BIRLA INSTITUTE OF APPLIED SCIENCES BHIMTAL

Department of Electronics and Communication Engineering

LAB MANUAL

B.Tech (CSE) VI Semester

MATLAB Programming

(BCSP-606)

Table of Contents

MODULE 1: INTRODUCTION

1.1 Basic MATLAB Program 1

1.2 Looping 4

1.3 Branching 6

1.4 Array and Matrix 8

1.5 Graphic application 11

1.6 External (user-defined) function 14

1.7 Use external files and prompt for input and output 16

Example 1.1 to 1.50

MODULE 2: PROGRAMMING IN MATLAB

2.1 M-File Scripts 18

2.1.1 Script side-effects 19

2.2 M-File Function9 19

2.2.1 Format of an M-File function 19

2.2.2 Input and Output arguments 20

2.3 Input to a script file 20

2.4 Output commands 21

2.5 Decision making statements 21

2.5.1 The if statement 21

2.5.2 The switch statement 22

2.6 Loop control statements 23

2.6.1 The ‘‘for...end’’ loop 23

2.6.2 The ‘‘while...end’’ loop 24

2.6.3 Other flow structures 24

2.7 Basic input/output (I/O) 24

2.7.1 The disp function 24

2.7.2 The input function 25

2.7.3 The save and load functions 25

2.8 Formatted I/O 25

2.8.1 fprintf 26

2.8.2 sprintf 26

2.9 File and Folder Manipulation 27

Example 2.1 to 2.14

MODULE 3: ARRAYS AND GRAPHICS
3.1 Introduction to Matrices 28
3.1.1 Entering a matrix 28
3.1.2 Colon operator 28
3.1.3 Linear spacing 28
3.1.4 Colon operator in a matrix 28
3.1.5 Creating a sub-matrix 29
3.1.6 Deleting row or column 30
3.1.7 Dimension 30
3.1.8 Transposing a matrix 30
3.1.9 Concatenating matrices 30
3.1.10 Expanding a matrix 31
3.1.11 Matrix generators 31
3.1.12 Special matrices 32
3.1.13 Matrix arithmetic operations 32
3.1.14 Array arithmetic operations 32
3.1.15 Solving linear equations 33
3.2 Introduction to plot 33
3.2.1 Creating simple plots 33
3.2.2 Adding titles, axis labels, and annotations 34
3.2.3 Multiple data sets in one plot 34
3.2.4 Creating a bar graph 34
3.2.5 Stair step Plots 34
3.2.6 Polar Plots 35
3.2.7 Stem Plots 35
3.2.8 Scatter Plots 35
3.2.9 Change Line Appearance 35
3.3 Fractals and Chaos 36
3.4 Conway game of life 39
Example 3.1 to 3.24
MODULE 4: FILE HANDLING AND DEBUGGING
4.1 File operations 40
4.2 Save and Load workspace variables 41
4.2.1 Save Workspace Variables 41
4.2.2 Load Workspace variables 42
4.3 View Contents of MAT-File 42

4.4 Read the data from a text document 42

4.5 Read and write in excel sheet using matlab 42

4.5.1 Reading data from an excel sheet using matlab 42

4.5.2 Writing data into excel sheet using matlab 43

4.6 Debug a MATLAB program 45

4.6.1 Set Breakpoints 45

4.6.2 Run File 46

4.6.3 Pause a running file 46

4.6.4 Find and Fix a Problem 46

4.6.5 Modify Section of Code While Debugging 46

4.6.6 Step through File 46

4.6.7 End Debugging Session 48

4.6.8 Debugging Commands 48

Example 4.1 to 4.6

MODULE 1: INTRODUCTION

1.1: Basic Matlab program

Example 1.1: Write your first Matlab program

```
a = 3;
b = 5;
c = a+b
```

Output:

```
8
```

Example 1.2: The meaning of "a = b"

In Matlab and in any programming language, the statement "a = b" does not mean"a equals b". Instead, it prompts the action of *replacing the content of a by the content of b*.

```
a = 3;
b = a;
b
```

Output:

```
3
```

Example 1.3: Basic math operations

```
a = 3;
b = 9;
c = 2*a+b^2-a*b+b/a-10
```

Output:

```
53
```

Example 1.4: The meaning of "a = b", continued

```
a = 3;
a = a+1;
a
```

Output:

```
4
```

Example 1.5: Formatted output

```
fprintf('Hello')
```

Output:

```
Hello
```

Example 1.6: Formatted output

```
a = 3;
b = a*a;
c = a*a*a;
d = sqrt(a);
fprintf('%4u square equals %4u \r', a, b)
fprintf('%4u cube equals %4u \r', a, c)
fprintf('The square root of %2u is %6.4f \r', a, d)
```

Output:

```
   3 square equals    9
   3 cube equals   27
 The square root of  3 is 1.7321
```

Example 1.7: Arrays

```
a = [3  6  7];
b = [1  9  4];
c = a + b
```

Output:

```
4 15  11
```

Example 1.8: Extracting an individual element of an array

```
a = [3  6  7];
b = [1  9  4  5];
c = a(2) + b(4)
```

Output:

```
c = 11
```

Example 1.9: Continuation to next line

```
summation1 = 1 + 3 + 5 + 7 ...
 + 9 + 11
```

Note: The three periods (...) allow continuation to the next line of commands. The two lines in the above example are essentially one line of "summation1 = 1+3+5+7+9+11".

Example 1.10: "Clear" a variable

```
c1 = 3;
c2 = c1+5;
clear c1
c1
```

Output:

```
??? Undefined function or variable 'c1'.
```

Example 1.11: Intrinsic math functions and constants

```
x = pi;
y = sin(pi/2)
z = exp(-sin(pi/2))
```

Output:

y = 1
z = 0.3679

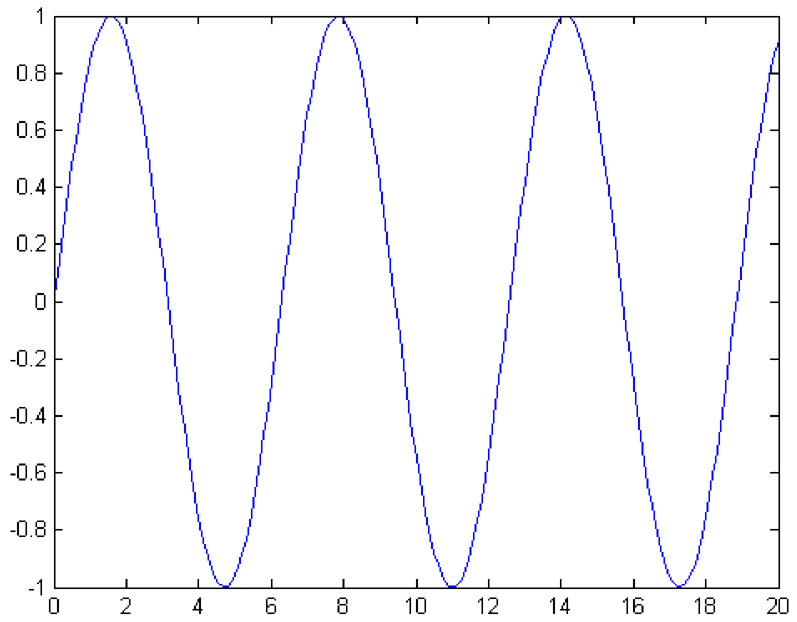
Example 1.12: Naming a variable

(i) Matlab variables are case sensitive. For example, "HILL" and "hill" are two differentvariables.
(ii) (ii) An underscore (_) or a number (0-9) can also be part of the name of a variable. For example, "Wood_12" is a legitimate variable name.
(iii) Some names are reserved for special constants. For example, "pi" is an intrinsic constant with a fixed value of 3.14159...

Example 1.13: Making a quick plot

```
x = [0:0.1:20];
y = sin(x);
plot(x,y)
```

The outcome will be the following plot:



1.2: Looping

1. The *for* loop

Example 1.14: Loop: Using **for** command

```
b = 3;
for k = 1:5
 b
end
```

Output:

```
3
3
3
3
3
```

Example 1.15: *For* loop:

```
b = 3;
for k = 1:5
 b^k
end
```

Output:

```
3
9
27
81
243
```

Example 1.16: *For* loop:

```
sum1 = 0;
for k = 1:9
  sum1 = sum1+k;
end
sum1
```

Output:

```
45
```

Remark: this program performs the summation of 1+2+3+4+5+6+7+8+9 (= 45).

Example 1.17: *For* loop:

```
sum1 = 0;
for k = 1:2:9
        sum1 = sum1+k;
end
sum1
```

Output:

```
25
```

Remark: this program performs the summation of 1+3+5+7+9 (= 25). The command "for k = 1:2:9" means we go through the loop only 5 times. First time with k = 1, secondtime with k = 1+2 (=3), third time with k = 1+2+2 (=5), and so on. The looping stops once k reaches 9.

