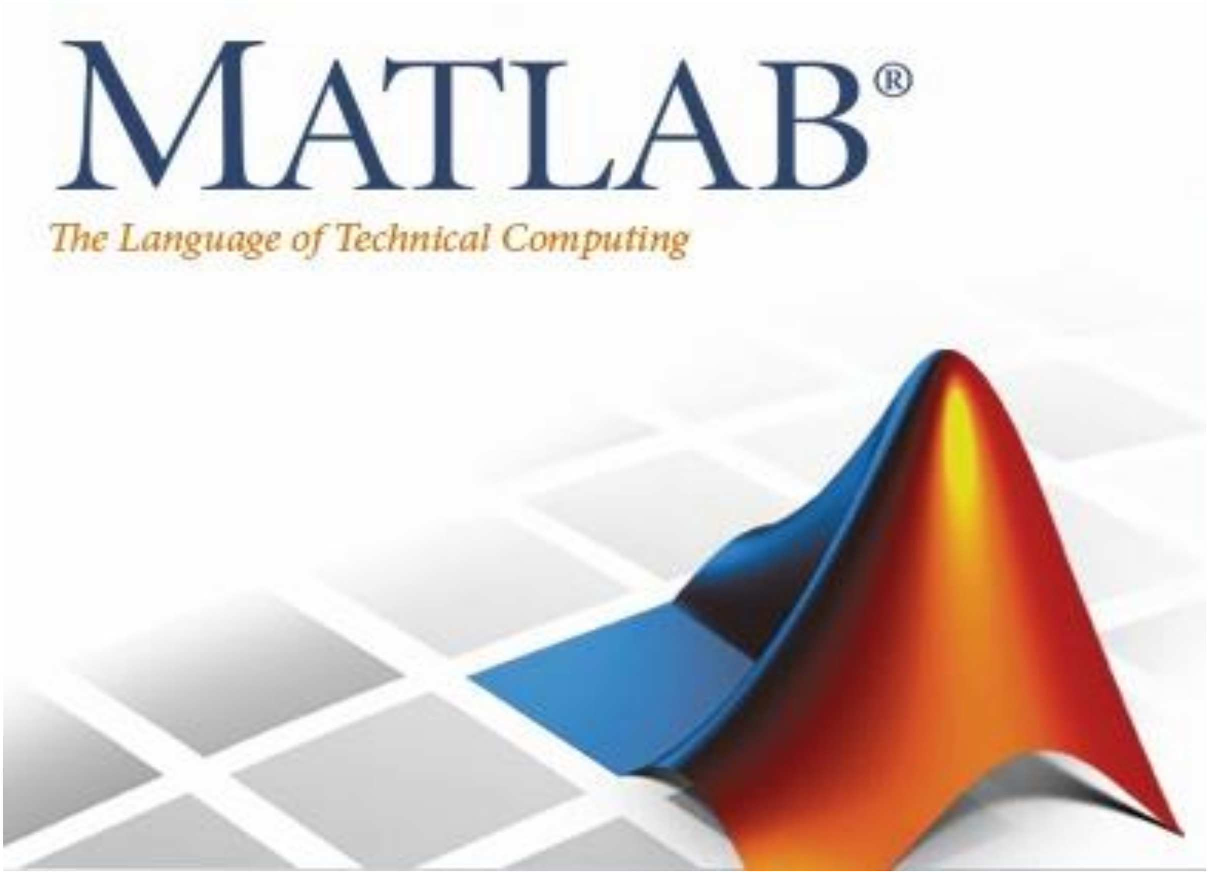
Second Partial and Mixed Derivatives

To take a second derivative of the symbolic expression f with respect to a variable y, enter:

```
>>syms x y
>>f = sin(x)^2 + cos(y)^2;
>>diff(f, y, 2)
ans =
          2*sin(y)^2 - 2*cos(y)^2
```

You get the same result by taking derivative twice: diff(diff(f, y)). To take mixed derivatives, use two differentiation commands. For example:

```
>>syms x y
>>f = sin(x)^2 + cos(y)^2;
>>diff(diff(f, y), x)
ans =
            0
```



