University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 1 - MATLAB Fundamentals I

Author: Dr. Ashraf E. Suyyagh

Table of Contents

What is MATLAB?

MathWorks has developed and maintained MATLAB since the 1970s. MATLAB is a programming language and software environment that manages data interactively.

MATLAB is historically oriented towards:

- Matrix operations and linear algebra.
- Numerical analysis.
- Fast and easy data plotting
- Interface with other programming languages
- Symbolic processing (*e.g.* differentiation and integration with symbols as you would solve problems in Calculus).

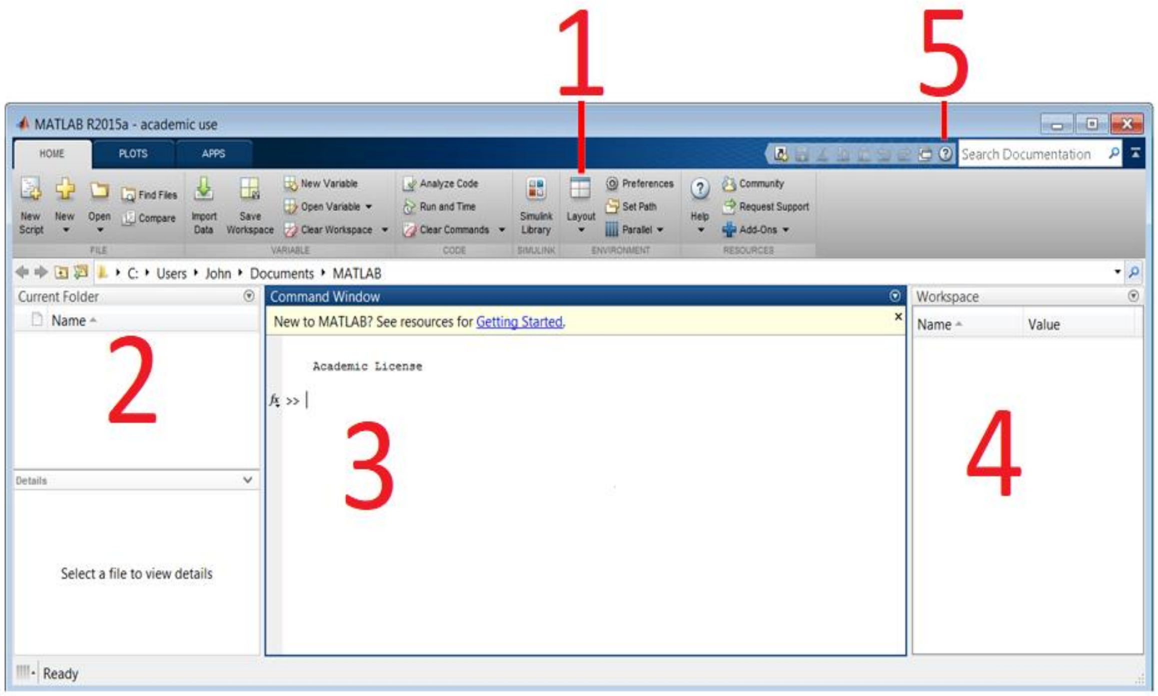
MATLAB has a wide array of toolboxes for image processing, signal processing, statistics, machine learning, control and automotive systems among dozen others (may require additional paid licenses, so we might not have all of them in the lab).

MATLAB provides through an additional integrated product called Simulink graphical model-based design and simulation for numerous engineering, scientific, and economic systems (may require additional paid licenses, so we might not have all of them in the lab).

C++ and Java are **compiled languages** where all the code is first compiled, then an executable is generated and run afterwards. In contrast, MATLAB is an **interpreted language. There is no code compilation. MATLAB** executes commands in order during run-time. That is; it understands (interprets) the commands one-by-one as they appear in the code/script.

MATLAB Environment

The main screen of MATLAB has five main windows:



1. The **Ribbon tab** where you have access to the IDE operations: opening new or existing files, save files, import data from files (*e.g.* Excel or CSV), set working path, run your code, and access MATLAB plots and numerous toolboxes.
2. **Current working directory** where you save all your MATLAB scripts, functions, plots ... *etc*. MATLAB commands expect to work with files in the current working directory ***(More on this later).*** You will learn how to setup the working path later.
3. **Command Line** where you write and execute your MATLAB commands. You will see the output of your commands in the same window.
4. **Workspace** where you all your session variables are saved and accessed.
5. **Help and Documentation** where you can search for any topic or command.

MATLAB as a Numerical Calculator

You can use MATLAB to quickly find the output of many mathematical operations. For example, try to copy/paste these codes one by one in the **command window**, then press **Enter** to see the output after each one.

```
10 + 10
```

```
ans = 20
```

```
65 - 35
```

```
ans = 30
```

```
85 * 5
```

```
ans = 425
```

```
2 ^ 10
```

```
ans = 1024
```

```
8 / 4
```

```
ans = 2
```

```
8 \ 4
```

```
ans = 0.5000
```

Note that in the last operation, we used the backslash instead of the division operator. The backslash acts as a division operator but in the reverse order. So, 8 \ 4 is equivalent to 4 / 8 = 0.5. This is called **backward division** while the normal division case is called **forward division.**

Notice that after each operation that you execute, your answer is stored in a special variable called ***ans*** *(*that is short for answer). There is only one ***ans*** variable that is constantly changing as it always stores the result of the most recent operation. Each subsequent operation essentially overrides the previous answer with its own result and stores it in ***ans***. You can see the ***ans*** variable to your right in the **Workspace** window. The workspace window displays all variables used in your session. Later on, when you work with large data (*e.g.* arrays), you can double-click on the variable and see its content in an Excel-like sheet.

MATLAB saves all your previous keystrokes. While in the command window, press the **Up-Arrow** on your keyboard. You will see all your previous commands. You can also access all previous commands by typing **commandhistory.** From there**,** you can select any command or group of commands by pressing **(SHIFT + LEFT Mouse button)** on each of them. You can revaluate these commands by **(Right-Click on your selection --> Evaluate Selection)**. You can edit the commands too by using **Backspace** and **Delete** keys and **Left** and **Right** arrow keys.

Order of Precedence of Arithmetic Operations

One easy way to remember the rules of precedence of arithmetic operations is by remembering the word: **PEMDAS.** Each letter stands for the first letter of the operations in their order of precedence.

1. **P**arenthesis
2. **E**xponentiation
3. **M**ultiplication and **D**ivision
4. **A**ddition and **S**ubtraction

When the precedence level is equal, evaluation is performed from **LEFT** to **RIGHT.**

Calculate the output of these examples by hand. Compare your answer by executing these examples in the command window. Copy/paste these expressions one by one in the **command window**, then press **Enter** to see the output after each one.

```
8 + 3 * 5
```

```
ans = 23
```

```
8 + (3 * 5)
```

```
ans = 23
```

```
(8 + 3) * 5
```

```
ans = 55
```

```
4 ^ 2 - 12 - 8 / 2 * 2
```

```
ans = -4
```

```
4 ^ 2 - 12 - 8 / (2 * 2)
```

```
ans = 2
```

```
4 ^ 2 - 12 - 8 \ (2 * 2)
```

```
ans = 3.5000
```

```
3 * 4 ^ 2 + 5
```

```
ans = 53
```

```
(3 * 4) ^ 2 + 5
```

```
ans = 149
```

```
27 ^ ( 1 /3 ) + 32 ^ (0.2)
```

```
ans = 5
```

```
27 ^ ( 1 /3 ) + 32 ^ 0.2
```

```
ans = 5
```

```
27 ^1 / 3 + 32 ^ 0.2
```

```
ans = 11
```

MATLAB Variables and the Assignment Operator

Unlike other programming languages like C++ or Java, you **DON'T** need to declare a MATLAB variable and define a type prior to its use. You can directly assign values to variables in MATLAB.

By default, and unless otherwise defined, **ALL** numeric values are treated as ***double-precision floating-point*** values in MATLAB.

Variables can directly store text, tables, structures, vectors and arrays (matrices) among other data types (more on this later).

As explained before, if no variable is used to assign the result to it, the default variable where the answer is stored is **ans.**

To define a new variable, simply write the variable name followed by the assignment operator (=) and the value to be stored in the variable. ONLY ONE variable can be written to the left hand side of the assignment operator. Expressions such as $x+5=20$ are **NOT ALLOWED.**

Copy/paste these examples one by one in the **command window**, then press **Enter** to see the output after each one.

```
a = 25
```

```
a = 25
```

```
b = a + 25
```

```
b = 50
```

```
5 * 20
```

```
ans = 100
```

```
c = ans * 10
```

```
c = 1000
```

```
d = 8
```

```
d = 8
```

```
d = 'numerical'
```

```
d = 'numerical'
```

Notice that we did not define any datatypes for our variables in the above examples **(There is no *int*, or *float*, or *double*, or *string*).**

Note that the output of 5 * 20 is implicitly stored in the **ans** variable (since the output was not assigned (stored) into any other variable). Also note that we can use the **ans** variable in further computations (c = ans * 10) even though *it is not a recommended practice* because **ans** always changes value.

Interestingly, in the above example, the variable **d** initially holds the numerical value 8. Then, we override it to store the **string** '*numerical*'. This is permissible since in MATLAB, if the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage.

Special Variables and Constants

MATLAB has a set of special variables **(case-sensitive):**

- **ans**: Temporary variable containing the most recent answer.
- **i** and **j**: the imaginary unit $\sqrt{-1}$.
- **inf or Inf**: the infinity ∞.
- **NaN**: stands for **N**ot **a**n **N**umber; indicates an undefined numerical result.
- **pi**: the math number π = 3.141592...
- **eps**: by default, this variable represents the machine's accuracy in double precision. It represents the distance from `1.0` to the next largest double-precision number. On a typical machine running a modern Intel processor ϵ = 2.220446049250313e-16
- **realmax:** Largest positive floating-point number
- **realmin:** Smallest normalized floating-point number

Copy/paste these examples one by one in the **command window**, then press **Enter** to see the output after each one:

```
d = 1 / 0
```

```
d = Inf
```

```
e = 0 / 0
```

```
e = NaN
```

While the above operations will result in an error on a hand-held calculator or in some other programming languages, MATLAB can handle results of infinity or undefined values.

Remember that in other programming languages such as C++ and Java that the range of signed integers of type **int** is -2147483648 to +2147483647. Now, try storing this number in MATLAB:

```
g = 4567892345638
```

```
g = 4.5679e+12
```

Surprisingly, the variable **g** was indeed declared, reserved in memory, and stored the number 4567892345638. But how could it store an integer value that exceeds the possible range?

Use the **class** command to determine the data type of the variable **g**

```
class(g)
```

```
ans = 'double'
```

Remember that MATLAB stores **all numeric values** **by default** as double-precision floating point numbers. This enables MATLAB to handle much larger range of integers and floating-point numbers. This becomes clear knowing that integers are a subset of floating-point numbers.

The largest/smallest integer number MATLAB can store **precisely** and **exactly in the default double floating-point type variable** without any errors is $2^{53} = \pm 9,007,199,254,740,992$. All other integers stored in MATLAB above or below this number are not exact integers. You should review the IEEE 754 standard which you have studied in the Computer Organization and Design course to know why.

Working with Complex Numbers

We have already introduced that the letters **i** and **j** are predefined variables in MATLAB to represent the imaginary number $\sqrt{-1}$. Note that you don't need a multiplication sign between the number and the imaginary unit **i** or **j**. Copy/paste these examples one-by-one in the **command window**, then press **Enter** to see the output after each one.

```
a = 1 +1j
```

```
a = 1.0000 + 1.0000i
```

```
b = 5 + 6i
```

```
b = 5.0000 + 6.0000i
```

```
c = a + b
```

```
c = 6.0000 + 7.0000i
```

When you are going to use **i** or **j** with any constant to represent a complex number, you must write the imaginary value without a multiplication sign. Even if the value is 1, it is ***recommended*** to *explicitly* precede it with 1, so it looks like *1i* or *1j*.