Example 1.18: Treatment of array within a loop

```
b = [3 8  9  4  7  5];
sum1 = 0;
for k = 1:4
  sum1 = sum1+b(k);
end
sum1
```

Output:

24

Remark: This program performs the summation of sum1 = b(1)+b(2)+b(3)+b(4) =3+8+9+4 = 24

Example 1.19: Treatment of array within a loop

```
b = [3 8 9 4 7 5];
sum1 = 0;
for k = 1:2:5
  sum1 = sum1+b(k);
end
sum1
```

Output:

19

Remark: This program performs the summation of sum1 = b(1)+b(3)+b(5) = 3+9+7 = 19

Example 1.20: Double loop

```
sum1 = 0;
for n = 1:2
   for m = 1:3
      sum1 = sum1+n*m;
   end
end
sum1
```

Output:

18

Remark: this program performs the summation of Sum1 = 1*1+1*2+1*3 +2*1+2*2+2*3 = 18

Example 1.21: Double loop

```
for n = 1:2
   for m = 1:3
      fprintf('n = %3u m = %3u \r', n, m)
   end
end
```

Output:

```
   n = 1      m = 1
   n = 1      m = 2
   n = 1      m = 3
   n = 2      m = 1
   n = 2      m = 2
   n = 2      m = 3
```

Example 1.22: More complicated use of loop and index

```
b = [2 5  7  4  9  8  3];
c = [2 3  5  7];
sum1 = 0;
for k = 1:4
   sum1 = sum1+b(c(k));
end
sum1
```

Output:

```
24
```

Remark: This program performs the summation of

sum1 = b(c(1))+b(c(2))+b(c(3))+b(c(4))
= b(2)+b(3)+b(5)+b(7)
= 5+7+9+3
= 24

1.3: Branching

Example 1.23: The *if* command

```
num1 = 7;
if (num1 > 5)
    fprintf('%4u is greater than 5 \r', num1)
else
    fprintf('%4u is less than or equal to 5 \r', num1)
end
```

Output:

```
   7 is greater than 5
```

Same program, but change first line to "num1 = 3;"

Output:

```
   3 is less than or equal to 5
```

Remark: In this program, if (num1 > 5) (num1 is greater than 5) is true, the statement"fprintf('%4u is greater than 5 \r', num1)" is executed. Otherwise, the statement "fprintf('%4u is less than or equal to 5 \r', num1)" is executed.

Example 1.24: *if - elseif - else*

```
num1 = 4;
if (num1 >= 5)
    fprintf('%4i is greater than or equal to 5 \r', num1)
elseif (num1 > 1)
    fprintf('%4i is less than 5 but greater than 1 \r', num1)
elseif (num1 == 1)
    fprintf('%4i equals 1 \r', num1)
elseif (num1 > -3)
    fprintf('%4i is less than 1 but greater than -3 \r', num1)
else
    fprintf('%4i is less than or equal to -3 \r', num1)
end
```

Example 1.25: An application - determine whether a given year is a leap year (try to change thegiven value of n year and observe the outcome)

```
nyear = 1975;
if (mod(nyear, 400) == 0) fprintf('%6u is
   a leap year', nyear)
elseif (mod(nyear,4) == 0) & (mod(nyear,100) ~= 0)
   fprintf('%6u is a leap year', nyear)
else
   fprintf('%6u is not a leap year', nyear)
end
```

Output:

```
1975 is not a leap year
```

Remarks:

(1) In the *elseif* command (4th line), "&" means "AND". Both statements "(mod(nyaer,4) == 0)" and "(mod(nyear,100) ~= 0)" have to be true for Matlab to execute the command, "fprintf('%6u is a leap year', nyear)". Also commonly used in an*if* statement is "|" (a vertical line), which means "OR".

(2) The symbols "~=" in line 4 means "NOT EQUAL TO". There are 6 commonly usedexpressions to compare two numbers in an *if* command:

A > B	A is greater than B
A < B	A is less than B
A >= B	A is greater than or equal to B A

<= B	A is less than or equal to B
A == B	A equals B
A ~= B	A does not equal B

(3) The "mod(A,B)" function returns the remainder of A divided by B. For example, mod(7,2) = 1, mod(10,4) = 2, and mod(25,5) = 0. If A is divisible by B, mod(A,B) = 0.This is a very useful function in many applications related to numerical methods.

Example 1.26: Combine looping and branching

```
sum1 = 0;
sum2 = 0;
N = 9
for k = 1:N
   sum1 = sum1+k; if
   (mod(k,3) == 0)
      sum2 = sum2+k;
   end
end
sum1
sum2
```

Output:

sum1 = 45

sum2 = 18

Remark: Sum1 = 1+2+3+4+5+6+7+8+9, while sum2 = 3+6+9.

Example 1.27: The *while* loop

```
x = 3;
while (x < 100)x
   = x*3;
end
x
```

Output:

x = 243

Remark: One can think of a *while* loop as a combination of a *for* loop and an *if* statement. Here, the looping will keep going indefinitely as long as the condition, (x < 100), is satisfied. Therefore, the value of x progresses from 3, 9, 27, 81, to 243when the loop is terminated.

<u>1.4: Array and Matrix</u>

1. Assign the content of an array/matrix; Basic operations

Example 1.28: Assign the content of a (one-dimensional) array; Addition of two arrays

```
a = [2 12 25];
b = [3 7 4];
c = a+b
```

Output:

```
c = 5 19  29
```

Example 1.29: Assign the content of a matrix; Addition of two matricesa =

```
[3 4; 1 6];
b = [5 2; 11 7];
c = a+b
```

Output:

```
c =     8     6
       12    13
```

This program performs the following acts:

$$\boldsymbol{a} = \begin{bmatrix} 3 & 4 \\ 1 & 6 \end{bmatrix}$$

$$\boldsymbol{b} = \begin{bmatrix} 5 & 2 \\ 11 & 7 \end{bmatrix}$$

$$\boldsymbol{c} = \boldsymbol{a} + \boldsymbol{b} = \begin{bmatrix} 8 & 6 \\ 12 & 13 \end{bmatrix}$$

Example 1.30: Multiplication involving a scalar and an array (or a matrix)

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

Output:

```
b =    6    10
       2     8
```

Example 1.31: Element-by-element multiplication involving two 1-D arrays or two matrices ofthe same dimension

```
a = [2 3 5];
b = [2 4 9];
c = a.*b
```

Output:

```
c = 4     12    45
```

Remark: The period preceding the mathematical operation, "*", indicates that the operation will be performed element-by-element. In this case, the content of c is

```
c = [a(1)*b(1)  a(2)*b(2)  a(3)*b(3)]
```

Also, c is automatically assigned as a 1-D array with 3 elements

Example 1.32: Element-by-element multiplication of two matrices

```
a = [2 3; 1 4];
b = [5 1; 7 2];
c = a.*b
```

Output:
```
c = 10     3
     7     8
```

Example 1.33: Direct (not element-by-element) multiplication of two matrices

```
a = [2 3; 1 4];
b = [5 1; 7 2];
c = a*b
```

Output:
```
c = 31     8
    33     9
```
Remark: Observe how the outcome of this example differs from Ex. 32.

Example 1.34: Elementary functions with a vectorial variable
```
a = [2 3 5];
b = sin(a)
```
Output:
```
b = 0.9092      0.1411      -0.9589
```
Remark: The content of b is [sin(2) sin(3) sin(5)].

Example 1.35: Another example of elementary functions with a vectorial variable
```
a = [2 3 5];
b = 2*a.^2+3*a+4
```
Output:
```
b = 18 31  69
```
Remark: The content of b is
b = [2*(a(1))^2+3*a(1)+4   2*(a(2))^2+3*a(2)+4   2*(a(3))^2+3*a(3)+4].

Example 1.36: An efficient way to assign the content of an array

```
a = [0:0.5:4];
a
```
Output:
```
a = 0  0.5 1  1.5  2  2.5  3  3.5  4
```

Example 1.37: Extracting the individual element(s) of a matrix

```
A = [3 5; 2  4];
c = A(2,2)+A(1,2)
```

Output:
```
c = 9
```
Remark: With the given A matrix, we have A(1,1) = 3, A(1,2) = 5, A(2,1) = 2, andA(2,2) = 4.

Example 1.38: Another example for the usage of index for a matrix

```
A = [3 5; 2  4];
norm1   =   0;
for  m  =  1:2
for n = 1:2
   norm1 = norm1+A(m,n)^2;end
end
norm1 = sqrt(norm1)
```

Output:

```
norm1 = 7.348
```

Remark: This program calculates the Euclidean norm of the A matrix.

Example 1.39: Solving a system of linear equation

```
A = [4 1  2; 0  3  1; 0  1  2];
b = [17 ; 19 ; 13];
x = inv(A)*b
```

Output:

```
  x =  1
       5
       4
```

Remark: This program solves [**A**] **x** = **b**. The solution is obtained by **x** = [**A**]$^{-1}$ **b**.

Example 1.40: An alternative to Ex. 39

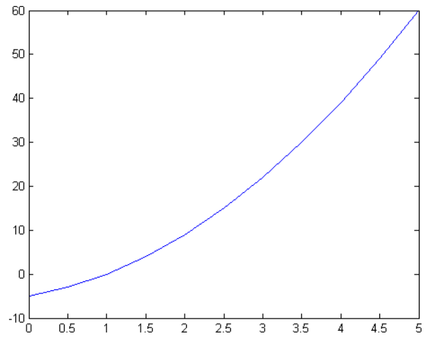
```
A = [4 1  2; 0  3  1; 0  1  2];
b = [17 ; 19 ; 13];
x = A\b
```

The output is the same as Ex. 39. Here, A\b is essentially inv(A)*b. (The "\" is called"back divide" in Matlab documentations.)

1.5: Graphic applications

Example 1.41: A quick example of plot command: Draw a curve

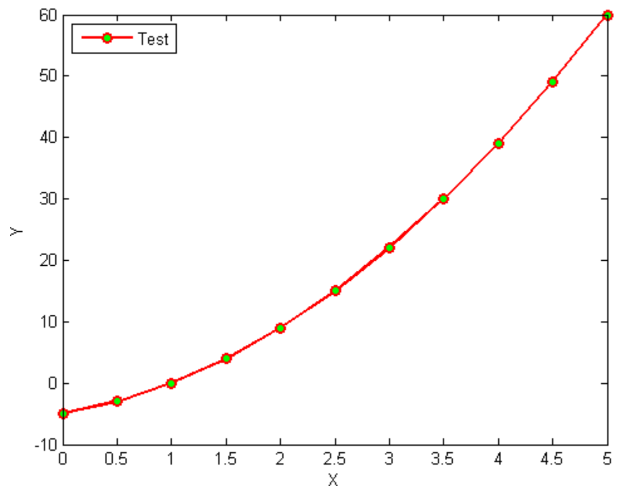
```
a = [0:0.5:5];
b = 2*a.^2 + 3*a -5;
plot(a,b)
```



Remarks:
(1) In "plot(a,b)", the array "a" should contain the data of the coordinate or "grid point)on the x-axis and "b" should be the corresponding values on the y-axis.
(2) After a plot is made, it can be further modified by using the interactive tool for graphics. For example, the labels of the x and y axes can be manually added to the plot.
(3) The plot can be saved in various formats (jpg, tif, eps, etc.).