Lecture 9

Integrate Symbolic Expressions in MATLAB

By: Abidulkarim K. I. Yasari

Integrate Symbolic Expressions

You can perform symbolic integration including:

- Indefinite and definite integration
- Integration of multivariable expressions

Indefinite Integrals of One-Variable Expressions

Suppose you want to integrate a symbolic expression. The first step is to create the symbolic expression:

```
>> syms x
>> f = sin(x)^2;
```

To find the indefinite integral, enter

```
>> int(f)
ans =
            x/2 - sin(2*x)/4
```

Indefinite Integrals of Multivariable Expressions

If the expression depends on multiple symbolic variables, you can designate a variable of integration. If you do not specify any variable, MATLAB chooses a default variable by the proximity to the letter x:

```
>> syms x y n;
>> f = x^n + y^n;
>> int(f)
ans =
            x*y^n + (x*x^n)/(n + 1)
```

You also can integrate the expression f = x^n + y^n with respect to y

```
>> syms x y n
>> f = x^n + y^n;
>> int(f, y)
ans =
            x^n*y + (y*y^n)/(n + 1)
```

If the integration variable is n, enter

```
>> syms x y n
>> f = x^n + y^n;
>> int(f, n)
ans =
            x^n/log(x) + y^n/log(y)
```

Definite Integrals

To find a definite integral, pass the limits of integration as the final two arguments of the int function:

```
>> syms x y n
>> f = x^n + y^n;
>> int(f,1,10)
ans =
piecewise([n == -1, log(10) + 9/y], [n ~= -1, (10*10^n - 1)/(n + 1) + 9*y^n])
```

where *piecewise* : Conditionally defined expression or function, for example:

pw = piecewise(cond1, val1, cond2, val2,..., otherwiseVal)

If the int function cannot compute an integral, it returns an unresolved integral:

```
>> syms x
>> int(sin(sinh(x)))
ans =
        int(sin(sinh(x)), x)
```

Solve Equations

You can solve different types of symbolic equations including:

- Algebraic equations with one symbolic variable
- Algebraic equations with several symbolic variables
- Systems of algebraic equations

Solve Algebraic Equations with One Symbolic Variable

Use the double equal sign (==) to define an equation. Then you can solve the equation by calling the solve function. For example, solve this equation:

```
>> syms x
>> solve(x^3 - 6*x^2 == 6 - 11*x)
ans =
       1
       2
       3
```

```
>> syms x
>> solve('x+3=2',x)
ans =
        -1
```

```
>> syms x
>> solve('x^2+2*x-3=12',x)
ans =
            -5
             3
```

```
>> syms x
>> solve('x^2+2*x-3=11',x)
ans =

            - 15^(1/2) - 1
            15^(1/2) – 1
```

```
>> double(ans)
ans =
            -4.872983346207417
            2.872983346207417
```

```
>>  syms x
>> solve('sin(x)-3',x)
ans =
      asin(3)
      pi - asin(3)
```

```
>> double(ans)
ans =

  1.570796326794897 - 1.762747174039086i
  1.570796326794897 + 1.762747174039086i
```

It is complex because 3 is out of the sin range or region

```
>> syms x
>> solve('sin(x)=1/2',x)
ans =
    pi/6
    (5*pi)/6
>> double(ans)
ans =
   0.523598775598299
   2.617993877991494
```

The answer is not complex because =1/2 in the region of sin

If you do not specify the right side of the equation, solve assumes that it is zero:

```
>> syms x
>> solve(x^3 - 6*x^2 + 11*x - 6)
```

```
ans =
            1
            2
            3
```