The above recommendation is important. Sometimes, you might by mistake override the variables **i** and **j** by any other value and MATLAB allows that without issues!

```
i = 2
```

```
i = 2
```

```
j = 3
```

```
j = 3
```

To restore the values of **i** and **j** to the default $\sqrt{-1}$, simply delete the variables by using the **clear** command and MATLAB will revert to the original meaning.

```
clear i j
```

As long as you don't override the values of **i** and **j**, these numbers are all complex.

```
clear i j
a1 = 1 + i        % COMPLEX
```

```
a1 = 1.0000 + 1.0000i
```

```
b1 = 1 + 1i       % COMPLEX
```

```
b1 = 1.0000 + 1.0000i
```

```
c1 = 1 + 1 * i    % COMPLEX
```

```
c1 = 1.0000 + 1.0000i
```

```
d1 = 1 + 2i       % COMPLEX
```

```
d1 = 1.0000 + 2.0000i
```

```
e1 = 1 + 2 * i    % COMPLEX
```

```
e1 = 1.0000 + 2.0000i
```

```
f1 = 5 + 6j       % COMPLEX
```

```
f1 = 5.0000 + 6.0000i
```

```
g1 = 5 + 6*j      % COMPLEX
```

```
g1 = 5.0000 + 6.0000i
```

However, once you override the default imaginary numbers with other values, the same expressions will be interpreted differently. Consider:

```
i = 1            % Overriding i and j
```

```
i = 1
```

```
j = 3
```

```
j = 3
```

```
a2 = 1 + i        % NOT COMPLEX
```

```
a2 = 2
```

```
b2 = 1 + 1i       % COMPLEX
```

```
b2 = 1.0000 + 1.0000i
```

```
c2 = 1 + 1 * i    % NOT COMPLEX
```

```
c2 = 2
```

```
d2 = 1 + 2i       % COMPLEX
```

```
d2 = 1.0000 + 2.0000i
```

```
e2 = 1 + 2 * i    % NOT COMPLEX
```

```
e2 = 3
```

```
f2 = 5 + 6j       % COMPLEX
```

```
f2 = 5.0000 + 6.0000i
```

```
g2 = 5 + 6*j      % NOT COMPLEX
```

```
g2 = 23
```

Notice that whenever you place a multiplication sign between the imaginary variable **i** or **j** and its coefficient, MATLAB no longer interpret **i** and **j** as $\sqrt{-1}$. Instead, they are considered as real-numbered variables and MATLAB uses the new overridden values. Also, if the imaginary coefficient is 1 and you forget to explicitly write the imaginary part as **1i** or **1j**, then in this case, the overridden value of **i** or **j** will be used.

The order of precedence will yield different values if you are not too careful in writing your expressions. Consider:

```
clear i j   % ensure default imaginary values are in use
a = 9/2*i   % a = 4.5i
```

```
a = 0.0000 + 4.5000i
```

```
b = 9/2i    % b = -4.5i
```

```
b = 0.0000 - 4.5000i
```

MATLAB treats the first expression as having three terms and will evaluate the expression according to the rules of precedence. Since division and multiplication have equal precedence, the expression will have the same result as:

```
a = (9/2)*i % a = 4.5i
```

```
a = 0.0000 + 4.5000i
```

Yet, MATLAB treats the second expression as having two terms only (due to the absence of the multiplication sign), so the second expression is basically the same as:

```
b = 9 /(2i) % b = -4.5i
```

```
b = 0.0000 - 4.5000i
```

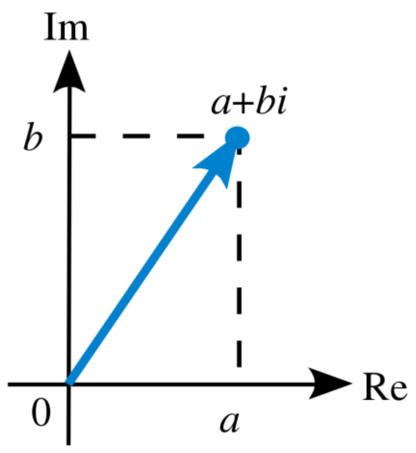
Perhaps a direct way to create MATLAB variable is by using MATLAB complex built-in function. If we represent a complex number as *a* + *bi*, then we can create a complex number by writing **complex(a, b)**

```
complex(5, 10)   % This is equivalent to 5 + 10i
```

```
ans = 5.0000 + 10.0000i
```

Complex Number Functions

While working with complex numbers, there are five main functions that we might frequently need. Splitting the imaginary and real parts. Determining the complex conjugate, and most importantly computing the magnitude and angle of the complex vector. Remember that a complex number *a* + *bi* can be projected on the complex plane as:



MATLAB has all these operations as built-in functions. The following example illustrates how to use them:

```
a = 5+3j
```

```
a = 5.0000 + 3.0000i
```

```
b = complex(7, -8)
```

```
b = 7.0000 - 8.0000i
```

```

abs(a)      % Magnitude of a complex number
```

```
ans = 5.8310
```

```
abs(b)
```

```
ans = 10.6301
```

```
angle(a)    % Phase angle of complex number (in RADIANs)
```

```
ans = 0.5404
```

```
angle(b)
```

```
ans = -0.8520
```

```
conj(a)     % Complex Conjugate
```

```
ans = 5.0000 - 3.0000i
```

```
conj(b)
```

```
ans = 7.0000 + 8.0000i
```

```
real(a)     % Extract real part of complex number
```

```
ans = 5
```

```
real(b)
```

```
ans = 7
```

```
imag(a)     % Extract imaginary part of complex number
```

```
ans = 3
```

```
imag(b)
```

```
ans = -8
```

MATLAB Built-in Mathematical Functions

MATLAB supports lots of common mathematical functions. We provide a listing of the major ones which you already know from algebra and calculus courses.

Please note that the **TRIGONOMETRIC FUNCTIONS** IN MATLAB USE ***<u>RADIAN</u>*** MEASURE by default.

Mathematical Functions

Exponential and Logarithmic Functions	
`exp(x)`	Exponential; e^x.
`log(x)`	Natural logarithm; ln(x).
`log10(x)`	Common (base 10) logarithm; log(x)= $\log_{10}(x)$.
`sqrt(x)`	Square root; $\sqrt{x}$.

Trigonometric Functions	
`acos(x)`	Inverse cosine; arcos x = $\cos^{-1}(x)$.
`acot(x)`	Inverse cotangent; arccot x = $\cot^{-1}(x)$.
`acsc(x)`	Inverse cosecant; arcs x = $\csc^{-1}(x)$.
`asec(x)`	Inverse secant; arcsec x = $\sec^{-1}(x)$.
`asin(x)`	Inverse sine; arcsin x = $\sin^{-1}(x)$.
`atan(x)`	Inverse tangent; arctan x = $\tan^{-1}(x)$.
`atan2(y,x)`	Four-quadrant inverse tangent.
`cos(x)`	Cosine; cos(x).
`cot(x)`	Cotangent; cot(x).
`csc(x)`	Cosecant; csc(x).
`sec(x)`	Secant; sec(x).
`sin(x)`	Sine; sin(x).
`tan(x)`	Tangent; tan(x).

Hyperbolic Functions	
`acosh(x)`	Inverse hyperbolic cosine; $\cosh^{-1}(x)$.
`acoth(x)`	Inverse hyperbolic cotangent; $\coth^{-1}(x)$.
`acsch(x)`	Inverse hyperbolic cosecant; $\mathrm{csch}^{-1}(x)$.
`asech(x)`	Inverse hyperbolic secant; $\mathrm{sech}^{-1}(x)$.
`asinh(x)`	Inverse hyperbolic sine; $\sinh^{-1}(x)$.
`atanh(x)`	Inverse hyperbolic tangent; $\tanh^{-1}(x)$.
`cosh(x)`	Hyperbolic cosine; cosh(x).
`coth(x)`	Hyperbolic cotangent; cosh(x)/sinh(x).
`csch(x)`	Hyperbolic cosecant; 1/sinh(x).
`sech(x)`	Hyperbolic secant; 1/cosh(x).
`sinh(x)`	Hyperbolic sine; sinh(x).
`tanh(x)`	Hyperbolic tangent; sinh(x)/cosh(x).

Let's try some of these functions:

```
a = sqrt(9)
```

```
a = 3
```

```
b = sqrt(-16)      % returns a complex number 0 + 4i
```

```
b = 0.0000 + 4.0000i
```

Since $10^2 = 100$, then $log_{10}(100) = 2$

```
c = log10(100)
```

```
c = 2
```

and $ln(e^3) = 3$

```
d = log(exp(3))
```

```
d = 3
```

But what if we need to find the natural logarithm for a base different than 10 or e? In this case, we need to use mathematical rules:

$$log_b(a) = \frac{log_{10}(a)}{log_{10}(b)}$$

so if you need to compute $log_3(6) = \frac{log_{10}(6)}{log_{10}(3)}$ which can be easily computed in MATLAB as:

```
log10(6) / log10(3)
```

```
ans = 1.6309
```

As we know, $sin(30^\circ) = 0.5$, attempting this directly in MATLAB will yield the wrong result:

```
sin(30)
```

```
ans = -0.9880
```

Remember, trigonometric functions in MATLAB use **RADIAN** as function input by default, so you must convert the value from decimal to RADIAN by multiplying it by $\frac{\pi}{180}$:

```
sin(30*pi/180)
```

```
ans = 0.5000
```

Similarly, we know that $cos^{-1}(0.5) = 60^{\circ}$, attempting this directly in MATLAB will yield the result in radian:

```
acos(0.5)
```

```
ans = 1.0472
```

so don't forget to correct the output - if necessary - by converting it to decimal by multiplying it by $\frac{180}{\pi}$:

```
acos(0.5)*180/pi
```

```
ans = 60.0000
```

If you wish to have the square of a trigonometric function, say $cos^2(x)$, then the correct way to do it is to square the whole function in MATLAB, as in $(cos(x))^2$

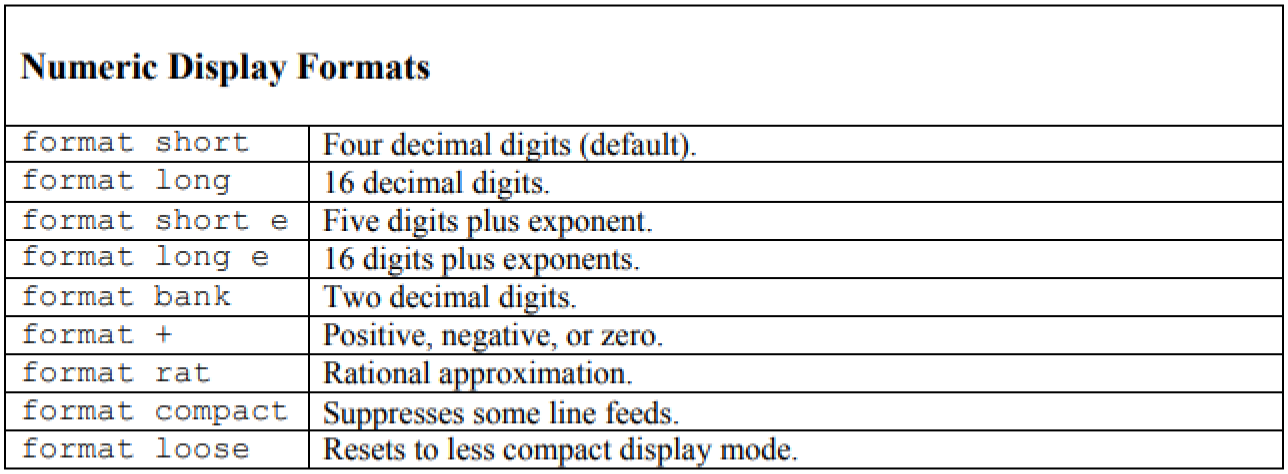
Finally, we already learnt that using the $abs()$ function with complex numbers returns the magnitude of the complex numbers. You can use the $abs()$ function with real numbers as well, in this case, it means the absolute value $|x|$, try:

```
abs(-5)
```

```
ans = 5
```

Numeric Display Formats

MATLAB's ***format*** command gives us control on *HOW TO DISPLAY* the output numbers. They **DO NOT** change the actual accuracy of the number stored in memory. Just what you see on the screen. Some formatting commands affect the look and feel of the workspace (*e.g*. line spacing). The following table lists all available output formatting commands:



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

The format **compact** and **loose** functions control the **line spacing** in the command window.