Example 1.42: Refine the plot: Line pattern, color, and thickness

```
a = [0:0.5:5];
b = 2*a.^2 + 3*a -5;
plot(a,b,'-or','MarkerFaceColor','g','LineWidth',2)
xlabel('X'); ylabel('Y'); legend('Test','Location','NorthWest')
```

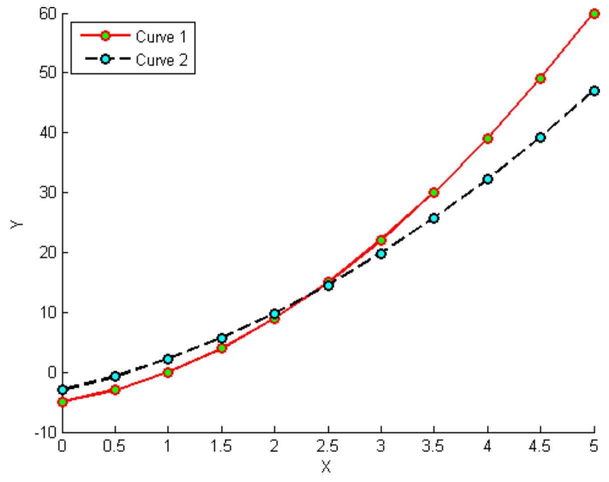


Remarks: The '-or' in the *plot* command set the line pattern. In this case, it's solid linewith circular symbol. The circular symbol is filled with green color ('g' for 'MarkerFaceColor'). The legend of the plot is set to locate at teh upper-left corner ('Location" set to 'NorthWest') inside the frame.

Example 1.43: Draw multiple curves

```
a = [0:0.5:5];
b = 2*a.^2 + 3*a -5;
c=1.2*a.^2+4*a-3;
hold on
plot(a,b,'-or','MarkerFaceColor','g','LineWidth',2)
plot(a,c,'--ok','MarkerFaceColor','c','LineWidth',2)
```

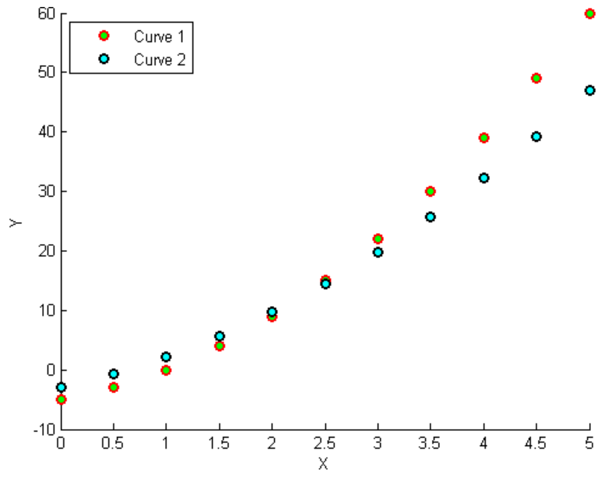
```
xlabel('X'); ylabel('Y'); legend('Curve 1','Curve 2','Location','NorthWest')
```



Remark: Without the "hold on" command, the second *plot* will override the first one and acts to erase the curve produced by the latter.

Example 1.44: Draw symbols

```
a = [0:0.5:5];
b = 2*a.^2 + 3*a -5;
c   =   1.2*a.^2+4*a-3;
hold on
plot(a,b,'or','MarkerFaceColor','g','LineWidth',2)
plot(a,c,'ok','MarkerFaceColor','c','LineWidth',2)
xlabel('X'); ylabel('Y'); legend('Curve 1','Curve 2','Location','NorthWest')
```



Example 1.45: Plot with multiple panels

```
a = [0:0.5:5];
b = 2*a.^2 + 3*a -5;
c=1.2*a.^2+4*a-3;
subplot(1,2,1)
```

```
plot(a,b,'-or','MarkerFaceColor','g','LineWidth',2)
xlabel('X'); ylabel('Y'); legend('Curve ','Location','NorthWest')
subplot(1,2,2)
plot(a,c,'--ok','MarkerFaceColor','c','LineWidth',2)
xlabel('X'); ylabel('Y'); legend('Curve 2','Location','NorthWest')
```

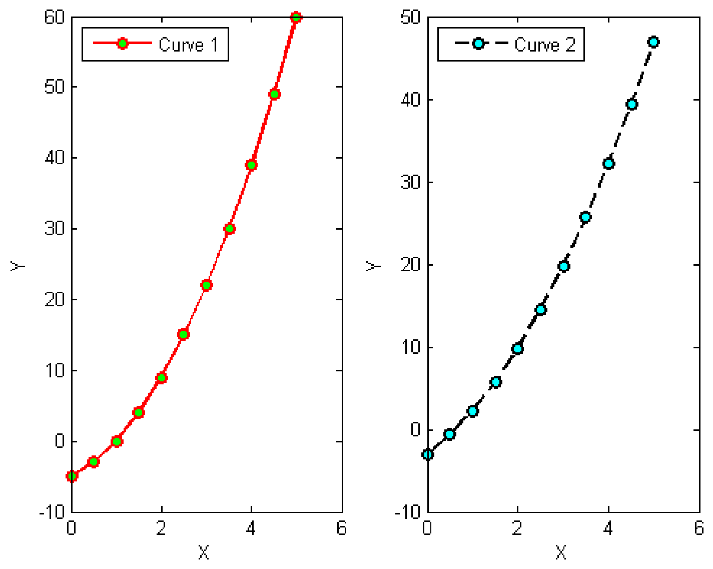


Table 1: Attributes for plot

SYMBOL	COLOR	SYMBOL	LINE STYLE	SYMBOL	MARKER
k	Black	–	Solid	+	Plus sign
r	Red	––	Dashed	o	Circle
b	Blue	:	Dotted	*	Asterisk
g	Green	–.	Dash-dot	.	Point
c	Cyan	none	No line	×	Cross
m	Magenta			*s*	Square
y	Yellow			*d*	Diamond

1.6: External (user-defined) function

Example 1.46: How to construct and use an external function

First, create the Matlab program for an external function as a separate file. The filenamehas to be identical to the name of the user-defined function

myfunc1.m

```
function outcome = myfunc1(x)
outcome = 2*x^2 + 3*x + 7;
```

This program defines a function, $f(x) \equiv 2\,x^2 + 3\,x + 7$. The "outcome" in the program isa dummy variable that is used to indicate the relation between the input parameter ("x")and the outcome after the evaluation of the function. After an external function is defined, it can be used in a main program that calls it. For example:

mainprog1.m

```
for n = 1:5

    x = n*0.1;
    z = myfunc1(x);
    fprintf('x = %4.2f  f(x) = %8.4f \r',x,z)
end
```

Output:

```
x = 0.10  f(x) =   7.3200
x = 0.20  f(x) =   7.6800
x = 0.30  f(x) =   8.0800
x = 0.40  f(x) =   8.5200
x = 0.50  f(x) =   9.0000
```

Note: The file for the external function should be stored in the same directory as the main program. Matlab will automatically find and link all required files for the externalfunctions.

Example 1.47: A function with multiple input parameters

myfunc2.m

```
function outcome = myfunc2(x,a,b,c)

outcome = a*x^2 + b*x + c;
```

This program defines a function, $f(x) \equiv a\,x^2 + b\,x + c$.

mainprog2.m

```
for n = 1:5
    x = n*0.1;

    z = myfunc2(x,2,3,7);

    fprintf('x = %4.2f  f(x) = %8.4f \r',x,z)
end
```

Output:

```
x = 0.10  f(x) =   7.3200
x = 0.20  f(x) =   7.6800
x = 0.30  f(x) =   8.0800
x = 0.40  f(x) =   8.5200
```

```
    x = 0.50   f(x) =   9.0000
```

1.7: Use external files and prompt for input and output

Example 1.48: Open a file and write the output to the file

```
fID1 = fopen('myoutput1.txt','w');
for n = 1:4
    b1 = n ; b2 = n^2 ; b3 = n^3;
    fprintf(fID1,'%7u %7u %7u \r',b1,b2,b3);
end
```

This program will produce no output on the screen but will create a file called "myoutput.txt" (under the same directory where the Matlab program itself is stored). Thecontent of the file is

```
      1       1       1
      2       4       8
      3       9      27
      4      16      64
```

Remarks:

(1) The first line of code opens a file, 'myoutput1.txt', with the permission type of "w" which means "write only". Upon executing the command, Matlab will pick a file ID (which is a number) and put it in the variable, fID1. A user does not need to know whatthis number is but will need to use it as the identifier if an output is to be written to that file.

(2) The command, fprintf(fID1,'%7u %7u %7u \r',b1,b2,b3), will direct the output to thefile with file ID given by fID1. If a file ID is not given, the shortened command fprintf('%7u %7u %7u \r',b1,b2,b3) will dump the output on screen which is the default destination of output.

Example 1.49: Read data from an existing file

```
fID1 = fopen('myoutput1.txt','r');
for n = 1:4
    b = fscanf(fID1,'%7u %7u %7u \r',3);
    btotal = b(1)+b(2)+b(3);
    fprintf('%7u + %7u + %7u = %7u \r', b(1), b(2), b(3), btotal)
end
```

Output:

```
      1 +       1 +       1 =       3
      2 +       4 +       8 =      14
      3 +       9 +      27 =      39
      4 +      16 +      64 =      84
```

Remark: We assume that the file "myoutput1.txt" has already been created by the Matlab program in Ex. 49 and properly stored in the same directory as the matlab code we are about to execute. The first command opens the file "myoutput1.txt" with the designated status of "r" which means "read only". The command "b = fscanf(fID1,'%7u
%7u %7u \r',3)" reads one record from the external file with file ID = fID1, using the format '%7u
%7u %7u \r'. The number "3" that follows indicates that only the first 3 elements in the record will be read and put into the array "b". (This is so the extra element, a "carriage return", will not be read.)