Solve Algebraic Equations with Several Symbolic Variables

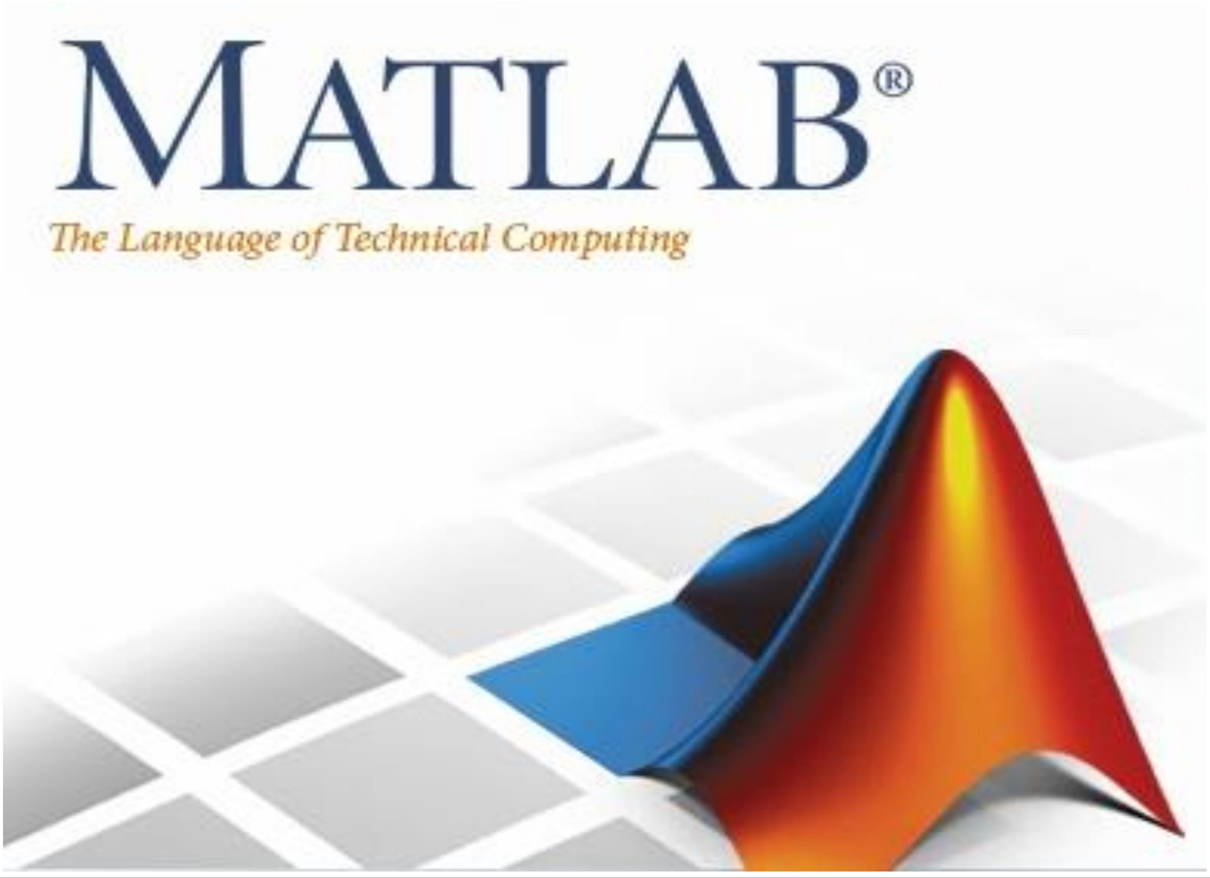
If an equation contains several symbolic variables, you can specify a variable for which this equation should be solved. For example, solve this multivariable equation with respect to y:

```
>> syms x y
>> solve(6*x^2 - 6*x^2*y + x*y^2 - x*y + y^3 - y^2 == 0, y)
ans =
   1
   2*x
 -3*x
```

```
>> syms x y z
>> solve('x+2*y=-3*z',y)
ans =
            - x/2 - (3*z)/2

>> pretty(ans)
           x   3 z
         - - - ---
           2    2


>> solve('x+2*y=-3*z',x)

ans =

         - 2*y - 3*z
>> solve('x+2*y=-3*z',z)

ans =

         - x/3 - (2*y)/3

------------------------------

>> syms x y z

>> solve('sin(x)+exp(z)=3*y',y)
```

```
ans =

        exp(z)/3 + sin(x)/3


>> solve('sin(x)+exp(z)=3*y',x)

ans =

     asin(3*y - exp(z))

     pi - asin(3*y - exp(z))


>> solve('sin(x)+exp(z)=3*y',z)

ans =

        log(3*y - sin(x))
```

If you do not specify any variable, you get the solution of an equation for the alphabetically closest to x variable.