Let's try few of them with the number π

```
format short
pi
```

```
 ans = 3.1416
```

```
format long
pi
```

```
 ans =
       3.141592653589793
```

```
format short e
pi
```

```
 ans =
       3.1416e+00
```

```
format long e
pi
```

```
 ans =
         3.141592653589793e+00
```

```
format bank
pi
```

```
 ans =
         3.14
```

```
format +
pi
```

```
 ans =
    +
```

```
format rat
pi
```

```
 ans =
         355/113
```

```
format hex        % Hexadecimal representation
pi
```

```
 ans =
       400921fb54442d18
```

```
format            % Resets to the default formatting
```

Discrete Mathematics in MATLAB

In the previous experiment, we learnt how to do various vector and matrix operations such as logical, arithmetic, and other manipulations. We mostly applied basic trigonometric and linear algebra math. In this section, we will learn how to handle mathematical sets in the same way you learnt in basic math and discrete math courses.

Sets in MATLAB

MATLAB allows users to enter mathematical sets as a vector or a matrix (array) - more on vectors and matrices later. For example, lets generate the two mathematical sets S_1 and S_2:

$$S_1 = \{1, 5, 4, 6, 7, 7, 3, 5, 2, 8, 0, 1, 1, 2, 7, 8, 5, 4, 2, 4, 1, 4\}$$

$$S_2 = \{0, 9, 8, 9, 9, 7, 3, 10, 5, 3, 6, 2, 4, 1, 1, 1, 11, 3\}$$

as two vectors:

```
S1 = [1, 5, 4, 6 ,7 ,7, 3, 5, 2, 8, 0, 1, 1, 2, 7, 8 ,5, 4, 2, 4, 1, 4];
S2 = [0, 9, 8, 9, 9, 7, 3, 10, 5, 3, 6, 2, 4, 1, 1, 1, 11, 3];
```

To extract the numbers that occur in the set without any repetitions; that is to extract the unique numbers in the set, use the function **`unique`**:

```
unique(S1)
```

```
ans = 1×9
     0     1     2     3     4     5     6     7     8
```

```
unique(S2)
```

```
ans = 1×12
     0     1     2     3     4     5     6     7     8     9    10    11
```

The function **`unique`** returns the existing elements in the set <u>in sorted ascending order</u>. If you wish to see the unique numbers in the set in the order they appear in, then use:

```
unique(S1, 'stable')
```

```
ans = 1×9
     1     5     4     6     7     3     2     8     0
```

To merge two sets together; that is, to perform the operation $S_1 \cup S_2$, use the **`union`** command:

```
union(S1, S2)
```

```
ans = 1×12
     0     1     2     3     4     5     6     7     8     9    10    11
```

The union commands can work with sets stored in matrices of different sizes and shapes. There is no need for the vectors or matrices to match each other's dimensions.

To find the elements that appear in both sets; that is, to perform the operation $S_1 \cap S_2$, use the **`intersect`** command:

```
intersect(S1, S2)
```

```
ans = 1×9
     0     1     2     3     4     5     6     7     8
```

The default output is sorted in ascending order, unless you use *'stable'* as before.

To see the elements that exist in one set but not the other, use set difference command **`setdiff`.** Note that the order you use in performing this operation matters:

```
setdiff(S1, S2)     % All elements in S1 appear in S2, so set difference is empty
```

```
ans =

  1×0 empty double row vector
```

```
setdiff(S2, S1)
```

```
ans = 1×3
     9    10    11
```

Another method to check if an element exists in a set is to use the **`ismember`** command. It returns **logical** 0 if the number does not exist in the set, and 1 otherwise. To check if the numbers **5, 8, 10** and **15** exist in S_1:

```
ismember([5, 8, 10, 15], S1)
```

```
ans = 1×4 logical array
   1   1   0   0
```

Discrete Mathematics

MATLAB has extremely powerful functions in the domain of discrete mathematics. We list them in the following table:

factor	Returns the prime factors of the number *n*
isprime	Determine which array elements are prime
primes	Lists prime numbers less than or equal to the input value
nextprime	Returns the next prime after input number *n*
prevprime	Returns the previous prime before input number *n*
nthprime	Returns the n^{th} prime number
gcd	Greatest common divisor
lcm	Least common multiple
factorial	Factorial of input *n*!
perms	Returns a matrix containing all permutations of the elements of the input vector in reverse lexicographic order

Let's try few examples. To list all prime numbers under 50, write:

```
primes(50)
```

```
ans = 1×15
     2     3     5     7    11    13    17    19    23    29    31    37
41 ⋯
```

The 100^{th} prime number is:

```
nthprime(100)
```

```
ans = 541
```

To factor the number 567838 into its factors, use:

```
factor(567840)
```

```
ans = 1×10
     2     2     2     2     2     3     5     7    13    13
```

The least common multiple of $10, 6$ is:

```
lcm(10, 6)
```

```
ans = 30
```

To list all possible combinations of the numbers $3,\ 7,\ 9$, use the function **perms**:

```
perms([3, 7, 9])
```

```
ans = 6×3
      9     7     3
      9     3     7
      7     9     3
      7     3     9
      3     9     7
      3     7     9
```

Performance Timing in MATLAB

Many times, we are not only interested in writing a correct code, but also an efficient code. Performance timing is important in order to compare how much time has lapsed in writing two versions of the same function or algorithm.

There are many ways to profile the timing of your code. but we will only explain a simple one:

- Use the **`tic`** and **`toc`** commands at the beginning and end of the function you want to profile. Once you execute the **`toc`** command, the elapsed time will appear. (***RECOMMENDED***)

```
tic
% Your algorithm goes here, do something
toc
```

```
Elapsed time is 0.001809 seconds.
```

NOTE: Usually, we do not profile the execution time based on a single execution. We place our function or algorithm in a loop and run it hundreds of times, then we take the average of execution times.

We shall learn about more MATLAB tools for code timing and profiling in the next experiment.

Number Rounding and Rational Fraction Approximation

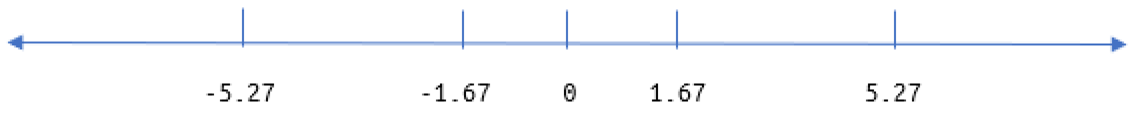
MATLAB also offers functions that perform the rounding operations that you are familiar with from math or previous programming courses. We summarize these functions in the following table:

Numeric Functions	
`ceil`	Rounds to the nearest integer toward ∞.
`fix`	Rounds to the nearest integer toward zero.
`floor`	Rounds to the nearest integer toward - ∞.
`round`	Rounds towards the nearest integer.
`sign`	Signum function.

The **`ceil`** and **`floor`** functions are the ones with the math symbols $\lceil x \rceil$ and $\lfloor x \rfloor$, respectively. The **`fix`** function returns the number closest to zero. The differences are clearer when comparing the results of positive and negative numbers. The **`sign`** function is a very useful function which returns the signs of the input number(s). We show some insightful examples in the following figure:

ceil (-5.27)	-5
fix (-5.27)	-5
floor (-5.27)	-6
round (-5.27)	-5
round (-5.27, 1)	-5.3
sign	-1

ceil (5.27)	6
fix (5.27)	5
floor (5.27)	5
round (5.27)	5
round (5.27, 1)	5.3
sign (5.27)	1



ceil (-1.67)	-1
fix (-1.67)	-1
floor (-1.67)	-2
round (-1.67)	-2
round (-1.67, 1)	-1.7
sign (-1.67)	-1

ceil (1.67)	2
fix (1.67)	1
floor (1.67)	1
round (1.67)	2
round (1.67, 1)	1.7
sign (1.67)	1

The rounding function **round** has different syntax for different cases; the default syntax takes one input (the number to be rounded) and works at the "half digit" boundary. **Positive numbers** with fractional parts 0.5 - 0.99... get rounded to the next largest number; otherwise, they get rounded to the lower integer value. Negative numbers rounding works in reverse. Consult the examples in the table above.

Let us try few rounding examples; let's round the number: 67585.891:

```
round(67585.891)          % Default roundig behaviour
```

```
ans = 67586
```

If we want to round to one decimal point:

```
round(67585.891, 1)
```

```
ans = 6.7586e+04
```

Notice that the result is given in engineering format, and that we cannot see the effect clearly. Precede the **round** command with **format long**

```
format long
round(67585.891, 1)
```

```
ans =
     6.758589999999999e+04
```

Notice that we expect the result to be 67585.9 yet MATLAB returns a result of 67585.899999999….. This is due to the IEEE 754 standard and hardware design of floating-point units inside computers which does not always allow us to represent floating-point error precisely. We shall discuss this in detail in a future experiment.

and to two decimal points:

```
format long
round(67585.891, 2)
```

```
ans =
    6.758589000000000e+04
```

Notice that we expect the result to be 67585.89 and MATLAB indeed returns the correct result.

Alternatively, one can round to the nearest 10 or nearest 100 or 1000 by writing:

```
round(67585.891, -1)
```

```
ans =
      67590
```

```
round(67585.891, -2)
```

```
ans =
      67600
```

```
round(67589.891, -3)
```

```
ans =
      68000
```

Commands for Managing the Work Session

In this section, we will present commands that are quite useful in managing your workspace.

We already know that all variables you declare in any MATLAB session can be found under the **Workspace** window (default location --> left pane). You can also list the session variables in the command window by using this command:

```
who
```

```
Your variables are:

S1   a    a2   b    b2   c1   d    d2   e1   f1   g    g2
S2   a1   ans  b1   c    c2   d1   e    e2   f2   g1
```

If you need further details about your variables, you can use:

```
whos
```

```
  Name      Size            Bytes  Class     Attributes

  S1        1x22              176  double
  S2        1x18              144  double
  a         1x1                 8  double
  a1        1x1                16  double    complex
  a2        1x1                 8  double
  ans       1x1                 8  double
  b         1x1                16  double    complex
  b1        1x1                16  double    complex
  b2        1x1                16  double    complex
  c         1x1                 8  double
  c1        1x1                16  double    complex
  c2        1x1                 8  double
  d         1x1                 8  double
  d1        1x1                16  double    complex
  d2        1x1                16  double    complex
  e         1x1                 8  double
  e1        1x1                16  double    complex
  e2        1x1                 8  double
  f1        1x1                16  double    complex
  f2        1x1                16  double    complex
  g         1x1                 8  double
  g1        1x1                16  double    complex
  g2        1x1                 8  double
```

Notice, that the size of most variables we used thus far is listed as `1x1.` This is because all our variables are scalars (not vectors nor matrices (arrays), just single-valued numbers). Even the complex numbers have a size of `1x1` as they are treated as one unit. This size attribute is different than the actual size these variables take inside the memory. You can find the memory size under **Bytes**. Since most our variables are double-precision floating-point numbers, we expect they will use 8 bytes, and complex numbers will take 16 bytes to store both the real and imaginary parts. You can find the data type of your variables under the **Class attribute**.

In huge projects, you might lose track of variables. You might forget if you have used the variable *'num'* for example, and you don't need to override it and cause errors in your program. To check if a variable exists, use the **exist** function. It takes in ONE variable name in single quotations or directly listed next to it. It returns 0 or 1 depending if the variable exists or not.

```
exist('num')
```

```
ans =
       0
```

```
exist a
```

```
ans =
       1
```

To delete variables from the workspace, use **clear** then list the variable names to be deleted. Let's delete variables a, b, c, and d.

```
exist a
```

```
ans =
       1
```

```
clear a b c d
exist a
```

```
ans =
       0
```

To clear all variables in the workspace, simply write **clear** on its own:

```
clear
```

To clear the content of the command window, simply use:

```
clc
```

The above command does not delete the history of your commands, only the command window. You can still see all previous commands by pressing the **UP** button or writing **commandhistory**.