Example 1.50: Create a prompt to request user
```
inputnum1 = input('Enter your age');
if (num1 > 17)
   fprintf('You are eligible to vote')else
   fprintf('You are not eligible to vote')end
```

Remark: This is a special case when the input is read not from an external file but from a prompt. The output of this program will depend on whether the number that the user enters (which will be put into the variable "num1") is greater than 17. If the input is expected to be a string (such as "YES", "Joe Smith"), an "s" option has to be added to the command. For example, "name1 = input('Enter your surname', 's')".

MODULE 2: PROGRAMMING IN MATLAB

The commands entered in the command window cannot be saved and executed again for several times. Therefore, a different way of executing repeatedly commands with MATLAB is:

1. To *create* a file with a list of commands,
2. *Save* the file, and
3. *Run* the file.

If needed, corrections or changes can be made to the commands in the file. The files that are used for this purpose are called script files or scripts for short.

Here we discuss:

- M-File Scripts
- M-File Functions

2.1 M-File Scripts

A script file is an external file that contains a sequence of MATLAB statements. Script files have a filename extension *.m* and are often called M-files. M-files can be scripts that simply execute a series of MATLAB statements, or they can be functions that can accept arguments and can produce one or more outputs.

Example 2.1:

Consider the system of equations, Find the solution x to the system of equations.

$x + 2y + 3z = 1$

$3x + 3y + 4z = 1$

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

Solution:

- Use the MATLAB editor to create a file: **File → New → M-file.**
- Enter the following statements in the file:

```
A = [1 2 3; 3 3 4; 2 3 3];
b = [1; 1; 2];
x = A\b
```

- Save the file, for example, Example 2.1
- Run the file, in the command line, by typing:

```
>> Example 2.1
x =
-0.5000
1.5000
-0.5000
```

When execution completes, the variables (A, b, and x) remain in the workspace. To see a listing of them, enter whos at the command prompt.

There is another way to open the editor:

```
>> edit
```

or

```
>> edit filename.m
```

to open filename.m

Example 2.2:

Plot the following cosine functions, y1 = 2 cos(x), y2 = cos(x), and y3 = 0.5 ∗ cos(x), in the interval $0 \leq x \leq 2\pi$.

- Create a file, say example2.m, which contains the following commands:

```
>>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)')
>>title('Typical example of multiple plots')
>>axis([0 2*pi -3 3])
```

- Run the file by typing example2 in the Command Window.

2.1.1 Script side-effects

All variables created in a script file are added to the workspace. This may have undesirable effects, because:

- Variables already existing in the workspace may be overwritten.
- The execution of the script can be affected by the state variables in the workspace.

As a result, because scripts have some undesirable side-effects, it is better to code any complicated applications using rather function M-file.

2.2 M-File function

As mentioned earlier, functions are programs (or routines) that accept input arguments and return output arguments. Each M-file function (or function or M-file for short) has its own area of workspace, separated from the MATLAB base workspace.

2.2.1 Format of an M-File function

This simple function shows the basic parts of an M-file.

```
function f = factorial(n)                    (1)
% FACTORIAL(N) returns the factorial of N.   (2)
% Compute a factorial value.                 (3)
f = prod(1:n);                               (4)
```

The first line of a function M-file starts with the keyword function. It gives the function name and order of arguments. In the case of function factorial, there are up to one output argument and one input argument.

Example 2.3:

As an example, for n = 5, the result is,

```
>> f = factorial(5)
f =
120
```

The table summarizes the M- file function.

Part no.	M-file element	Description
(1)	Function definition line	Define the function name, and the number and order of input and output arguments
(2)	H1 line	A one line summary description of the program, displayed when you request Help
(3)	Help text	A more detailed description of the program
(4)	Function body	Program code that performs the actual computations

Both functions and scripts can have all of these parts, except for the function definition line which applies to function only.

In addition, it is important to note that function name must begin with a letter, and must be no longer than the maximum of 63 characters. Furthermore, the name of the text file that you save will consist of the function name with the extension .m. Thus, the above example file would be factorial.m.

Differences between scripts and functions

SCRIPTS	FUNCTIONS
Do not accept input arguments or return output arguments.	Can accept input arguments and return output arguments.
Store variables in a workspace that is shared with other scripts.	Store variables in a workspace internal to the function.
Are useful for automating a series of commands.	Are useful for extending the MATLAB language for your application.

2.2.2 Input and output arguments

The input arguments are listed inside parentheses following the function name. The output arguments are listed inside the brackets on the left side. They are used to transfer the output from the function file. The general form is

```
function [outputs] = function_name(inputs)
```

Function file can have none, one, or several output arguments.

Example of input and output arguments

function C=FtoC(F)	One input argument and one output argument
function area=TrapArea(a,b,h)	Three inputs and one output
function [h,d]=motion(v,angle)	Two inputs and two outputs

2.3 Input to a script file

When a script file is executed, the variables that are used in the calculations within the file must have assigned values. The assignment of a value to a variable can be done in three ways.

1. The variable is defined in the script file.
2. The variable is defined in the command prompt.

3. The variable is entered when the script is executed.

We have seen the first two cases let see the third case. The variable is defined in the script file. When the file is executed, the user is prompted to assign a value to the variable in the command prompt. This is done by using the input command.

Example 2.4:

This script file calculates the average of points scored in three games. The points from each game are assigned to a variable. Inputs are taken by using the ‘input’ command.

```
game1 = input(‘Enter the points scored in the first game’);
game2 = input(‘Enter the points scored in the second game’);
game3 = input(‘Enter the points scored in the third game’);
average = (game1+game2+game3)/3
```

Run this file script file in the command prompt (saved as example3)

```
>> example3
>> Enter the points scored in the first game 20
>> Enter the points scored in the second game 15
>> Enter the points scored in the third game 16
>>average = 17
```

2.4 Output commands

MATLAB automatically generates a display when commands are executed. In addition to this automatic display, MATLAB has several commands that can be used to generate displays or outputs.

Two commands that are frequently used to generate output are: disp and fprintf.

disp	:Simple to use. Provide limited control over the appearance of output.
fprintf	:Slightly more complicated than disp. Provide total control over the appearance of output.

2.5 Decision making statements

MATLAB has some decision making structures for control of command execution.

- the if statement
- the switch statement

When an "Algorithm" makes a choice to do one of two (or more things) this is called branching.

2.5.1 The if statement

MATLAB supports the variants of “if”, like

- if ... end
- if ... else ... end
- if ... elseif ... else ... end

The simplest form of the if statement is

```
if expression
   statements
end
```

Example 2.5:

1.
```
discr = b*b - 4*a*c;
if discr < 0
    disp('Warning: discriminant is negative, roots are imaginary');
end
```
2.
```
discr = b*b - 4*a*c;
if discr < 0
    disp('Warning: discriminant is negative, roots are imaginary');
else
    disp('Roots are real, but may be repeated')
end
```
3.
```
discr = b*b - 4*a*c;
if discr < 0
    disp('Warning: discriminant is negative, roots are imaginary');
elseif discr == 0
    disp('Discriminant is zero, roots are repeated')
else
    disp('Roots are real')
end
```

It should be noted that:

- elseif has no space between else and if (one word)
- no semicolon (;) is needed at the end of lines containing if, else, end
- indentation of if block is not required, but facilitate the reading.
- the end statement is required

2.5.2 The switch statement

The syntax of switch statement in MATLAB is:

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

Example 2.6: Create a script file

```
grade='B';
        Switch(grade)
        Case 'A'
                fprintf ('One!\n');
        Case 'B'
                fprintf('Two\n');
        Case 'C'
                fprintf('Three\n');
        otherwise
                fprintf('Four\n');
        end
```

Run this file, it displays –

```
Two
```

2.6 Loop control statements

With loop control statements, you can repeatedly execute a block of code. There are two types of loops:

- the for loop
- the while loop

2.6.1 The ''for...end'' loop

In for ... end loop, the execution of a command is repeated at a fixed and predetermined number of times. The syntax is

```
for variable = expression
    statements
end
```

Usually, expression is a vector of the form i:s:j.

Example 2.7:

```
for ii=1:5
   x=ii*ii
end
```

Multiple for loops can be nested. The following statements forms the 5-by-5 symmetric matrix A with (i,j) element i/j for j ≥i:

Example 2.8:

```
n = 5; A = eye(n);
for j=2:n
        for i=1:j-1
                A(i,j)=i/j;
                A(j,i)=i/j;
        end
end
```

2.6.2 The ‘‘while...end’’ loop

This loop is used when the number of passes is not specified. The looping continues until a stated condition is satisfied. The while loop has the form:

```
while expression
        statements
end
```

The statements are executed as long as expression is true.

```
x = 1
while x <= 10
        x = 3*x
end
```

It is important to note that if the condition inside the looping is not well defined, the looping will continue indefinitely. If this happens, we can stop the execution by pressing Ctrl-C.

2.6.3 Other flow structures

- The break statement. A while loop can be terminated with the break statement, which passes control to the first statement after the corresponding end. The break statement can also be used to exit the for loop.
- The continue statement can also be used to exit the for loop to pass immediately to the next iteration of the loop, skipping the remaining statements in the loop.
- Other control statements include return, continue, switch, etc.

2.7 Basic input/output (I/O)

Matlab automatically displays the value of a variable when you type its name at the command line.

Example 2.9:

```
>>x = 4.5;
>>x
 x =
          4.5
```

2.7.1 The disp function

Most basic console I/O can be implemented with two functions: disp and input.