Lecture 10

Solve Systems of Algebraic Equations

By: Abidulkarim K. I. Yasari

Solve Systems of Algebraic Equations:

You also can solve systems of equations. For example:

```
>>syms x y z
>>[x, y, z] = solve(z == 4*x, x == y, z == x^2 + y^2)
x =
 0
 2

y =
 0
 2

z =
0
8
```

Simplify Symbolic Expressions:

Symbolic Math Toolbox provides a set of simplification functions allowing you to manipulate the output of a symbolic expression. For example, the following polynomial of the golden ratio phi

```
>>phi = (1 + sqrt(sym(5)))/2;
>>f = phi^2 - phi - 1
```

returns

```
f =
(5^(1/2)/2 + 1/2)^2 - 5^(1/2)/2 - 3/2
```

You can simplify this answer by entering

```
>>simplify(f)
```

and get a very short answer:

```
ans =
0
```

Symbolic simplification is not always so straightforward. There is no universal simplification function, because the meaning of a simplest representation of a symbolic expression cannot be defined clearly. Different problems require different forms of the same mathematical expression. Knowing what form is more effective for solving your particular problem, you can choose the appropriate simplification function.

For example, to show the order of a polynomial or symbolically differentiate or integrate a polynomial, use the standard polynomial form with all the parentheses multiplied out and all the similar terms summed up. To rewrite a polynomial in the standard form, use the expand function:

```
>>syms x
>>f = (x ^2- 1)*(x^4 + x^3 + x^2 + x + 1)*(x^4 - x^3 + x^2 - x + 1);
>>expand(f)
ans =
x^10 – 1
```

The factor simplification function shows the polynomial roots. If a polynomial cannot be factored over the rational numbers, the output of the factor function is the standard polynomial form. For example, to factor the third-order polynomial, enter:

```
>>syms x
```

```
>>g = x^3 + 6*x^2 + 11*x + 6;
>>factor(g)
ans =
[ x + 3, x + 2, x + 1]
```

The nested (Horner) representation of a polynomial is the most efficient for numerical evaluations:

```
>>syms x
>>h = x^5 + x^4 + x^3 + x^2 + x;
>>horner(h)
ans =
x*(x*(x*(x*(x + 1) + 1) + 1) + 1)
```

Substitutions in Symbolic Expressions:
Substitute Symbolic Variables with Numbers

You can substitute a symbolic variable with a numeric value by using the subs function. For example, evaluate the symbolic expression f at the point x = 1/3:

```
>>syms x
>>f = 2*x^2 - 3*x + 1;
>>subs(f, 1/3)
ans =
2/9
```

The subs function does not change the original expression f:

```
>>f
f =
2*x^2 - 3*x + 1
```

Substitute in Multivariate Expressions

When your expression contains more than one variable, you can specify the variable for which you want to make the substitution. For example, to substitute the value x = 3 in the symbolic expression

```
>>syms x y
>>f = x^2*y + 5*x*sqrt(y);
```

enter the command

```
>>subs(f, x, 3)
```

```
ans =

9*y + 15*y^(1/2)
```

Substitute One Symbolic Variable for Another

You also can substitute one symbolic variable for another symbolic variable. For example to replace the variable y with the variable x, enter

```
>>subs(f, y, x)

ans =

x^3 + 5*x^(3/2)
```

Substitute a Matrix into a Polynomial

You can also substitute a matrix into a symbolic polynomial with numeric coefficients. There are two ways to substitute a matrix into a polynomial: element by element and according to matrix multiplication rules.

Element-by-Element Substitution. To substitute a matrix at each element, use the subs command:

```
>>syms x

>>f = x^3 - 15*x^2 - 24*x + 350;

>>A = [1 2 3; 4 5 6];

>>subs(f,A)

ans =

[ 312, 250,  170] %the first element will be the value of f while x =1 and

[  78, -20, -118] % the second element while x = 2 and so on
```

You can do element-by-element substitution for rectangular or square matrices.