When you are writing large MATLAB scripts and functions (*discussed in later experiments*), you are only interested in the final output. To suppress (hide) the output of MATLAB lines, use the semicolon at the end. Compare:

```
a = (5 + 4j)*(6 - 2j)
```

```
a =
 38.000000000000000 +14.000000000000000i
```

```
b = (7 + 6j)*(7 - 9j);
```

In both cases, the output is computed and stored in the variables **a** and **b**. In the first case, MATLAB shows the output in the command window, while in the latter case, the semicolon suppressed the output from showing. To see the output of the variable **b**, you can either type the variable directly:

```
b
```

```
b =
      1.030000000000000e+02 - 2.100000000000000e+01i
```

or use the display **disp()** function:

```
disp(b)
```

```
      1.030000000000000e+02 - 2.100000000000000e+01i
```

When you are writing long lines in MATLAB, and you wish to continue writing on the next line, you can use the Ellipsis (three dots ...)

```
1245 + 86844 + 767683 + 34334 + ...
456  + 97800
```

```
ans =
      988362
```

You can automatically capture and log (store) all entered command, keyboard input and command window output using the **diary** command by using it with the **on** and **off** switches. MATAB will create a text file called *diary* in the working directory that contains all session command window interactions.

```
diary on
% Some commands
diary off
```

You can also display the content of text files inside MATLAB's command window by using the **type** command. MATLAB will look for the supplied filename inside the current path (working directory), or any added path. Otherwise, you need to provide the full path to the **type** command:

```
type sampleNumbers.txt
```

```
0.0975    0.1576    0.1419    0.6557    0.7577    0.7060    0.8235    0.4387
0.2785    0.9706    0.4218    0.0357    0.7431    0.0318    0.6948    0.3816
0.5469    0.9572    0.9157    0.8491    0.3922    0.2769    0.3171    0.7655
0.9575    0.4854    0.7922    0.9340    0.6555    0.0462    0.9502    0.7952
0.9649    0.8003    0.9595    0.6787    0.1712    0.0971    0.0344    0.1869
```

The file *`sampleNumbers.txt`* contains a few lines of numbers that can be printed inside the command window.

To quit MATLAB from the command line, simply write quit.

```
quit
```

MATLAB Help and Documentation

MATLAB has an extensive well-written documentation with numerous examples as well an online MATLAB answers forum. In MATLAB, there are two options to access info on all commands: a quick help command and another for full documentation. For example, if you need to quickly see the function and syntax of a MATLAB command, type **help** followed by the command name, and a short documentation will appear inside the command window and it will list all available MATLAB commands that handle binary numbers one way or another:

```
help format
```

```
  format Set output format.
     format with no inputs sets the output format to the default appropriate
```

```
    for the class of the variable. For float variables, the default is
    format SHORT.

    format does not affect how MATLAB computations are done. Computations
    on float variables, namely single or double, are done in appropriate
    floating point precision, no matter how those variables are displayed.
    Computations on integer variables are done natively in integer. Integer
    variables are always displayed to the appropriate number of digits for
    the class, for example, 3 digits to display the INT8 range -128:127.
    format SHORT and LONG do not affect the display of integer variables.

    format may be used to switch between different output display formats
    of all float variables as follows:
      format SHORT     Short fixed point format with 4 digits after the
                       decimal point.
      format LONG      Long fixed point format with 15 digits after the
                       decimal point for double values and 7 digits after
                       the decimal point for single values.
      format SHORTE    Short scientific notation with 4 digits after the
                       decimal point.
      format LONGE     Long scientific notation with 15 digits after the
                       decimal point for double values and 7 digits after
                       the decimal point for single values.
      format SHORTG    Short fixed format or scientific notation,
                       whichever is more compact, with a total of 5 digits.
      format LONGG     Long fixed format or scientific notation, whichever
                       is more compact, with a total of 15 digits for
                       double values and 7 digits for single values.
      format SHORTENG  Engineering format with 4 digits after the decimal
                       point and a power that is a multiple of three.
      format LONGENG   Engineering format that has exactly 15 significant
                       digits and a power that is a multiple of three.

    format may be used to switch between different output display formats
    of all numeric variables as follows:
      format HEX       Hexadecimal format.
      format +         The symbols +, - and blank are printed
                       for positive, negative and zero elements.
                       Imaginary parts are ignored.
      format BANK      Currency format with 2 digits after the decimal
                       point.
      format RATIONAL  Approximation by ratio of small integers. Numbers
                       with a large numerator or large denominator are
                       replaced by *.

    format may be used to affect the spacing in the display of all
    variables as follows:
      format COMPACT  Suppresses extra line-feeds.
      format LOOSE    Puts the extra line-feeds back in.
```

If you need to access the full documentation, simply type **doc** followed by the command name. The documentation window will open outside of MATLAB.

```
doc format
```

But what if you don't know the command name? How to find a certain command that does a specific job you want? In this case, use the command **lookfor** with **one keyword** that describes what you want, and MATLAB will list all available close commands with a summary of what they do.

For example, to look for MATLAB commands that can work with binary numbers, simply write:

```
lookfor binary
```

```
  bsxfun                        - Binary Singleton Expansion Function
  fread                         - Read binary data from file.
  fwrite                        - Write binary data to file.
  bin2dec                       - Convert text representation of binary number to double
  value
  dec2bin                       - Convert decimal integer to its binary representation
  binary                        - Sets binary transfer type.
  binary                        - Sets binary transfer type.
  bsxfun                        - Binary Singleton Expansion Function
  invokeBinaryComparison        - Invokes GE, LT etc.
  bops                          - o = BOPS( obj )  Returns the binary operator nodes in obj
  isbop                         - b = ISBOP( obj )   Boolean array, true if node is binary
  binaryVectorToDecimal         - Convert binary vector to a decimal number.
  binaryVectorToHex             - Convert binary vector to a hexadecimal character string.
  binvec2dec                    - Convert binary vector to decimal number.
  dec2binvec                    - Convert decimal number to a binary vector.
  decimalToBinaryVector         - Convert decimal number to a binary vector.
  hexToBinaryVector             - Convert hex number to a binary vector.
  dspblkbinaryfilereader        - DSP System Toolbox binary file reader block
  dspblkbinaryfilewriter        - DSP System Toolbox binary file writer block
  dec2mvl                       - Convert decimal integer to a binary string.
  fibinscaling                  - Fi Binary Point Scaling Demo
  fiscalingdemo                 - Perform Binary-Point Scaling
  bin                           - Binary representation of stored integer of fi object
  isscalingbinarypoint          - Determine whether fi object has binary point scaling
  BinaryPointScaling            -
  bwarea                        - Area of objects in binary image.
  bwareafilt                    - Extract objects from binary image by size.
  bwareaopen                    - Remove small objects from binary image.
  bwboundaries                  - Trace region boundaries in binary image.
  bwconncomp                    - Find connected components in binary image.
  bwconvhull                    - Generate convex hull image from binary image.
  bwdist                        - Distance transform of binary image.
  bwdist_old                    - Distance transform of binary image.
  bwdistgeodesic                - Geodesic distance transform of binary image.
  bweuler                       - Euler number of binary image.
  bwfill                        - Fill background regions in binary image.
  bwhitmiss                     - Binary hit-miss operation.
  bwlabel                       - Label connected components in 2-D binary image.
  bwlabeln                      - Label connected components in binary image.
  bwmorph                       - Morphological operations on binary image.
  bwmorph3                      - Morphological operations on binary volume.
  bwpack                        - Pack binary image.
  bwperim                       - Find perimeter of objects in binary image.
```

You can see in the list two functions of interest: **bin2dec** and **dec2bin** which allows you to convert between binary and decimal numbers in text formatting. Try them out.

```
                                           Experiment version 1.2
                              Original Experiment October 5th, 2020
                                       Last Updated March 3rd, 2022
                               Dr. Ashraf Suyyagh - All Rights Reserved
```

Revision History

Ver. 1.2 (March 2022)
Merged the first and third Experiments together
Removed discussion for hexadecimal and binary number formats section.
Removed the Integer Arithmetic section
Removed the time and date sections
Removed serial dates section
Removed the characters and strings sections
Removed the sections related to writing and loading excel sheets and text files using the command line and keep the GUI import tool yet moved it to the second experiment.
Ver. 1.1 (July 2021)
Corrected formatting and spelling mistakes
Added the *diary* and *type* commands.
Added discussion for hexadecimal and binary number formats.

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 2 - MATLAB Fundamentals II

Author: Dr. Ashraf E. Suyyagh

Table of Contents

Scalars, Vectors, and Matrices

In the previous experiment, we have introduced single-valued variables and many of the scalar operations associated with them. However, MATLAB really shines when working with vectors and matrices. In many other programming languages that you might have learnt already (C++ and Java), you stored vectors and matrices inside 1D and 2D arrays, respectively. You conducted all array related operations through loops. This is time consuming and error prone.

In MATLAB, storing vectors and matrices is straightforward through a simple assignment operator. Further, all associated operations DO NOT need loops. You can use simple mathematical operations almost directly.

This is one main reason why MATLAB is widely used by millions of engineers and scientists.

MATLAB vectors and arrays can hold numbers, characters (array of strings), or logical values (0 or 1). But an array or matrix **CANNOT** hold a mix of these types at the same time.

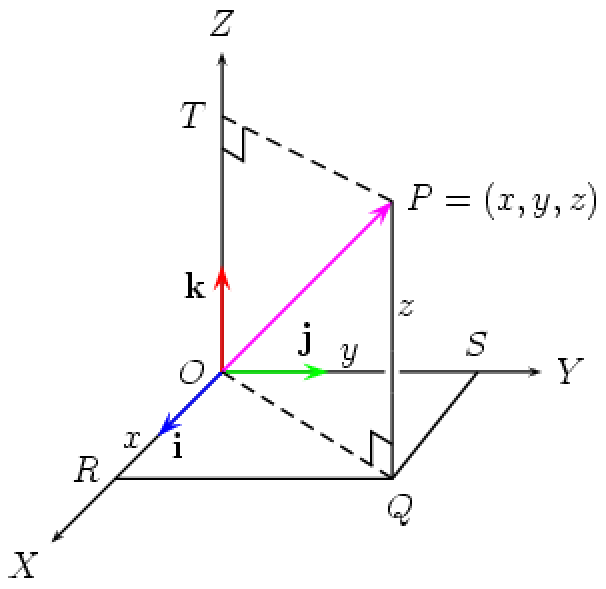
Previously, when we used the ***whos*** command, we noticed that all single-valued variables (scalars) had a size of 1x1. MATLAB treats scalars as one-element vectors of size 1x1.

Vectors

A vector is simply a one-dimensional matrix. In Physics and Mathematics, and other engineering courses, you often worked with vectors that are projected onto a 3D Cartesian coordinate system. The vector in this coordinate system is often given as:

$$x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

where x, y, z are the scalar magnitudes of the vector projections onto the Cartesian axes.



Creating Vectors

In MATLAB, you can easily define a 3D vector in two forms: a row vector or a column vector.