The disp function can be used to display the values of variables without the extra "ans = " prefix.

Example 2.10:

```
>>[x,y]
ans =
          3.9      5.3
>>disp([x,y])
          3.9      5.3
```

2.7.2 The input function

The input() function can be used to prompt the user for input.

Example 2.11:

```
>>x = input('please enter the value of x :');
please enter the value of x : 4.3
>>disp(x)
4.3
```

We can input arrays

```
>>y = input('enter a 1-by-3 vector : ')
enter a 1-by-3 vector : [1,2,3]
y =
1         2         3
```

And strings

```
>>fname = input('output file : ')
output file : 'test.txt'
fname =
          test.txt
```

2.7.3 The save and load functions

The save function is used to save current workspace variables to a specified file. The load function is used to load variables from a specified file to the workspace.

Example 2.12: Let workspace variables are x, y and z.

Commands	Description
>>save test	All variables from current workspace are saved to the file test. The file test will be created in the working directory with a *.mat extension. The current variables x, y, z are all saved to the file test.
>>save xtest x,y	This saves only variable x and y from current workspace to the file xtest.
>>clear all	Workspace cleared of all variables.
>>load test	All variables from the file test are loaded back to the workspace i.e. x, y, z
>>clear all	Workspace cleared of all variables.
>>load test x,y	Specific variables x, y from the file test are loaded back to the workspace.
>>clear all	Workspace cleared of all variables.
>>load xtest	All variables from the file xtest are loaded to the workspace i.e. x,y

2.8 Formatted I/O

Matlab implement versions of the C fprintf and sprintf functions for formatted output.

Fprintf	File PRINT Formatted	Use this for printing text to a file or the command window.
Sprint	String PRINT Formatted	Use this for creating a MATLAB string (character vector).

sprintf works same as fprintf except that it returns its formatted string instead of printing it to the command window.

2.8.1 fprintf

Example 2.13:

```
>>fprintf('Hello')
Hello>>fprintf('Hello\n')      % Formats a string and writes it to the command window even when there is
                               a semicolon at the end(k>>).
Hello
>>fprintf('Hello, my name'' John\n')
Hello my name's John           % In workspace, John is the current value of the string variable name
>>fprintf('Hello, my name''s %s\n', name)
Hello, my name's John
>>name='Pinto'
name=
   Pinto
>>>>fprintf('Hello, my name''s %s\n', name)
Hello, my name's Pinto
>>x=10; y=20; z=30;
>>fprintf('Hello, my name''s %s, and my test average is % 0.1f\n, name,(x+y+z)/3)
Hello, my name's Pinto, and my test average is 20.0
>>fprintf('Hello, my name''s %s, and my test average is % 0.2f\n, name,(x+y+z)/3)
Hello, my name's Pinto, and my test average is 20.00
```

- Special escape sequences, beginning with the backslash '\' character, are used to represent "non-printable" characters. The most commonly used escape sequences are:

\n	a "new line" character (start a new line of output)
\t	a "tab" -- spaces over to the next tab stop
\r	a "carriage return"
\\	output a single backslash

- The most commonly used data types are:

%d	display the value as an integer (this only works if the number really is an integer)
%e	display the value in exponential format (3.23e5)
%f	display the value in floating point format (3.451)
%g	display the value showing only significant digits -- use this most of the time
%s	display the value as a string - a series of UNICODE characters
%c	display the value as a single UNICODE character

2.8.2 sprintf

Example 2.14:

```
>>a=sprintf('Train');
```

```
>>b= sprintf('departure,');
>>c=10;
>>d= sprintf('is at %d.\n',c);
>>disp([a b d])
>> Train departure, is at 10.
```

2.9 File and Folder Manipulation

MATLAB has several commands that manipulate entire files, such as:

copyfile('source', 'destination')	Copies the entire contents of one file into another file.
movefile('source','destination')	Moves a file from one folder to a different folder.
delete('FileName')	Deletes a file.

MATLAB has several commands that manipulate folders, such as:

s = pwd()	Return the name of the "present working directory."
cd('FolderName')	Makes the specified folder the "present working directory."
files = dir('FolderName')	Returns information about all the files in the specified folder.
mkdir('FolderName')	Make a new sub-folder in the "present working directory."
rmdir('FolderName','s')	Remove (delete) a folder and all the files in the folder.

MODULE 3: ARRAYS AND GRAPHICS

3.1 Introduction to Matrices

3.1.1 Entering a matrix

Example 3.1:

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

Single elements of a matrix are accessed as A(i,j), where $i \geq 1$ and $j \geq 1$. Zero or negative subscripts are not supported in MATLAB.

3.1.2 Colon operator

```
>> x = 0:1:10;
```

The row vector has 11 elements.

3.1.3 Linear spacing

```
y = linspace(a,b)
```

generates a row vector y of 100 points linearly spaced between and including a and b.

```
y = linspace(a,b,n)
```

generates a row vector y of n points linearly spaced between and including a and b. This is useful when we want to divide an interval into a number of subintervals of the same length.

Example 3.2:

```
>> theta = linspace(0,2*pi,101)
```

divides the interval $[0, 2\pi]$ into 100 equal subintervals, then creating a vector of 101 elements.

3.1.4 Colon operator in a matrix

Example 3.3:

```
>> A(2,:)
ans =
4 5 6
```

is the second row elements of A

```
>> A(:,2:3)
 ans =
2 3
5 6
8 0
```

A(:,2:3) is a sub-matrix with the last two columns of A.

```
>> A(:,2)=[]
ans =
1 3
4 6
7 0
```

3.1.5 Creating a sub-matrix

To extract a submatrix B consisting of rows 2 and 3 and columns 1 and 2 of the matrix A, do the following.

Example 3.4:

```
>> B = A([2 3],[1 2])
B =
4 5
7 8
```

To interchange rows 1 and 2 of A, use the vector of row indices together with the colon operator.

```
>> C = A([2 1 3],:)
C =
4 5 6
1 2 3
7 8 0
```

It is important to note that the colon operator (:) stands for all columns or all rows. To create a vector version of matrix A, do the following

```
>> A(:)
ans =
1
2
3
4
5
6
7
8
0
```

The submatrix comprising the intersection of rows p to q and columns r to s is denoted by A(p:q,r:s).
As a special case, a colon (:) as the row or column specifier covers all entries in that row or column; thus

- A(:,j) is the jth column of A, while
- A(i,:) is the ith row, and
- A(end,:) picks out the last row of A.

The keyword end, used in A(end,:), denotes the last index in the specified dimension.

```
>> A
A =
1 2 3
4 5 6
7 8 9
>> A(2:3,2:3)
ans =
5 6
8 9
>> A(end:-1:1,end)
ans =
9
6
3
>> A([1 3],[2 3])
ans =
2 3
8 9
```

3.1.6 Deleting row or column

To delete a row or column of a matrix, use the empty vector operator, [ ].

Example 3.5:

```
>> A(3,:) = []
A =
1 2 3
4 5 6
```

Third row of matrix A is now deleted. To restore the third row, we use a technique for creating a matrix.

```
>> A = [A(1,:);A(2,:);[7 8 0]]
A =
1 2 3
4 5 6
7 8 0
```

Matrix A is now restored to its original form.

3.1.7 Dimension

To determine the dimensions of a matrix or vector, use the command size.

Example 3.6:

```
>> size(A)
ans =
3 3
```

means 3 rows and 3 columns.

Or more explicitly with,

```
>> [m,n]=size(A)
```

3.1.8 Transposing a matrix

```
>> A’
ans =
1 4 7
2 5 8
3 6 0
```

3.1.9 Concatenating matrices

Matrices can be made up of sub-matrices.

Example 3.7:

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

The new matrix B will be,

```
>> B = [A 10*A; -A [1 0 0; 0 1 0; 0 0 1]]
B =
1 2 3 10 20 30
4 5 6 40 50 60
7 8 9 70 80 90
-1 -2 -3 1 0 0
-4 -5 -6 0 1 0
-7 -8 -9 0 0 1
```

3.1.10 Expanding a matrix

We can add one or more elements to a matrix by placing them outside of the existing row and column index boundaries. MATLAB automatically pads the matrix with zeros to keep it rectangular.

Example 3.8:

```
>>A=[1      2      3;6     7      8]
>>A
A=
        1      2      3
        6      7      8
>>A(3,4)=11
>>A
A=
        1      2      3      0
        6      7      8      0
        0      0      0      11
```

We can also expand the size by inserting a new matrix outside of the existing index ranges.

```
>>A(4:5,5:6)=[10     20;30   40]
>>A
A=
        1      2      3      0      0      0
        6      7      8      0      0      0
        0      0      0      11     0      0
        0      0      0      0      10     20
        0      0      0      0      30     40
```

3.1.11 Matrix generators

MATLAB provides a function that generates elementary matrices.

Elementary matrices

eye(m,n)	Returns an m-by-n matrix with 1 on the main diagonal
eye(n)	Returns an n-by-n square identity matrix
zeros(m,n)	Returns an m-by-n matrix of zeros
ones(m,n)	Returns an m-by-n matrix of ones
diag(A)	Extracts the diagonal of matrix A
rand(m,n)	Returns an m-by-n matrix of random numbers

For a complete list of elementary matrices and matrix manipulations, type help elmat or doc elmat.