Substitution in a Matrix Sense:

If you want to substitute a matrix into a polynomial using standard matrix multiplication rules, a matrix must be square. For example, you can substitute the magic square A into a polynomial f:

1. Create the polynomial:
2. >>syms x

   >>f = x^3 - 15*x^2 - 24*x + 350;
3. Create the magic square matrix:

```
>>A = magic(3)

A =

    8     1     6

    3     5     7
    4     9     2
```

4. Get a row vector containing the numeric coefficients of the polynomial f:

```
>>b = sym2poly(f)
b =
     1   -15   -24   350
```

5. Substitute the magic square matrix A into the polynomial f. Matrix A replaces all occurrences of x in the polynomial. The constant times the identity matrix eye(3) replaces the constant term of f:

```
>>A^3 - 15*A^2 - 24*A + 350*eye(3)
ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

The polyvalm command provides an easy way to obtain the same result:

```
>>polyvalm(b,A)
ans =
   -10     0     0
     0   -10     0
     0     0   -10
```

Substitute the Elements of a Symbolic Matrix

To substitute a set of elements in a symbolic matrix, also use the subs command. Suppose you want to replace some of the elements of a symbolic circulant matrix A

```
>>syms a b c
>>A = [a b c; c a b; b c a]
A =
[ a, b, c]
[ c, a, b]
[ b, c, a]
```

To replace the (2, 1) element of A with beta and the variable b throughout the matrix with variable alpha, enter

```
>>alpha = sym('alpha');
>>beta = sym('beta');
```

```
>>A(2,1) = beta;
>>A = subs(A,b,alpha)
```

The result is the matrix:

```
A =
[     a, alpha,     c]
[  beta,     a, alpha]
[ alpha,     c,     a]
```

Plot Symbolic Functions

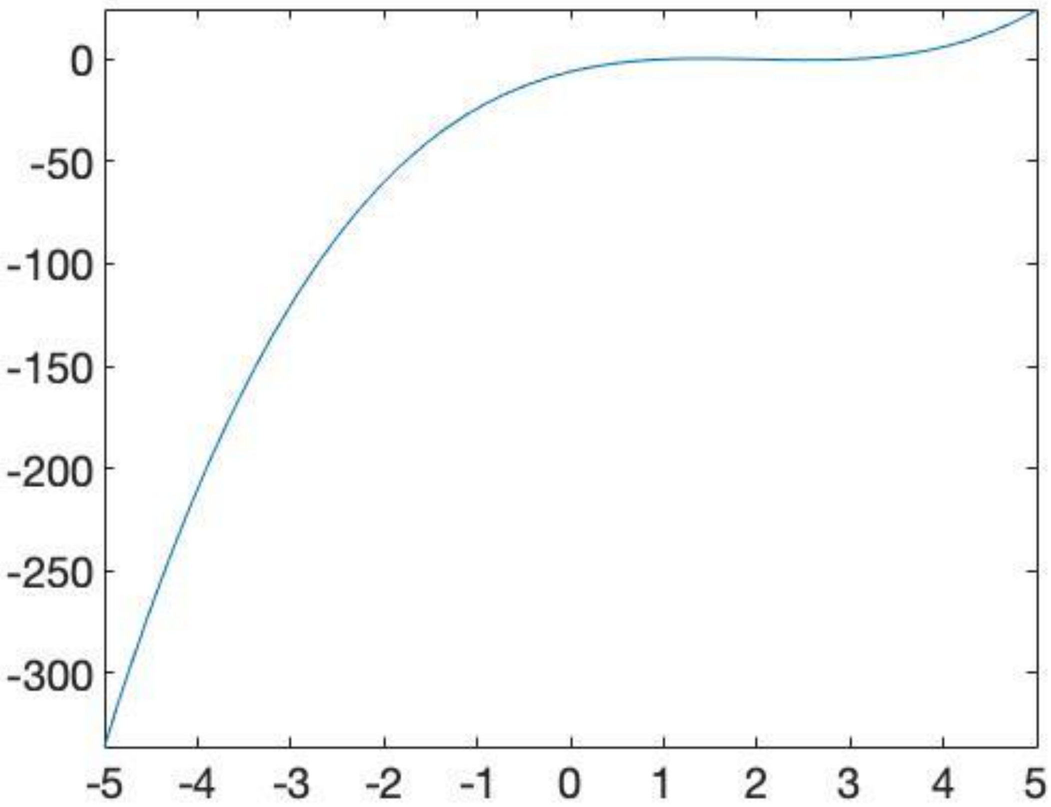
Symbolic Math Toolbox provides the plotting functions:

- fplot to create 2-D plots of symbolic expressions, equations, or functions in Cartesian coordinates.
- fplot3 to create 3-D parametric plots.
- ezpolar to create plots in polar coordinates.
- fsurf to create surface plots.
- fcontour to create contour plots.
- fmesh to create mesh plots.

Create a 2-D line plot by using fplot. Plot the expression .