So, we can write $5\mathbf{i} + 8\mathbf{j} - 9\mathbf{k}$ in row form: $[5 \;\; 8 \;\; -9]$ or column form: $\begin{bmatrix} 5 \\ 8 \\ -9 \end{bmatrix}$ as follows:

Row form **(Use <u>commas</u> to separate elements):**

```
v1 = [5, 8, -9]
```

```
v1 = 1×3
     5     8    -9
```

Column form **(Use <u>semicolons</u> to separate elements):**

```
v2 = [5; 8; -9]
```

```
v2 = 3×1
      5
      8
     -9
```

You can convert between column and row vectors using the transpose operator ('), for example:

```
v3 = [5, 8, -9]'     % Creates a column vector by transposing a row vector
```

```
v3 = 3×1
      5
      8
     -9
```

```
v4 = [5; 8; -9]'     % Creates a row vector by transposing a column vector
```

```
v4 = 1×3
      5     8    -9
```

```
v5 = v3'             % Creates a column vector by transposing a row vector
```

```
v5 = 1×3
      5     8    -9
```

Vectors are not restricted to 3-elements. You can create row or column vectors of any size n. Use **whos** command with the variable name to see detailed info about the vector variable.

```
v6 = [0.5, 6, 9, 0, 0, 12, -8, 8]       % This is a 1x8 row vector
```

```
v6 = 1×8
    0.5000    6.0000    9.0000         0         0   12.0000   -8.0000 ⋯
```

```
whos v6
```

```
  Name      Size            Bytes  Class     Attributes
  v6        1x8                64  double
```

```
v7 = [0.9; 0; 0; 9; -9; 1; -3; 6]       % This is a 8x1 column vector
```

```
v7 = 8×1
    0.9000
         0
         0
    9.0000
   -9.0000
    1.0000
   -3.0000
    6.0000
```

```
whos v7
```

```
  Name      Size            Bytes  Class     Attributes

  v7        8x1                64  double
```

Note that unlike C++ or Java, we did not need to declare and reserve memory for a 1D array then use loops to store elements in the 1D array. It is straightforward and simple.

Appending Vectors

You can create vectors from other vectors by appending them together. The only condition is that these vectors are of the same data type and shape; that is, you can only append row vectors together, or column vectors together, but not both.

Let's append the row vector $v1$ with the row vector $v6$ (as before, use the **comma** in creating row vectors):

```
v8 = [v1, v6]
```

```
v8 = 1×11
    5.0000    8.0000   -9.0000    0.5000    6.0000    9.0000         0 ⋯
```

Let's append the column vector $v2$ with the column vector $v7$ (as before, use the **semicolon** in creating column vectors):

```
v9 = [v2; v7]
```

```
v9 = 11×1
    5.0000
    8.0000
   -9.0000
    0.9000
         0
         0
    9.0000
   -9.0000
    1.0000
   -3.0000
```

Appending row and column vectors will result in an error, try:

```
[v1, v7]
[v1; v7]
```

You can append row vectors with column vectors ONLY if you transpose one of them to match the other, try:

```
v10 = [v1, v7']     % Appending into row vector
```

```
v10 = 1×11
    5.0000    8.0000   -9.0000    0.9000         0         0    9.0000 ⋯
```

```
v11 = [v2; v6']      % Appending into column vector
```

```
v11 = 11×1
     5.0000
     8.0000
    -9.0000
     0.5000
     6.0000
     9.0000
          0
          0
    12.0000
    -8.0000
       ⋮
```

Useful techniques for creating large vectors

Many times, you will work with vectors of hundreds or thousands of elements. Hand entry is impossible.

If you want to create a vector with regularly spaced elements, use the colon (:). The syntax is $[m : q \ : n]$ where:

- m is the starting point.
- q is the spacing between elements.
- n is the upper limit (not necessarily the last element, but the last element will not exceed n). if $m - n$ is an integer multiple of q, then n will be the last element.

In this syntax, the number of elements is given by:

$$No = \left\lfloor \frac{n-m}{q} \right\rfloor + 1$$

```
v12 = [0: 2: 8]
```

```
v12 = 1×5
     0     2     4     6     8
```

```
v13 = [0: 2: 7]
```

```
v13 = 1×4
     0     2     4     6
```

```
v14 = [1: 1: 10]
```

```
v14 = 1×10
     1     2     3     4     5     6     7     8     9    10
```

Notice that if don't specify a value for the increment q, the default value is 1.

```
v15 = [1:10]
```

```
v15 = 1×10
         1     2     3     4     5     6     7     8     9    10
```

The increment q can be a floating-point number, or negative:

```
v16 = [0: 0.25: 4]
```

```
v16 = 1×17
           0    0.2500    0.5000    0.7500    1.0000    1.2500    1.5000 ⋯
```

```
v17 = [10: -2: -10]
```

```
v17 = 1×11
    10     8     6     4     2     0    -2    -4    -6    -8   -10
```

To retrieve the number of elements in a vector, use the **length** command:

```
length(v12)
```

```
ans = 5
```

```
length(v16)
```

```
ans = 17
```

If you need to create a vector with n elements, and you know the starting and ending points, but not the exact spacing, you can use the **linspace** command. It has the syntax:

$linspace(start, end, n)$

The increment is automatically computed and is equal to:

$$Increment = \frac{end - start}{n-1}$$

```
v18 = linspace(5, 8, 31)
```

```
v18 = 1×31
    5.0000    5.1000    5.2000    5.3000    5.4000    5.5000    5.6000 ⋯
```

The above command is similar to:

```
v19 = [5:0.1:8]
```

```
v19 = 1×31
    5.0000    5.1000    5.2000    5.3000    5.4000    5.5000    5.6000 ⋯
```

In engineering problems, sometimes we need entries with logarithmic spacing instead of regular (fixed) spacing. In this case, we can use the **logspace** command. By default, this command creates

50 elements unless the number of entries is specified. The elements lie between 10^a and 10^b, where a and b are user defined.

```
v20 = logspace(1,5)     % creates 50 elements between 10 and 100000
```

```
v20 = 1×50
10^5 ×
     0.0001     0.0001     0.0001     0.0002     0.0002     0.0003     0.0003 ⋯
```

Notice that the above command provides the results in engineering notation misleading you to think that some elements are identical (*e.g.,* 0.0001). In fact, if you change the display format to **long** you will see more digits of the generated number.

```
v21 = logspace(1,5, 5)  % creates  5 elements between 10 and 100000
```

```
v21 = 1×5
          10          100         1000        10000       100000
```

Matrices

MATLAB matrices are similar in notion to 2D matrices in C++ and Java; albeit, much simpler to use. A matrix consists of both rows and columns. Therefore, a vector is a special case of a matrix that has either one row or one column.

To create a matrix in MATLAB, elements of each row are separated by a **comma** (,), and rows are separated by a **semicolon**. To save this matrix into variable M:

$$\begin{bmatrix} 2 & 4 & 10 & 3 \\ -9 & 0 & 6 & 7 \\ 1 & 4 & 12 & 6 \end{bmatrix}$$

```
M = [2 4 10 3; -9 0 6 7; 1 4 12 6]
```

```
M = 3×4
     2     4    10     3
    -9     0     6     7
     1     4    12     6
```

You can retrieve the dimensions of the matrix by using the **size** function. It will return the number of rows and column, respectively. In this case, the size of M is `3x4`.

```
size(M)
```

```
ans = 1×2
     3     4
```

To retrieve the total number of elements in the matrix, use the **numel** command (short for ***num**ber of **el**ements*). You can use this command with vectors as well, and in this specific case it will be equivalent to the **length** command .

```
numel(M)
```

```
ans = 12
```

Suppose we have two matrices:

$$K = \begin{bmatrix} 1 & 4 & 6 & 3 \\ -6 & 0 & 6 & 7 \end{bmatrix} \qquad L = \begin{bmatrix} 2 & 5 & 17 & 8 \\ -1 & 4 & 4 & 0 \end{bmatrix}$$

```
K = [1, 4, 6, 3 ; -6, 0, 6, 7]
```

```
K = 2×4
     1     4     6     3
    -6     0     6     7
```

```
L = [2, 5, 17, 8; -1, 4, 4, 0]
```

```
L = 2×4
     2     5    17     8
    -1     4     4     0
```

You can concatenate the two matrices horizontally or vertically only if they match each other in the number of elements along the side you wish to concatenate them in.

We can concatenate matrices K and L vertically in three different ways:

```
V1 = [K; L]             % Using the semicolon
```

```
V1 = 4×4
     1     4     6     3
    -6     0     6     7
     2     5    17     8
    -1     4     4     0
```

```
V2 = vertcat(K, L)      % Using the explicit vertical concatenation
```

```
V2 = 4×4
     1     4     6     3
    -6     0     6     7
     2     5    17     8
    -1     4     4     0
```

```
V3 = cat(1,K,L)         % Using the generic concatenate function. 1 means
vertical concatenate, 2 horizontal
```

```
V3 = 4×4
     1     4     6     3
    -6     0     6     7
     2     5    17     8
    -1     4     4     0
```

We can concatenate matrices K and L horizontally in three different ways:

```
H1 = [K, L]             % Using the colon
```

```
H1 = 2×8
     1     4     6     3     2     5    17     8
    -6     0     6     7    -1     4     4     0
```

```
H2 = horzcat(K, L)      % Using the explicit horizontal concatenation
```

```
H2 = 2×8
     1     4     6     3     2     5    17     8
    -6     0     6     7    -1     4     4     0
```

```
H3 = cat(2,K,L)         % Using the generic concatenate function. 1 means
vertical concatenate, 2 horizontal
```

```
H3 = 2×8
     1     4     6     3     2     5    17     8
    -6     0     6     7    -1     4     4     0
```

Vector and Matrix Indexing and Addressing

The most important rule which you must never forget **is that MATLAB indices always start from ONE, not 0.**

```
myVector = [2 : 0.5: 5]
```

```
myVector = 1×7
    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000    5.0000
```

To access the first, fifth, and last elements in this vector, one can write:

```
myVector(1)
```

```
ans = 2
```

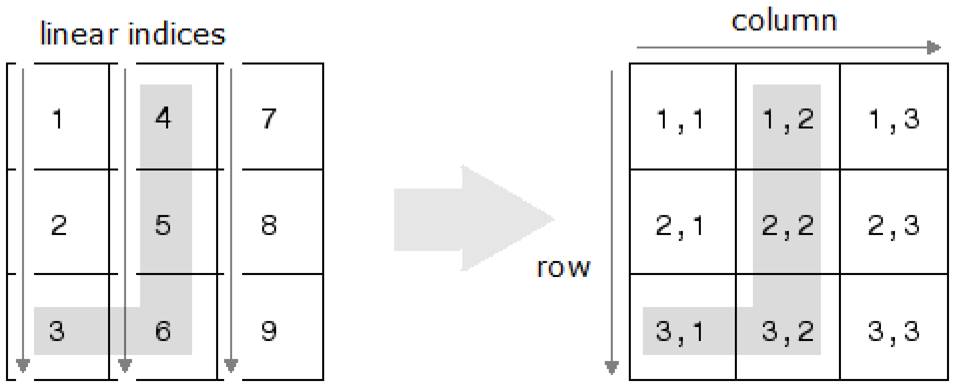
```
myVector(5)
```

```
ans = 4
```

```
myVector(numel(myVector))
```

```
ans = 5
```

You can retrieve any element in an array either using ***linear indexing*** or ***subscript indexing***. In linear indexing, you can think of the array as consecutive columns whose elements are numbered from 1 to n. Subscript indexing is similar to the one you use in C++ and Java, with the only difference that indices start from 1 not 0.



So, in the previous example with the matrices K and L, one can access the element 0 in matrix K by either:

```
K(4)
```

```
ans = 0
```

```
K(2,2)
```

```
ans = 0
```

To **replace** the last element in matrix L by 5, one can write:

```
L(numel(L)) = 5
```

```
L = 2×4
     2     5    17     8
    -1     4     4     5
```

```
L(2,4) = 5
```

```
L = 2×4
     2     5    17     8
    -1     4     4     5
```

The colon (:) operator is used to access a range of indices. For example, to retrieve all elements in matrix K, one can write:

```
K(:)
```

```
ans = 8×1
     1
    -6
     4
     0
     6
     6
     3
     7
```

To retrieve only the second row in matrix K, one can write:

```
K(2, :)     % This means access the second row, and retrieve all its elements.
```

```
ans = 1×4
    -6     0     6     7
```

To retrieve the middle two columns in array K:

```
K(:, 2:3)
```

```
ans = 2×2
     4     6
     0     6
```

This means access all columns from 2 to 3, and retrieve all rows.