1.
```
>> b=ones(3,1)
b =
1
1
1
>> b=[1;1;1]
```
2.
```
>> eye(3)
ans =
1 0 0
0 1 0
0 0 1
```
3.
```
>> c=zeros(2,3)
c =
0 0 0
0 0 0
```
4.
```
>> D = [C zeros(2); ones(2) eye(2)]
D =
1 2 0 0
3 4 0 0
1 1 1 0
1 1 0 1
```

3.1.12 Special matrices

hilb	Hilbert matrix	magic	Magic square
invhilb	Inverse Hilbert matrix	pascal	Pascal matrix
toeplitz	Toeplitz matrix	vander	Vandermonde matrix

3.1.13 Matrix arithmetic operations

A+B or B+A	is valid if A and B are of the same size
A*B	is valid if A's number of column equals B's number of rows
A^2	is valid if A is square and equals A*A
α*A or A*α	multiplies each element of A by α

3.1.14 Array arithmetic operations

.*	Element-by-element multiplication
./	Element-by-element division
.^	Element-by-element exponentiation

```
>> C = A.*B
>> A.^2
```

Summary of matrix and array operations

Operation	**Matrix**	**Array**
Addition	+	+
Subtraction	-	-
Multiplication	*	*
Division	/	./
Left division	\	.\
Exponentiation	^	.^

3.1.15 Solving linear equations

Example 3.9:

Consider the following system of linear equations

x + 2y + 3z = 1

4x + 5y + 6z = 1

7x + 8y = 1

The coefficient matrix A is

A= $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ and the vector b= $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$

With matrix notation, a system of simultaneous linear equations is written Ax = b

This equation can be solved for x using linear algebra. The result is x = A−1 b.

There are typically two ways to solve for x in MATLAB:

1.
```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
 >> x = inv(A)*b
x =
-1.0000
1.0000
-0.0000
```

2.
```
>> A = [1 2 3; 4 5 6; 7 8 0];
>> b = [1; 1; 1];
>> x = A\b
x =
-1.0000
1.0000
-0.0000
```

3.2 Introduction to plot

3.2.1 Creating simple plots

Example 3.10:

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

Example 3.11:

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

3.2.2 Adding titles, axis labels, and annotations

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

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').

3.2.3 Multiple data sets in one plot

Multiple (x, y) pairs arguments create multiple graphs with a single call to plot.

Example 3.12:

These statements plot three related functions of x: y1 = 2cos(x), y2 = cos(x), and y3 = 0.5* cos(x), in the interval $0 \leq x \leq 2\pi$.

```
>> 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)')
>> title('Typical example of multiple plots')
>> axis([0 2*pi -3 3])
```

3.2.4 Creating a bar graph

bar(y) creates a bar graph with one bar for each element in y. If y is an *m*-by-*n* matrix, then bar creates *m* groups of *n* bars.

bar(x,y) draws the bars at the locations specified by x.

Example 3.13:

```
>>y=[10     15     20     25     30     45     60]
>>bar(y)
```

3.2.5 Stair step Plots

The stairs function creates a stairstep plot. It can create a stairstep plot of Y values only or a stairstep plot of x and y values.

Example 3.14:

```
>>x = 0:0.25:10;
>>y = sin(x);
>>stairs(x,y)
```

3.2.6 Polar Plots

The polarplot function draws a polar plot of the angle values in theta (in radians) versus the radius values in rho.

Example 3.15:

```
>>theta = 0:0.01:2*pi;                          % angle
>>rho = abs(sin(2*theta).*cos(2*theta));         % radius
>>polarplot(theta,rho)
```

3.2.7 Stem Plots

The stem function draws a marker for each x and y value with a vertical line connected to a common baseline.

Example 3.16:

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

3.2.8 Scatter Plots

The scatter function draws a scatter plot of x and y values.

Example 3.17:

```
>>load patients Height Weight Systolic          % load data
>>scatter(Height,Weight)                         % scatter plot of Weight vs. Height
>>xlabel('Height')
>>ylabel('Weight')
```

3.2.9 Change Line Appearance

We can change the line color, line style, or add markers by including an optional line specification when calling the plot function. For example:

- ':' plots a dotted line.
- 'g:' plots a green, dotted line.
- 'g:*' plots a green, dotted line with star markers.
- '*' plots star markers with no line.

Example 3.18:

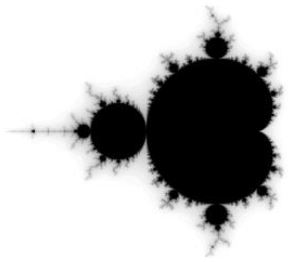
Plot a dotted line. Add a second plot that uses a dashed, red line with circle markers.

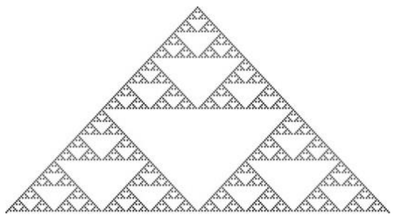
```
>>x = linspace(0,2*pi,50);
>>y = sin(x);
>>plot(x,y,':')
>>hold on
>>y2 = cos(x);
>>plot(x,y2,'--ro')
>>hold off
```

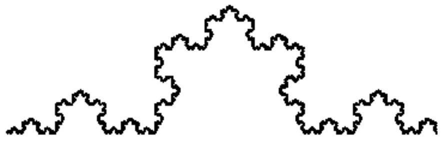
3.3 Fractals and Chaos

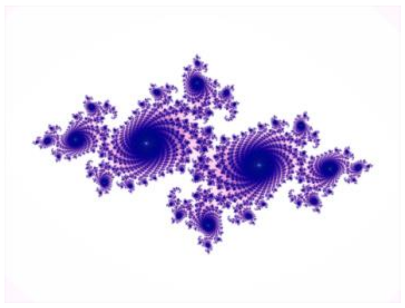
The term ‘fractal’ was coined in 1975 by Benoit Mandelbrot from the Latin fractus, meaning “broken” or “irregular.” This term was used to describe shapes that have the characteristic of self-similarity, i.e. that when you magnify any part it looks just like (or has the same structure) as the original. He is widely known for the Mandelbrot set.

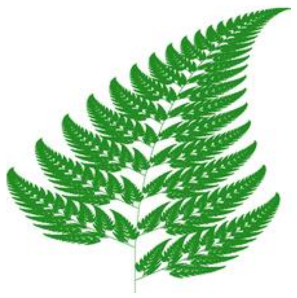
Some other fractals are The Sierpinski triangle, The Koch curve, The Julia set, The Barnsley fern. Trees, clouds, snowflakes, mountains, and rivers are some other naturally occurring fractals. The Chaos Game is a very fast way to generate fractals.


The Mandelbrot set


The Sierpinski Triangle


The Koch curve


The Julia set


The Barnsley fern

Example 3.19: Mandelbrot set 3D

```
col=20;
m=200;
cx=-.6;
cy=0;
l=1.5;
x=linspace(cx-l,cx+l,m);
y=linspace(cy-l,cy+l,m);
[X,Y]=meshgrid(x,y);
Z=zeros(m);
C=X+i*Y;
for k=1:col;
Z=Z.^2+C;
W=-abs(Z);
A = ( W <-.5);
B = 1-A;
V = abs(C.*C);
D = B.*sin(-0.2*V);
end
E = D-1;
colormap copper(256);
mesh(0.1*A+E);
shading flat;
axis([0 130 0 130 -2 0])
axis('square','equal','off');
```

Example 3.20: Mandelbrot sets f(z)=z^2+c

```
col=20;
m=400;
cx=-.6;
cy=0;
l=1.5;
x=linspace(cx-l,cx+l,m);
y=linspace(cy-l,cy+l,m);
[X,Y]=meshgrid(x,y);
Z=zeros(m);
C=X+i*Y;
for k=1:col;
Z=Z.^2+C;
W=exp(-abs(Z));
end
colormap copper(256);
pcolor(W);
shading flat;
axis('square','equal','off');
```

Example 3.21: Mandelbrot sets f(z)=z^3+c	**Example 3.22: Mandelbrot sets f(z)=z^{50}+c**

col=20;
m=400;
cx=-.6;
cy=0;
l=1.5;
x=linspace(cx-l,cx+l,m);
y=linspace(cy-l,cy+l,m);
[X,Y]=meshgrid(x,y);
Z=zeros(m);
C=X+i*Y;
for k=1:col;
Z=Z.^3+C;
W=exp(-abs(Z));
end
colormap copper(256);
pcolor(W);
shading flat;
axis('square','equal','off');
	col=20;
	m=400;
	cx=-.2;
	cy=0;
	l=1.5;
	x=linspace(cx-l,cx+l,m);
	y=linspace(cy-l,cy+l,m);
	[X,Y]=meshgrid(x,y);
	Z=zeros(m);
	C=X+i*Y;
	for k=1:col;
	Z=Z.^50+C;
	W=exp(-abs(Z));
	end
	colormap copper(256);
	pcolor(W);
	shading flat;
	axis('square','equal','off');

Example 3.23: Sierpinski triangle

```
px=[1,0,2];
py=[sqrt(3),0,0];
pinitx(1)=0;
pinity(1)=0;
for i=2:5000
N=1;
        a=1;
        b=3;
        random=a+(b-(a-1))*rand(1,N);
        dice=floor(random);
if dice ==1
        pinitx(i)=(pinitx(i-1)+px(1))/2;
        pinity(i)=(pinity(i-1)+py(1))/2;
elseif dice==2
pinitx(i)=(pinitx(i-1)+px(2))/2;
        pinity(i)=(pinity(i-1)+py(2))/2;
elseif dice==3
        pinitx(i)=(pinitx(i-1)+px(3))/2;
                pinity(i)=(pinity(i-1)+py(3))/2;
end
plot(px,py,'k.',pinitx,pinity,'k.')
pause(1e-100)
```

```
end
title('THE CHAOS GAME: SIERPINSKI TRIANGLE')
```

3.4 Conway game of life

It is a famous cellular automata. The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Example 3.24:

```
len=50; GRID=int8(rand(len,len));
up=[2:len 1]; down=[len 1:len-1]; %the world is round
colormap(gray(2));
for i=1:100
   neighbours=GRID(up,:)+GRID(down,:)+GRID(:,up)+GRID(:,down)+...
   GRID(up,up)+GRID(up,down)+GRID(down,up)+GRID(down,down);
   GRID = neighbours==3 | GRID & neighbours==2;
   image(GRID*2); pause(0.02);
end
```

MODULE 4: FILE HANDLING AND DEBUGGING

4.1 File operations

Find, view and change files and folders.