```
>>syms x
>>f = x^3 - 6*x^2 + 11*x - 6;
>>fplot(f)
```

The output will be the following figure bellow:



Add labels for the x- and y-axes. Generate the title by using texlabel(f). Show the grid by using grid on.

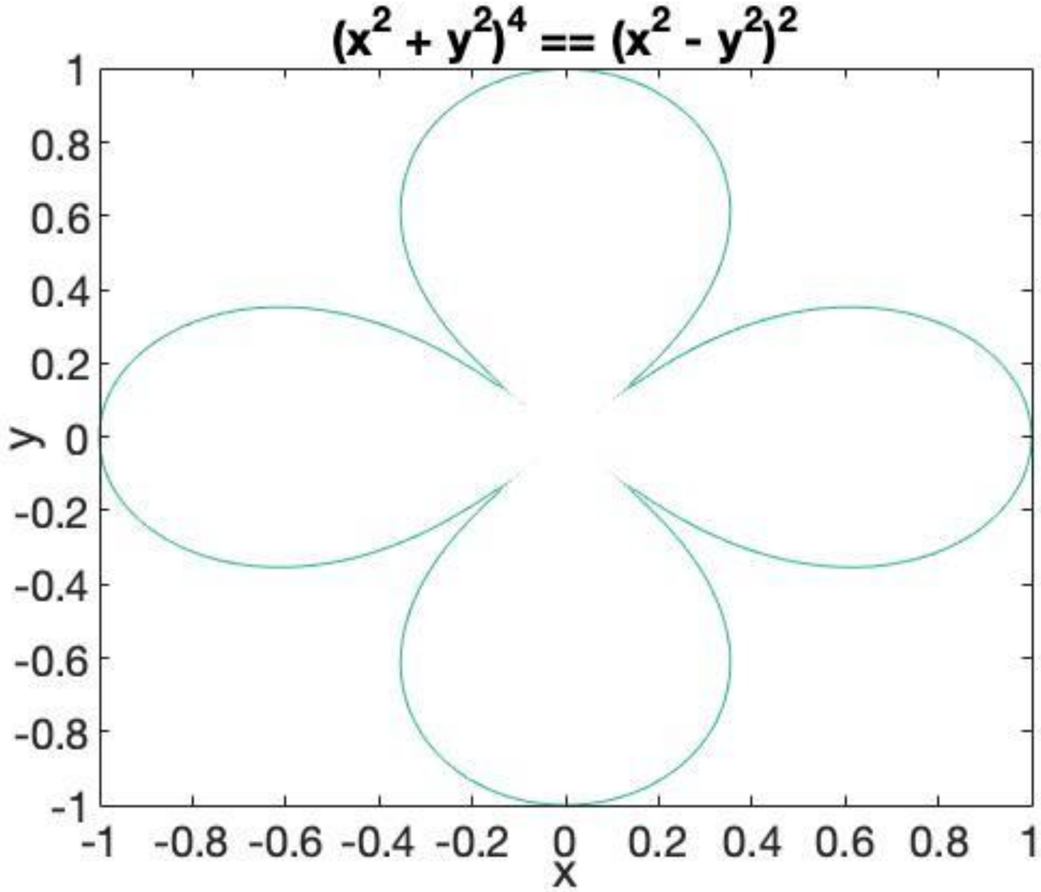
```
>>xlabel('x')
>>ylabel('y')
>>title(texlabel(f))
>>grid on
```

Plot equations and implicit functions using ezplot.

Plot the equation over.