To delete the last column in matrix L, one can write:

```
disp(L)
```

```
     2     5    17     8
    -1     4     4     5
```

```
L(:, 4) = []
```

```
L = 2×3
     2     5    17
    -1     4     4
```

This means, delete all elements in the fourth column.

Suppose the matrix A is given as: $A = \begin{bmatrix} 21 & 53 & 17 & 58 & 60 \\ 17 & 48 & 94 & 70 & 99 \\ 15 & 44 & 14 & 37 & 19 \\ 68 & 78 & 88 & 80 & 15 \\ 11 & 58 & 77 & 10 & 23 \\ 32 & 26 & 78 & 79 & 10 \end{bmatrix}$

```
A = [21, 53, 17, 58, 60; ...
     17, 48, 94, 70, 99; ...
     15, 44, 14, 37, 19; ...
     68, 78, 88, 80, 15; ...
     11, 58, 77, 10, 23; ...
     32, 26, 78, 79, 10]
```

```
A = 6×5
    21    53    17    58    60
    17    48    94    70    99
    15    44    14    37    19
```

```
        68      78      88      80      15
        11      58      77      10      23
        32      26      78      79      10
```

To access the inner elements without the border elements, one can write:

```
A(2:5, 2:4)
```

```
ans = 4×3
        48      94      70
        44      14      37
        78      88      80
        58      77      10
```

Also note how we used the ellipses (...) to write the array in a more eye-friendly way.

Special MATLAB Vectors and Matrices

MATLAB has functions to create special vectors and matrices that are useful in many instances.

The first matrix is an ALL-ones matrix. A matrix whose elements are initialized to the value 1. The second matrix is an ALL-zeros matrix whose elements are initialized to the value of 0.

You can create row vectors, column vectors, or matrices by passing the number of rows and columns to the function. If you pass one argument n, the function will generate a square matrix of size nxn.

```
ones(1,4)
```

```
ans = 1×4
         1       1       1       1
```

```
ones (3,1)
```

```
ans = 3×1
         1
         1
         1
```

```
ones(2,3)
```

```
ans = 2×3
         1       1       1
         1       1       1
```

```
ones(3)   % Supplying one input n creates a square array of that number nxn
```

```
ans = 3×3
         1       1       1
         1       1       1
         1       1       1
```

You can do the same as above using the **zeros** function:

```
zeros(1,4)
```

```
ans = 1×4
      0     0     0     0
```

```
zeros (3,1)
```

```
ans = 3×1
      0
      0
      0
```

```
zeros(2,3)
```

```
ans = 2×3
      0     0     0
      0     0     0
```

```
zeros(3)   % Supplying one input n creates a square array of that number nxn
```

```
ans = 3×3
      0     0     0
      0     0     0
      0     0     0
```

The identity matrix is a matrix whose diagonal is ones while all other elements are zero. Remember, that multiplying any SQAURE matrix with a SQAURE identity matrix results in the same original matrix. Use the **eye** function to create identity matrices.

```
eye (4)
```

```
ans = 4×4
      1     0     0     0
      0     1     0     0
      0     0     1     0
      0     0     0     1
```

You can also create rectangular identity matrices. In these matrices, columns whose elements don't fall on the diagonal are zeroed out.

```
eye(2, 3)
```

```
ans = 2×3
      1     0     0
      0     1     0
```

Block diagonal matrices combine multiple arrays together but aligns them diagonally. All remaining empty matrix locations are filled with zero. To do this in MATLAB:

```
A1 = ones(2,2);
A2 = 2*ones(3,2);
A3 = 3*ones(2,3);
B = blkdiag(A1,A2,A3)
```

```
B = 7×7
     1     1     0     0     0     0     0
     1     1     0     0     0     0     0
     0     0     2     2     0     0     0
     0     0     2     2     0     0     0
     0     0     2     2     0     0     0
     0     0     0     0     3     3     3
     0     0     0     0     3     3     3
```

A magic matrix is a special matrix with interesting mathematical properties:

- It is a square matrix of size $n \times n$, and $n \geq 3$
- All elements in the matrix fall between 1 and n^2
- All rows and all column elements sum to the same value

```
magic(3)
```

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

Matrix Manipulation

Similar to vectors, you can perform the matrix transpose operation where rows become columns.

For example:

$$M = \begin{bmatrix} 2 & 4 & 10 & 3 \\ -9 & 0 & 6 & 7 \\ 1 & 4 & 12 & 6 \end{bmatrix}$$

```
M = [2 4 10 3; -9 0 6 7; 1 4 12 6]
```

```
M = 3×4
     2     4    10     3
    -9     0     6     7
     1     4    12     6
```

And one can obtain its **transpose** as:

```
transpose(M)
```

```
ans = 4×3
     2    -9     1
     4     0     4
    10     6    12
     3     7     6
```

or simply by using the (') operator:

```
M'
```

```
ans = 4×3
     2    -9     1
     4     0     4
    10     6    12
     3     7     6
```

If you have a matrix and wish to create another matrix out of its replicas, you can use the **repmat** command. In this command, you specify how many times is the matrix replicated horizontally, then vertically, respectively.

```
repmat(M, 2, 3)
```

```
ans = 6×12
     2     4    10     3     2     4    10     3     2     4    10     3
    -9     0     6     7    -9     0     6     7    -9     0     6     7
     1     4    12     6     1     4    12     6     1     4    12     6
     2     4    10     3     2     4    10     3     2     4    10     3
    -9     0     6     7    -9     0     6     7    -9     0     6     7
     1     4    12     6     1     4    12     6     1     4    12     6
```

A similar command is repeat elements **repelem**. It works on vectors and matrices.

Suppose we have a small vector of three values $[1 \;\; 0.5 \;\; 0]$ and you wish to repeat each element three times:

```
repelem([1, 0.5 0], 3)
```

```
ans = 1×9
    1.0000    1.0000    1.0000    0.5000    0.5000    0.5000         0
0         0
```

To repeat each element in matrix M three times horizontally, and two times vertically, one can write:

```
repelem(M,3,2)
```

```
ans = 9×8
     2     2     4     4    10    10     3     3
     2     2     4     4    10    10     3     3
     2     2     4     4    10    10     3     3
    -9    -9     0     0     6     6     7     7
    -9    -9     0     0     6     6     7     7
    -9    -9     0     0     6     6     7     7
     1     1     4     4    12    12     6     6
     1     1     4     4    12    12     6     6
     1     1     4     4    12    12     6     6
```

You can reshape a matrix by using the reshape command which takes in as input a matrix, and the dimensions of its new shape.

```
reshape(M, 6, 2)
```

```
ans = 6×2
     2    10
    -9     6
     1    12
     4     3
     0     7
     4     6
```

However, the number of elements in the new shape must be the same as in the old matrix. Otherwise, MATLAB will issue an error.

```
reshape(M, 5, 3) % No. of elements 15 instead of 12, FAIL
```

The command **circshift (A, K, dim)** circularly shifts the elements in matrix ***A*** by ***K*** positions. The parameter ***dim*** specifies if one wishes to do the circular shift on the rows (dim = 1), or columns (dim = 2). These are the rules of how this command works:

- If K is positive, dim = 1, rows move downward
- If K is negative, dim = 1, rows move upward
- If K is positive, dim = 2, columns move left
- If K is negative, dim = 2, columns move right

```
circshift(M,  1, 1)
```

```
ans = 3×4
     1     4    12     6
     2     4    10     3
    -9     0     6     7
```

```
circshift(M, -1, 1)
```

```
ans = 3×4
    -9     0     6     7
     1     4    12     6
     2     4    10     3
```

```
circshift(M,  1, 2)
```

```
ans = 3×4
     3     2     4    10
     7    -9     0     6
     6     1     4    12
```

```
circshift(M, -1, 2)
```

```
ans = 3×4
     4    10     3     2
     0     6     7    -9
     4    12     6     1
```

You can rotate any vector or matrix **counter-clockwise** by 90° or its multiples by using the **rot90** command:

```
rot90(M, 1)     % Rotate M counter-clockwise by 90  degrees
```

```
ans = 4×3
     3     7     6
    10     6    12
     4     0     4
     2    -9     1
```

```
rot90(M, 3)     % Rotate M counter-clockwise by 90 x 3 = 270  degrees
```

```
ans = 4×3
     1    -9     2
     4     0     4
    12     6    10
     6     7     3
```

You can flip arrays either around their central row or column using two flip commands:

```
fliplr(M)       % Flip array left to right
```

```
ans = 3×4
     3    10     4     2
     7     6     0    -9
     6    12     4     1
```

```
flipud(M)       % Flip array yp to down
```

```
ans = 3×4
     1     4    12     6
    -9     0     6     7
     2     4    10     3
```

To extract the diagonal elements of the matrix, simply use the **diag** command:

```
diag(M)
```

```
ans = 3×1
     2
     0
    12
```

Sorting Vectors and Matrices

`sort(A)` sorts the elements of vector or matrix **`A`** in <u>**ascending**</u> order.

- If A is a vector, then sort(A) sorts the vector elements.
- If A is a matrix, then sort(A) treats the columns of A as vectors and sorts each column.

```
sort(M)
```

```
ans = 3×4
    -9     0     6     3
     1     4    10     6
     2     4    12     7
```

To sort the elements in each row, simply sort the transpose of a matrix:

```
sort(M')
```

```
ans = 4×3
     2    -9     1
     3     0     4
     4     6     6
    10     7    12
```

To sort the elements in descending order, the command slightly changes to **sort(A,'descend')**

```
sort(M,'descend')
```

```
ans = 3×4
     2     4    12     7
     1     4    10     6
    -9     0     6     3
```

Finally, an interesting sorting command is **sortrows**. Unlike the **sort** command, this command sorts the entire row block based on the values of the first column. If there is a tie, it decides based on the values of the second column, and so.

```
sortrows(M)
```

```
ans = 3×4
    -9     0     6     7
     1     4    12     6
     2     4    10     3
```

Vector and Matrix Mathematical Operations

The beauty of MATLAB is that it allows quick mathematical and logical operations on vectors and matrices.

Again, suppose we have the previous two matrices and a third square matrix:

$$K = \begin{bmatrix} 1 & 4 & 6 & 3 \\ -6 & 0 & 6 & 7 \end{bmatrix} \qquad L = \begin{bmatrix} 2 & 5 & 17 & 8 \\ -1 & 4 & 4 & 0 \end{bmatrix} \qquad P = \begin{bmatrix} 5 & 7 \\ 8 & 1 \end{bmatrix}$$

```
K = [1, 4, 6, 3 ; -6, 0, 6, 7]
```

```
K = 2×4
     1     4     6     3
    -6     0     6     7
```

```
L = [2, 5, 17, 8; -1, 4, 4, 0]
```

```
L = 2×4
     2     5    17     8
    -1     4     4     0
```

```
P = [5, 7; 8, 1]
```

```
P = 2×2
     5     7
     8     1
```

To add or subtract them together on an element-by-element basis, you simply write:

```
K + L
```

```
ans = 2×4
     3     9    23    11
    -7     4    10     7
```

```
K - L
```

```
ans = 2×4
    -1    -1   -11    -5
    -5    -4     2     7
```

We know from mathematics that matrix multiplication is different than regular multiplication. You cannot multiply any two matrices together unless they satisfy a matrix dimension restriction:

$$1^{st} array : n \times m$$
$$2^{nd} array : m \times l$$

That is the number of columns of the first array equals the number of rows in the second array. For example, we cannot perform matrix multiplication on the arrays ***K*** and ***L*** above as MATLAB will give an error because the size of each is 2×4.

```
K * L        % Error using * Incorrect dimensions for matrix
multiplication.
```

Yet, multiplying ***K*** by the transpose of ***L*** works fine because ***K*** has a size of 2×4, the transpose of ***L*** has the size of 4×2, and the result will have a size of 2×2.

```
K * L'
```

```
ans = 2×2
   148    39
   146    30
```

Remember matrix multiplication is not commutative. $AB \neq BA$. So, multiplying the transpose of ***L*** by ***K*** in this order will give a totally different result:

```
L' * K
```

```
ans = 4×4
     8     8     6    -1
   -19    20    54    43
    -7    68   126    79
     8    32    48    24
```