Functions

dir	List folder contents
ls	List folder contents
pwd	Identify current folder
fileattrib	Set or get attributes of file or folder
exist	Check existence of variable, script, function, folder, or class
isfile	Determine if input is file
isfolder	Determine if input is folder
type	Display contents of file
visdiff	Compare two files or folders
what	List MATLAB files in folder
which	Locate functions and files
cd	Change current folder
copyfile	Copy file or folder
delete	Delete files or objects
recycle	Set option to move deleted files to recycle folder
mkdir	Make new folder
movefile	Move or rename file or folder
rmdir	Remove folder
open	Open file in appropriate application
winopen	Open file in appropriate application (Windows)
Current Folder Browser	Open Current Folder browser

Example 4.1:

To create a new folder: Use the mkdir function.

Create a subfolder named newdir in a parent folder named parentFolder:

```
mkdir('parentFolder','newdir');
```

To move a file or folder: Use the movefile function.

Move the file named myfile.m in the current folder to the folder, d:/work:

```
movefile('myfile.m','d:/work');
```

To rename a file or folder: Use the movefile function.

In the current folder, rename myfile.m to oldfile.m:

```
movefile('myfile.m','oldfile.m');
```

To Delete a file or folder: Use the delete function.

Delete a file named myfile.m in the current folder:

```
delete('myfile.m');
```

To read data from text file: Use fscanf

Syntax

A = fscanf(fileID,formatSpec)

A = fscanf(fileID,formatSpec,sizeA)

[A,count] = fscanf(___)

Description

A = fscanf(fileID,formatSpec) reads data from an open text file into column vector A and interprets values in the file according to the format specified by formatSpec. The fscanf function reapplies the format throughout the entire file and positions the file pointer at the end-of-file marker. If fscanf cannot match formatSpec to the data, it reads only the portion that matches and stops processing.

The text file is indicated by the file identifier, fileID. Use fopen to open the file, specify the character encoding, and obtain the fileID value. When you finish reading, close the file by calling fclose(fileID).

A = fscanf(fileID,formatSpec,sizeA) reads file data into an array, A, with dimensions, sizeA, and positions the file pointer after the last value read. fscanf populates A in column order. sizeA must be a positive integer or have the form [m n], where m and n are positive integers.

[A,count] = fscanf(___) additionally returns the number of fields that fscanf reads into A. For numeric data, this is the number of values read. You can use this syntax with any of the input arguments of the previous syntaxes.

4.2 Save and Load workspace variables

The workspace is not maintained across sessions of MATLAB. When you quit MATLAB, the workspace clears. However, you can save any or all the variables in the current workspace to a MAT-file (.mat). You can then reuse the workspace variables later during the current MATLAB session or during another session by loading the saved MAT-file.

4.2.1 Save Workspace Variables

There are several ways to save workspace variables:

- To save all workspace variables to a MAT-file, on the **Home** tab, in the **Variable** section, click **Save Workspace**.
- To save a subset of your workspace variables to a MAT-file, select the variables in the Workspace browser, right-click, and then select **Save As**. You also can drag the selected variables from the Workspace browser to the Current Folder browser.
- To save variables to a MATLAB script, click the **Save Workspace** button or select the **Save As** option, and in the **Save As** window, set the **Save as type** option to **MATLAB Script**. Variables that cannot be saved to a script are saved to a MAT-file with the same name as that of the script.

By using save function

Example 4.2:

To save all current workspace variables to the file june10.mat, use the command

```
>>save('june10')
```

To save only variables A and B to the file june10.mat, use the command

```
>>save('june10','A','B')
```

4.2.2 Load Workspace variables

To load saved variables from a MAT-file into your workspace, double-click the MAT-file in the Current Folder browser.

To load a subset of variables from a MAT-file on the **Home** tab, in the **Variable** section, click **Import Data**. Select the MAT-file you want to load and click **Open**. You also can drag the desired variables from the Current Folder browser Details panel of the selected MAT-file to the Workspace browser.

To load variables saved to a MATLAB script into the workspace, simply run the script.

By using load function

Example 4.3:

Load all variables from the file ABC.mat

```
>>load('ABC')
```

To load variables X and map from the file ABC.mat

```
>>load('ABC','X','map')
```

4.3 View Contents of MAT-File

Use the command whos -file filename. This function returns the name, dimensions, size, and class of all variables in the specified MAT-file. For example, you can view the contents of the example file ABC.mat.

```
>>whos -file ABC.mat
```

4.4 Read the data from a text document

Example 4.4:

Suppose we have a text document abc.txt in the current folder, data in the folder is
Hi how are you? to read this data we have:

```
clear all; close all; clc;

fid=fopen('abc.txt');
text=fread(fid, 'uint8=>char')';        % Without transpose data shown as a column vector
fclose(fid);

msgbox(text);                           % Shows the data in the message box on the screen
```

Run

```
>> text
text=
Hi how are you?
```

4.5 Read and write in excel sheet using matlab

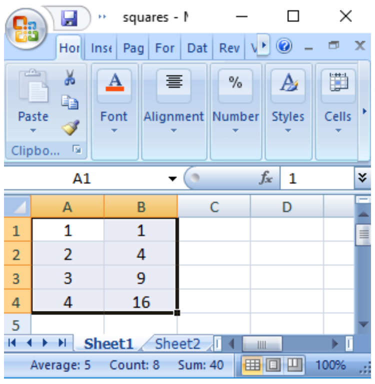
Two built-in function for this

- xlsread
- xlswrite

4.5.1 Reading data from an excel sheet using matlab

Example 4.5:

Create an excel file and save it as “squares” the content is



	A	B	C	D
1	1	1		
2	2	4		
3	3	9		
4	4	16		
5				

```
>>data=xlsread(‘squares.xlsx’)
data =
1           1
2           4
3           9
4           16
```

We perform different operation from the data of this file.

```
>>firstC=data( :,1)
firstC=
            1
            2
            3
            4
>>secondC=data( :,2)
secondC=
            1
            4
            9
            16
>>plot(secondC,firstC)
```

4.5.2 Writing data into excel sheet using matlab

Example 4.6:

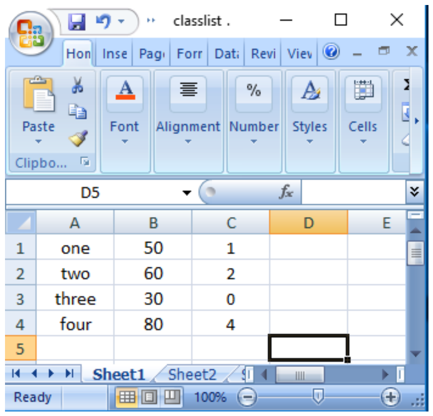
```
>>classinfo={one,50,1; two,60,2; three,30,0; four,80,4}
classinfo=
one         50          1
two         60          2
three       30          0
four        80          4
```

- Any 2D array can be written into an excel file. Here a cell array classinfo is defined and written into the file classlist.
- Also shown is how to specify appropriate worksheets and/or ranges in the excel file.
- Define a cell array classinfo.
- Use matlab command xlswrite to write classinfo into an excel file classlist.

```
>>xlswrite(‘classlist’,classinfo)
```

classlist is the first argument and is defined as a string.

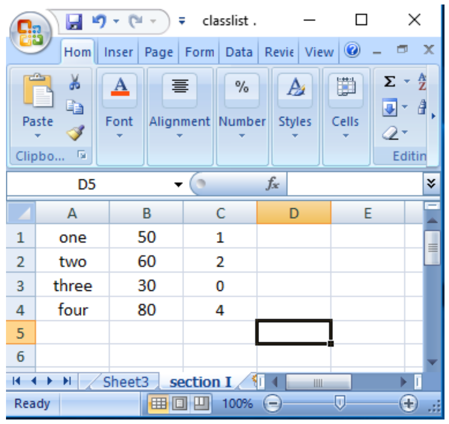
Second argument is the array containing the data to be written.
The excel file created in the current working folder if it does not already exist. If it already exists it will be modified accordingly.


classlist .
Hon Inse Pag Forr Dat Revi Viev
Paste Font Alignment Number Styles Cells
Clipbo...
D5
Sheet1 Sheet2
Ready 100%

	A	B	C	D	E
1	one	50	1		
2	two	60	2		
3	three	30	0		
4	four	80	4		
5					

To write classinfo into an excel file classlist in the worksheet section I:

>>xlswrite(‘classlist’,classinfo,’section I’)


classlist .
Hom Inser Page Form Data Revie View
Paste Font Alignment Number Styles Cells
Clipbo... Editin
D5
Sheet3 section I
Ready 100%

	A	B	C	D	E
1	one	50	1		
2	two	60	2		
3	three	30	0		
4	four	80	4		
5					
6					

The name of the worksheet is defined as a string and is the third argument to xlswrite.

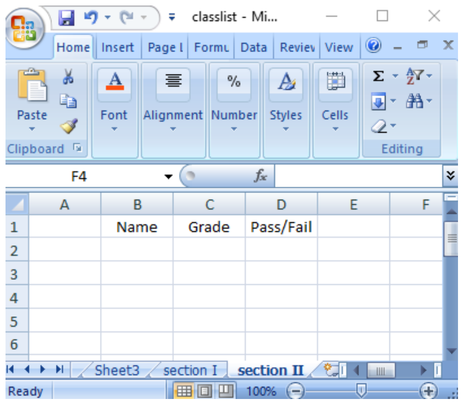
Define a cell array header that will serve as the header for the worksheet.