```
syms x y
eqn = (x^2 + y^2)^4 == (x^2 - y^2)^2;
ezplot(eqn, [-1 1])
```

The output as in the following figure:

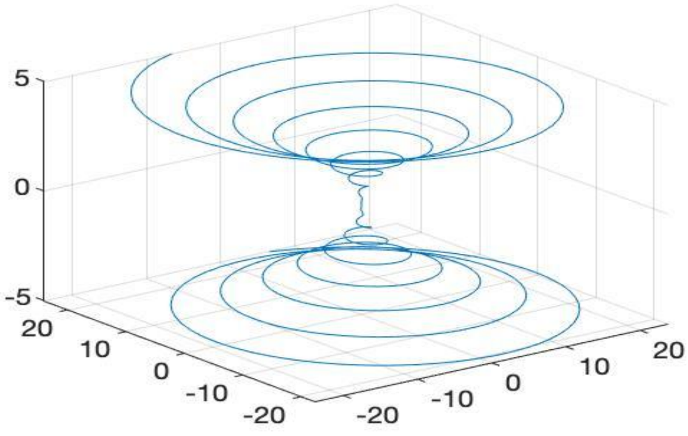


3-D Plot

Plot 3-D parametric lines by using fplot3.

Plot the parametric line

```
>>syms t
>>fplot3(t^2*sin(10*t), t^2*cos(10*t), t)
```

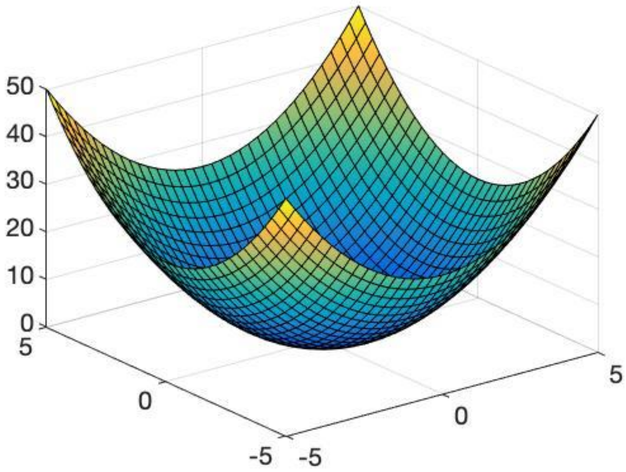


Create Surface Plot

Create a 3-D surface by using fsurf.

Plot the paraboloid.

```
syms x y
fsurf(x^2 + y^2)
```



Solve Multivariate Equations and Assign Outputs to Variables:

Avoid ambiguities when solving equations with symbolic parameters by specifying the variable for which you want to solve an equation. If you do not specify the variable, solve chooses a variable using symvar. First, solve the quadratic equation without specifying a variable. solve chooses *x* to return the familiar solution. Then solve the quadratic equation for a to return the solution for a.

```
>> syms a b c x
>> sol = solve(a*x^2 + b*x + c == 0)
sol =
 -(b + (b^2 - 4*a*c)^(1/2))/(2*a)
 -(b - (b^2 - 4*a*c)^(1/2))/(2*a)

>> sola = solve(a*x^2 + b*x + c == 0, a)
sola =
-(c + b*x)/x^2
```

When solving for more than one variable, the order in which you specify the variables defines the order in which the solver returns the solutions.

Solve this system of equations and assign the solutions to variables solv and solu by specifying the variables explicitly. The solver returns an array of solutions for each variable.

```
>>syms u v
>>[solv, solu] = solve([2*u^2 + v^2 == 0, u - v == 1], [v, u])
solv =
 - (2^(1/2)*1i)/3 - 2/3
   (2^(1/2)*1i)/3 - 2/3
solu =
 1/3 - (2^(1/2)*1i)/3
 (2^(1/2)*1i)/3 + 1/3
```

Entries with the same index form the solutions of a system.

```
>>solutions = [solv, solu]

solutions =

[ - (2^(1/2)*1i)/3 - 2/3, 1/3 - (2^(1/2)*1i)/3]

[   (2^(1/2)*1i)/3 - 2/3, (2^(1/2)*1i)/3 + 1/3]
```

A solution of the system is v = - (2^(1/2)*1i)/3 - 2/3, and u = 1/3 - (2^(1/2)*1i)/3.