But what if you wish to do element-by-element multiplication, and not matrix multiplication. In this case, you must precede the * operator by a dot, so we have a new operator (.*)

```
K .* L
```

```
ans = 2×4
     2    20   102    24
     6     0    24     0
```

To do element-by-element operations on matrices, you must precede any operator by a dot. For example, dividing each element by 8:

```
K ./ 8
```

```
ans = 2×4
    0.1250    0.5000    0.7500    0.3750
   -0.7500         0    0.7500    0.8750
```

Or raising each element to a power of 2 or 4:

```
K .^ 2
```

```
ans = 2×4
     1    16    36     9
    36     0    36    49
```

```
K .^ 4
```

```
ans = 2×4
           1         256        1296          81
        1296           0        1296        2401
```

However, if you write:

```
K ^ 2
```

It will flag an error because this means you are multiplying $K \times K$ and their dimensions do not agree. You can only do this with square matrices:

```
P ^ 2
```

```
ans = 2×2
    81    42
    48    57
```

```
P ^ 4
```

```
ans = 2×2
        8577        5796
        6624        5265
```

What if the array elements are the powers that you wish to raise a scalar to, say we want to raise the number 4 to the power of elements in matrix K.

```
4 .^ K
```

```
ans = 2×4
    0.0004    0.0256    0.4096    0.0064
    0.0000    0.0001    0.4096    1.6384
```

You cannot raise a matrix to a matrix:

```
K ^ P % Wrong
```

What if you want to multiply the square root of each element in array K by the log_{10} of each corresponding element in array L:

```
sqrt(K) .* log10(L)
```

```
ans = 2×4
    0.3010    1.3979    3.0140    1.5642
   -3.3420         0    1.4747      -Inf
```

In linear algebra courses, you learnt of the inverse of a matrix. MATLAB has a special inverse function called **inv.** One can compute the inverse of a matrix only if it is square.

This would **not** work:

```
inv(K)
inv(L)
```

But this does:

```
inv(P)
```

```
ans = 2×2
   -0.0196    0.1373
    0.1569   -0.0980
```

A very useful command is the **sum** command. It sums all the values in a vector, or if the input is a matrix, it sums all the values in each column.

```
sum([7 8 9 5 0])
```

```
ans = 29
```

```
sum(K)
```

```
ans = 1×4
    -5     4    12    10
```

Another useful command is the **prod** command. It multiplies all the values in a vector, or if the input is a matrix, it multiplies all the values in each column.

```
prod([7 8 9 5 1])
```

```
ans = 2520
```

```
prod(K)
```

```
ans = 1×4
    -6     0    36    21
```

Vector and Matrix Logical Operations

Like all other programming languages, MATLAB supports logical and relational operators. We list them in the following table.

Logical and Relational Operators	
`==`	Relational operator: equal to.
`~=`	Relational operator: not equal to.
`<`	Relational operator: less than.
`<=`	Relational operator: less than or equal to.
`>`	Relational operator: greater than.
`>=`	Relational operator: greater than or equal to.
`&`	Logical operator: AND.
`|`	Logical operator: OR.
`~`	Logical operator: NOT.
`xor`	Logical operator: EXCLUSIVE OR.

When you use these operations, the output is either 0 (false) or 1 (true). These are not numeric 0 or 1, but logical values. In the same way, if we compare matrices using relational operators, the output matrix of $0s$ and $1s$ is not numeric, but logical.

These operators work on an **element-by-element basis.**

Again, suppose we have:

$$K = \begin{bmatrix} 1 & 4 & 6 & 3 \\ -6 & 0 & 6 & 7 \end{bmatrix} \qquad L = \begin{bmatrix} 2 & 5 & 17 & 8 \\ -1 & 4 & 4 & 0 \end{bmatrix}$$

```
K = [1, 4, 6, 3 ; -6, 0, 6, 7]
```

```
K = 2×4
     1     4     6     3
    -6     0     6     7
```

```
L = [2, 5, 17, 8; -1, 4, 4, 0]
```

```
L = 2×4
     2     5    17     8
    -1     4     4     0
```

One can check if each element in ***K*** is less than each corresponding element in ***L*** by writing:

```
R = K < L
```

```
R = 2×4 logical array
   1   1   1   1
   1   1   0   0
```

Check for the type (class) of the resulting matrix by typing:

```
whos R
```

```
  Name      Size            Bytes  Class      Attributes

  R         2x4                 8  logical
```

You can extract the numbers that meet a specific criterion using relational operators easily. For example, to store the numbers that are less than 5 in array K in a new matrix, simply write:

```
Q = K(K < 5)
```

```
Q = 5×1
     1
    -6
     4
     0
     3
```

The step K < 5 first compares each element in K if it is less than five or not and it returns a logical array of 0's and 1's. This array is passed to K () which retrieves the corresponding elements for each one that appears in the logical array.

If you want to retrieve the linear index of the elements that are less than 5 in array K, use the **find** function:

```
find(K < 5)
```

```
ans = 5×1
     1
     2
     3
     4
     7
```

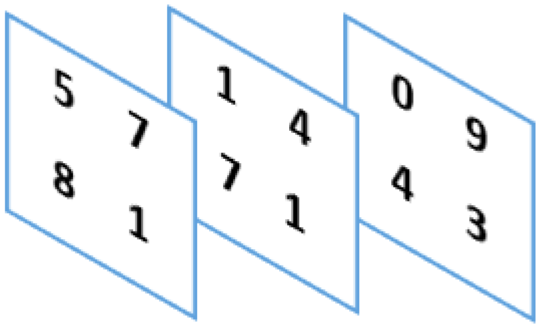
The step K < 5 first compares each element in K if it is less than five or not and it returns a logical array of 0's and 1's. This array is passed to the **find** function which retrieves the linear index for each one that appears in the logical array.

Multidimensional Matrices

3D matrices consist of *pages* of 2D matrices. Suppose one has:

$$A = \begin{bmatrix} 5 & 7 \\ 8 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 4 \\ 7 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 0 & 9 \\ 4 & 3 \end{bmatrix}$$

and that we wish to store them as a 3D matrix of three pages as follows:



One can construct a 3D matrix by first storing the 2D matrices, then appending them:

```
A = [5, 7; 8, 1]
```

```
A = 2×2
      5     7
      8     1
```

```
B = [1, 4; 7, 1]
```

```
B = 2×2
      1     4
      7     1
```

```
C = [0, 9; 4, 3]
```

```
C = 2×2
      0     9
      4     3
```

```
A (:, :, 2) = B
```

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

     5     7
     8     1


A(:,:,2) =

     1     4
     7     1
```

The previous commands append the second array B to the first array A creating a 3D array.

```
A (:, :, 3) = C
```

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

     5     7
     8     1


A(:,:,2) =

     1     4
     7     1


A(:,:,3) =

     0     9
     4     3
```

The previous commands append the second array C to the 2D array of A and B expanding the 3D array.

Addressing 3D matrices is simple. As before, using **subscript addressing**, define the row and column location in this respective order, then finally specify the page.

For example, to retrieve the number 9 on the third page:

```
A(1,2,3)
```

```
ans = 9
```

To get the whole second page:

```
A(:, :, 2)
```

```
ans = 2×2
     1     4
     7     1
```

Data Import and Pre-processing

MATLAB has extremely powerful tools to import and export data whether it is text, spreadsheets, audio, video, images, or data stored in special scientific formats. In this experiment, we will focus on local MATLAB files (**`.mat`**), text files **`(.txt, .csv)`**, and spreadsheets **`(.xls, .xlsx)`**.

Loading and Storing MATLAB Variables

You can save all your workspace variables that you have created in a MATLAB session into a special file with the extension **`.mat.`** You can load these variables again at the start of your next session and start from where you last stopped. This is done through the simple **`save`** and **`load`** commands. You must specify a file name (as a string or character array), and *optionally* followed by the names of the variables you want to save/load. If you do not specify any variables, the commands will save or load **ALL** variables.

```
save("exp3_all_variables.mat")
save('exp3_some_variables.mat', "s1", "s2", "s3")
```

But where are your variables stored? The variables are stored in the current working path. You can see the working path right under the ribbon (and you can change it from there too). Alternatively, you can write the "**p**rint **w**orking **d**irectory" command:

```
pwd
```

```
ans = 'D:\Google Drive\UJ-Courses\CPE213-Numerical_Analysis\Experiments'
```

To load your MATLAB work session variables, use the **`load`** command which has an identical syntax.

```
load("exp3_all_variables.mat")
load('exp3_some_variables.mat', "s1", "s2", "s3")
```

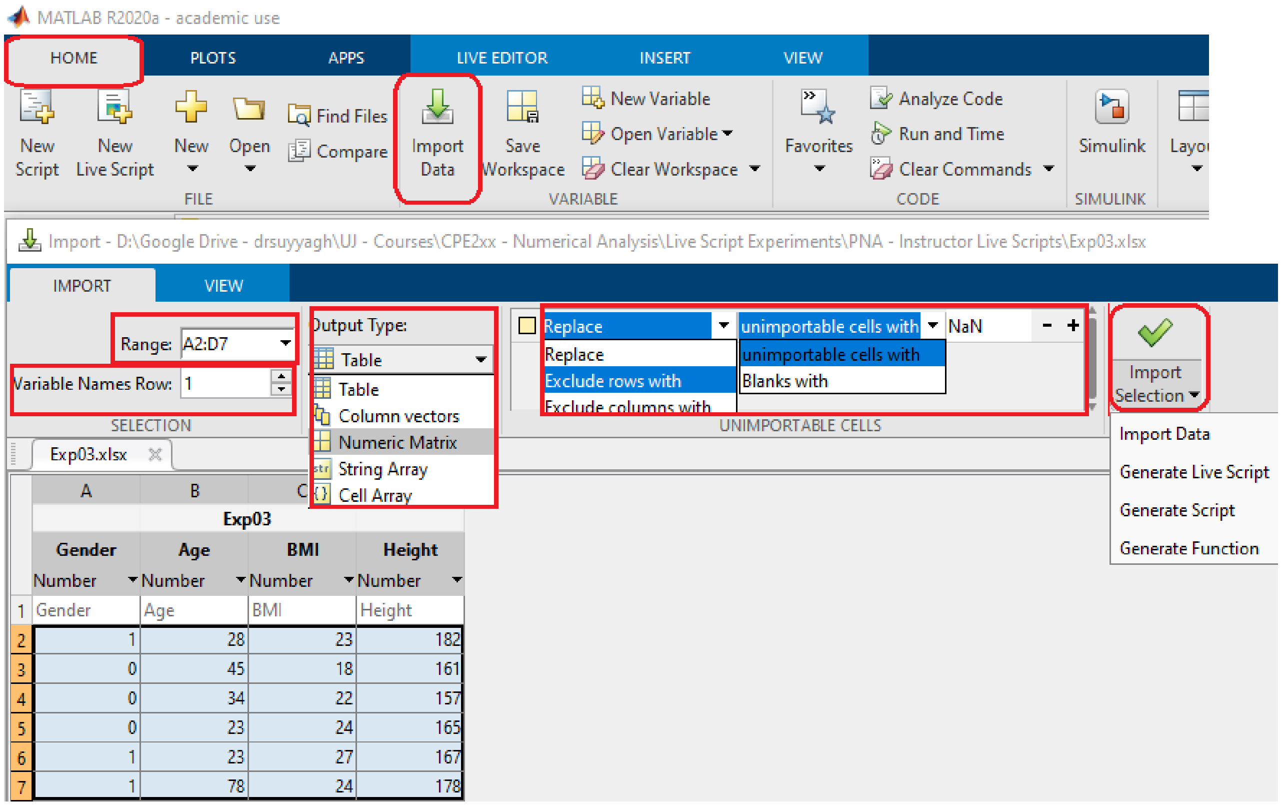
Using the Interactive Import Tool

Many times, numeric data is stored in textfiles or Excel spreadsheets. You can use MATLAB commands that we introduced in the previous section. However, if you are working with one or two files, then you can use MATLAB's **`Import Data`** tool to import data into MATLAB's workspace using a GUI tool.