>>header(‘’,‘Name’,‘Grade’,‘Pass/Fail’)

header=

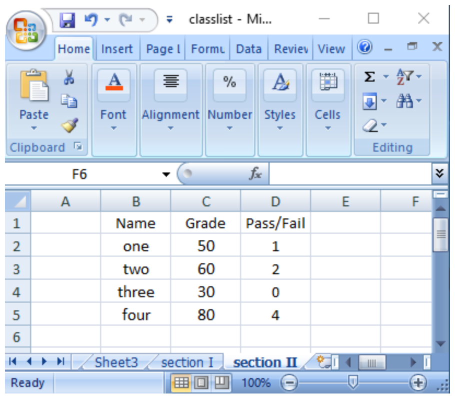
‘’ ‘Name’ ‘Grade’ ‘Pass/Fail’

>>xlswrite(‘classlist,header,’section II’)


classlist - Mi...
Home Insert Page l Formu Data Reviev View
Paste Font Alignment Number Styles Cells
Clipboard Editing
F4
Sheet3 section I section II
Ready 100%

	A	B	C	D	E	F
1		Name	Grade	Pass/Fail		
2						
3						
4						
5						
6						

>>xlswrite(‘classlist’,classinfo,’section II’,‘B2’)



	A	B	C	D	E	F
1		Name	Grade	Pass/Fail		
2		one	50	1		
3		two	60	2		
4		three	30	0		
5		four	80	4		
6						

4.6 Debug a MATLAB program

To debug your MATLAB program graphically, use the Editor/Debugger. Alternatively, you can use debugging functions in the Command Window. Both methods are interchangeable.

Before you begin debugging, make sure that your program is saved and that the program and any files it calls exist on your search path or in the current folder.

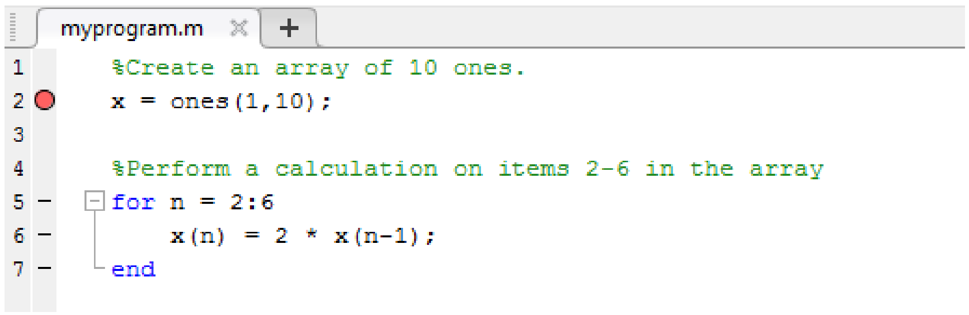
- If you run a file with unsaved changes from within the Editor, then the file is automatically saved before it runs.
- If you run a file with unsaved changes from the Command Window, then MATLAB software runs the saved version of the file. Therefore, you do not see the results of your changes.

4.6.1 Set Breakpoints

Set breakpoints to pause the execution of a MATLAB file so you can examine the value or variables where you think a problem could be. You can set breakpoints using the Editor, using functions in the Command Window, or both.

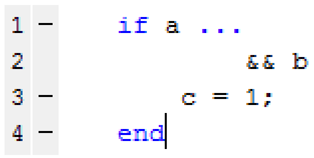
There are three different types of breakpoints: standard, conditional, and error. To add a *standard* breakpoint in the Editor, click the breakpoint alley at an executable line where you want to set the breakpoint. The *breakpoint alley* is the narrow column on the left side of the Editor, to the right of the line number. You also can use the **F12** key to set the breakpoint.

Executable lines are indicated by a dash (—) in the breakpoint alley. For example, click the breakpoint alley next to line 2 in the code below to add a breakpoint at that line.



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

If an executable statement spans multiple lines, you can set a breakpoint at each line in that statement, even though the additional lines do not have a — (dash) in the breakpoint alley. For example, in this code. you can set a breakpoint at all four lines:

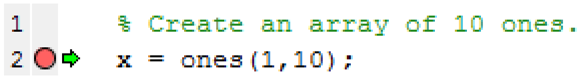


```
if a ...
        && b
    c = 1;
end
```

4.6.2 Run File

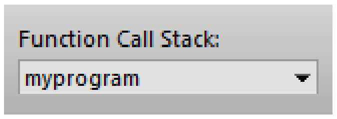
After setting breakpoints, run the file from the Command Window or the Editor. Running the file produces these results:

- The **Run** ▷ button changes to a **Pause** ⏸ button.
- The prompt in the Command Window changes to K>> indicating that MATLAB is in debug mode and that the keyboard is in control.
- MATLAB pauses at the first breakpoint in the program. In the Editor, a green arrow just to the right of the breakpoint indicates the pause. The program does not execute the line where the pause occurs until it resumes running. For example, here the debugger pauses before the program executes x = ones(1,10);.



```
% Create an array of 10 ones.
x = ones(1,10);
```

- MATLAB displays the current workspace in the **Function Call Stack**, on the **Editor** tab in the **Debug** section.



Function Call Stack:

myprogram

If you use debugging functions from the Command Window, use dbstack to view the Function Call Stack.

4.6.3 Pause a running file

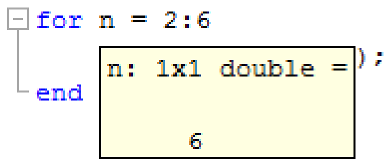
Pausing is useful if you want to check on the progress of a long running program to ensure that it is running as expected.

4.6.4 Find and Fix a Problem

While your code is paused, you can view or change the values of variables, or you can modify the code.

View or Change Variable While Debugging

View the value of a variable while debugging to see whether a line of code has produced the expected result or not. To do this, position your mouse pointer to the left of the variable. The current value of the variable appears in a data tip.



```
for n = 2:6
                     );
end
```

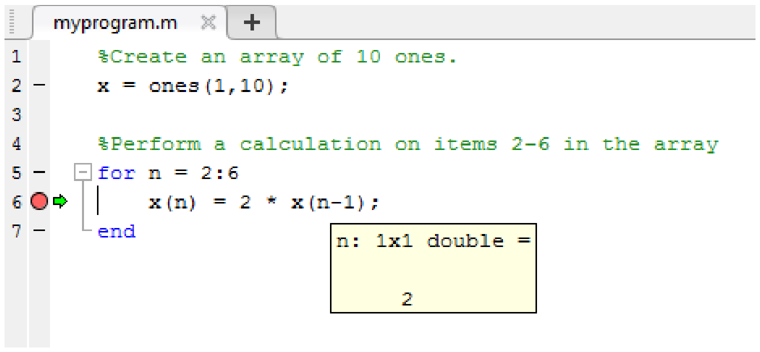
```
n: 1x1 double =

      6
```

The data tip stays in view until you move the pointer. If you have trouble getting the data tip to appear, click the line containing the variable, and then move the pointer next to the variable.

You can change the value of a variable while debugging to see if the new value produces expected results. With the program paused, assign a new value to the variable in the Command Window, Workspace browser, or Variables Editor. Then, continue running or stepping through the program.

For example, here MATLAB is paused inside a for loop where n = 2:



```
%Create an array of 10 ones.
x = ones(1,10);

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

```
n: 1x1 double =

  2
```

- Type n = 7; in the command line to change the current value of n from 2 to 7.
- Press **Continue** to run the next line of code.

MATLAB runs the code line x(n) = 2 * x(n-1); with n = 7.

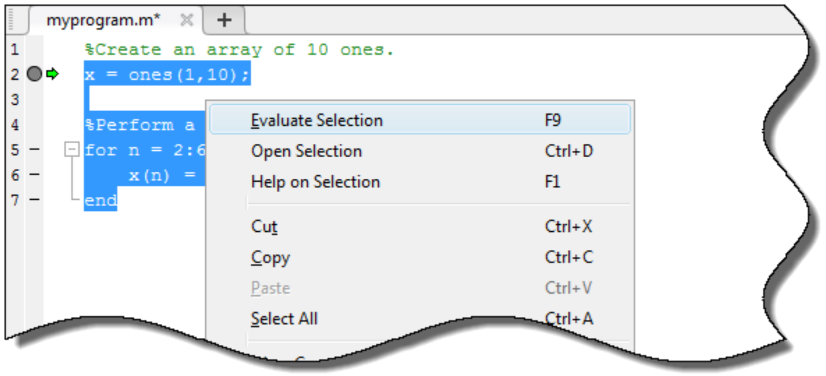
4.6.5 Modify Section of Code While Debugging

To modify a program while debugging:

1. While your code is paused, modify a part of the file that has not yet run.

   Breakpoints turn gray, indicating they are invalid.

2. Select all the code after the line at which MATLAB is paused, right-click, and then select **Evaluate Selection** from the context menu.



```
%Create an array of 10 ones.
x = ones(1,10);

%Perform a
for n = 2:6
    x(n) =
end
```

After the code evaluation is complete, stop debugging and save or undo any changes made before continuing the debugging process.

4.6.6 Step through File

While debugging, you can step through a MATLAB file, pausing at points where you want to examine values.

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

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

4.6.7 End Debugging Session

After you identify a problem, end the debugging session by going to the **Editor** tab and clicking **Quit Debugging** 🟥. You must end a debugging session if you want to change and save a file, or if you want to run other programs in MATLAB.

After you quit debugging, pause indicators in the Editor display no longer appear, and the normal >> prompt reappears in the Command Window in place of the K>>. You no longer can access the call stack.

If MATLAB software becomes nonresponsive when it pauses at a breakpoint, press **Ctrl+c** to return to the MATLAB prompt.

4.6.8 Debugging Commands

dbclear	Remove breakpoints
dbcont	Resume execution
dbdown	Reverse dbup workspace shift
dbquit	Quit debug mode
dbstack	Function call stack
dbstatus	List all breakpoints
dbstep	Execute next executable line from current breakpoint
dbstop	Set breakpoints for debugging
dbtype	Display file with line numbers
dbup	Shift current workspace to workspace of caller in debug mode
keyboard	Give control to keyboard
echo	Display statements during function execution