1. You can access the tool from the **Home** ribbon, then Import tool.
2. A simple Open File window will pop out, choose the file you want to import (`.txt, .csv. .xlsx,` ... etc). There are different options for Spreadsheets and Text Files
3. A new Import screen opens where you select the import settings. Basically, the settings are identical, except for text files, you get the extra option to choose the delimiter that splits the data into columns.

Importing Spreadsheets using the GUI tool

1. By default, it selects the entire range of the spreadsheets' cells. You can change the range to import a subset of this range.
2. Specify the row number that holds the text data (*e.g.,* Column headings). This helps MATLAB exclude it if you are importing it into a **numeric** array.
3. Always make sure to import your data into a **<u>Numeric matrix</u>**, or a **<u>String Array</u>** if you want to import the data as text.
4. Choose the import behaviour if your input data has problems, say if it has empty cells, or characters instead of numbers. You can replace these values with ***NaN*** or write any other value you want, or you can exclude the rows or columns that has these erroneous cells. It all depends on your application and what you want to do.
5. Once you are done, click Import Selection, you can either simply save the data or generate the commands that import the data.



Import - D:\Google Drive - drsuyyagh\UJ - Courses\CPE2xx - Numerical Analysis\Live Script Experiments\PNA - Instructor Live Scripts\Exp03.xlsx

Range: A2:D7

Variable Names Row: 1

Output Type: Table

Replace unimportable cells with NaN

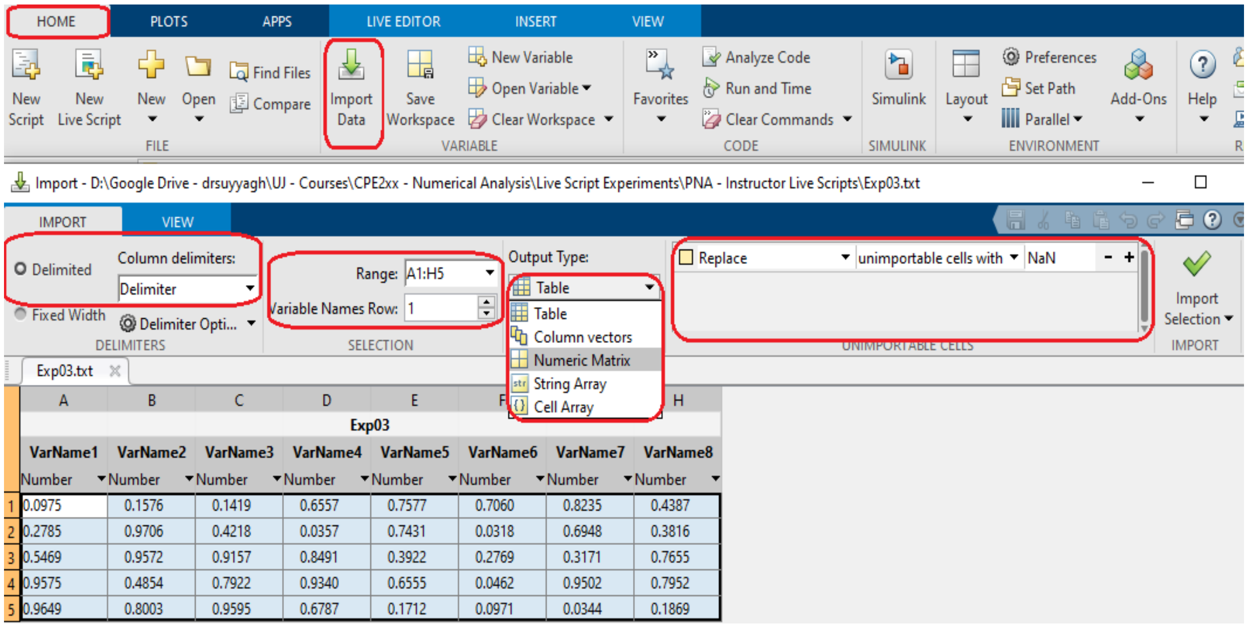
Exp03.xlsx

	A	B	C	D
	Exp03			
	Gender	Age	BMI	Height
	Number	Number	Number	Number
1	Gender	Age	BMI	Height
2	1	28	23	182
3	0	45	18	161
4	0	34	22	157
5	0	23	24	165
6	1	23	27	167
7	1	78	24	178

Importing Text Files using the GUI Tool

1. By default, the tool will try to find the delimiter (space, tab, comma ... etc) that best separates the text file into columns.
2. You can select the entire range of the data, or you can change the range to import a subset of this range.
3. Specify the row number that holds the text data (*e.g.,* Column headings). This helps MATLAB exclude it if you are importing it into a **numeric** array.
4. Always make sure to import your data into a **<u>Numeric matrix</u>**, or a **<u>String Array</u>** if you want to import the data as text.

5. Choose the import behaviour if your input data has problems, say if it has empty cells, or characters instead of numbers. You can replace these values with ***NaN*** or write any other value you want, or you can exclude the rows or columns that has these erroneous cells. It all depends on your application and what you want to do.
6. Once you are done, click Import Selection, you can either simply save the data or generate the commands that import the data.



Import - D:\Google Drive - drsuyyagh\UJ - Courses\CPE2xx - Numerical Analysis\Live Script Experiments\PNA - Instructor Live Scripts\Exp03.txt

Delimited [✓] Fixed Width
Column delimiters: Delimiter
Range: A1:H5
Variable Names Row: 1
Output Type: Table
Replace unimportable cells with NaN

	A	B	C	D	E	F	G	H
	Exp03							
	VarName1	VarName2	VarName3	VarName4	VarName5	VarName6	VarName7	VarName8
	Number	Number	Number	Number	Number	Number	Number	Number
1	0.0975	0.1576	0.1419	0.6557	0.7577	0.7060	0.8235	0.4387
2	0.2785	0.9706	0.4218	0.0357	0.7431	0.0318	0.6948	0.3816
3	0.5469	0.9572	0.9157	0.8491	0.3922	0.2769	0.3171	0.7655
4	0.9575	0.4854	0.7922	0.9340	0.6555	0.0462	0.9502	0.7952
5	0.9649	0.8003	0.9595	0.6787	0.1712	0.0971	0.0344	0.1869

It is worth noting that the GUI import tool can be used to import many other data formats such as audio and video files, different types of images and other datafiles.

Experiment version 1.2
Original Experiment October 5th, 2020
Last Updated March 11th, 2022
Dr. Ashraf Suyyagh - All Rights Reserved

Revision History

Ver. 1.2
Moved the Data Import and Pre-processing Part from the old Fundamentals III into this experiment
Ver. 1.1
Corrected formatting and spelling mistakes.
Corrected the equation that computes the number of generated elements in a vector by added the floor symbol.
Added more info about the display format of the *logspace* command.
Added more clarification on retrieving values or indices using logical operations.
Added more clarification to appending 2D arrays to form 3D arrays.

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 3 - Scripts, Functions, and Control Flow Operations

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Scripts

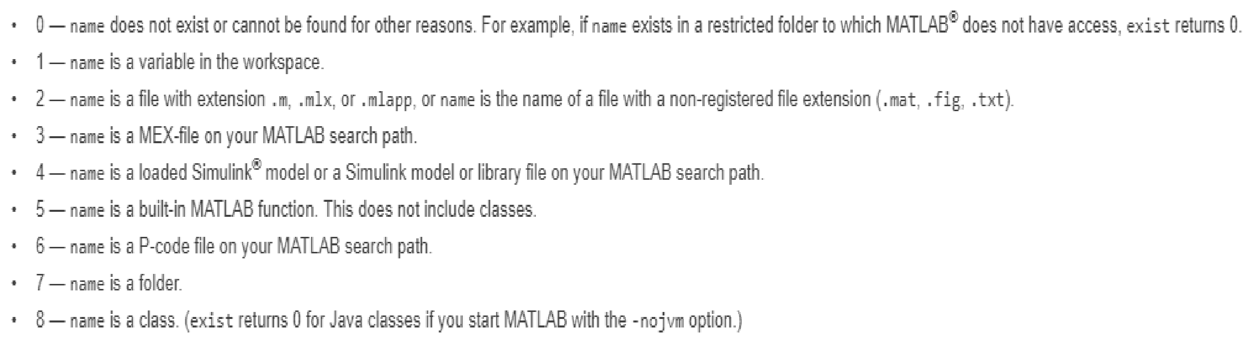
In the past few experiments, we have used MATLAB exclusively in interactive mode. You interacted with MATLAB by writing instructions inside the command window and seeing results in the same window. While interactive mode works well when one needs to perform quick calculations, for larger codes this might not be feasible. We resort to MATLAB scripts instead to write, store, debug, and execute longer codes with specific purposes much like any other programming language. Scripts save you time by allowing you to change a command, modify or fix your code in place rather than type everything every over and over in the command window.

Script Creation and Access

To create a MATLAB script file, simple go to `New --> Script File` and an `untitled` script will show up immediately in MATLAB workspace. When you save the script file, you must take precautions in naming your script to avoid collisions. MATLAB scripts have an extension of ***'filename.m'***. The rules for naming MATLAB script files are:

- The name of a script file must begin with a letter, and may include digits and the underscore character, up to 31 characters.
- Do not give a script file the same name as a variable, MATLAB command, or MATLAB function or other script files.

Previously, we have used the **`exist`** command to check if a variable exists in the workspace or not. We can use the **`exist`** command again to check if the name we want to choose for our script is in no conflict with variables, other files, commands, *etc.* If you type **`exist`** with a potential script name, it must return zero before proceeding. A non-zero value means your suggested script name is already in use for something else as given in the list below:



- 0 — `name` does not exist or cannot be found for other reasons. For example, if `name` exists in a restricted folder to which MATLAB® does not have access, `exist` returns 0.
- 1 — `name` is a variable in the workspace.
- 2 — `name` is a file with extension `.m`, `.mlx`, or `.mlapp`, or `name` is the name of a file with a non-registered file extension (`.mat`, `.fig`, `.txt`).
- 3 — `name` is a MEX-file on your MATLAB search path.
- 4 — `name` is a loaded Simulink® model or a Simulink model or library file on your MATLAB search path.
- 5 — `name` is a built-in MATLAB function. This does not include classes.
- 6 — `name` is a P-code file on your MATLAB search path.
- 7 — `name` is a folder.
- 8 — `name` is a class. (`exist` returns 0 for Java classes if you start MATLAB with the `-nojvm` option.)

When you save the script, you can save it anywhere but when you run it, make sure that the script is in the current path. Remember you can use the function **`pwd`** to print the current path. You can save your scripts in other folders or subdirectories, but in this case, you must add these folders to the path using the command **`addpath.`**

For example, to create a subdirectory in the current directory named *'myMATLAB'* and add it to the path, one can write:

```
mkdir('myMATLAB')
```

```
If the directory already exists, it will give "Warning: Directory already exists."
```

```
addpath('myMATLAB')
```

To check if your newly created folder (or any folder for that matter) is in MATLAB's path, simply check by using the command **`path`**:

```
path
```

```
		MATLABPATH
	D:\Google Drive - drsuyyagh\UJ - Courses\CPE2xx - Numerical Analysis\Live Script
Experiments\PNA - Instructor Live Scripts\myMATLAB
	C:\Users\drsuyyagh\Documents\MATLAB
	C:\Users\drsuyyagh\AppData\Local\Temp\Editor_vxdxq
	C:\Program Files\MATLAB\R2020a\toolbox\matlab\capabilities
	C:\Program Files\MATLAB\R2020a\toolbox\matlab\datafun
```

```
C:\Program Files\MATLAB\R2020a\toolbox\matlab\datatypes
C:\Program Files\MATLAB\R2020a\toolbox\matlab\elfun
…, etc.
```

Now let us create a script named **`'firstScript.m'`** and save it inside *'myMATLAB*' folder. Inside the file, write the command:

```
disp('This is my first script')
```

```
This is my first script
```

To run the script, simply call it by its name. The content of the script will execute as long as you have added its location to MATLAB's path.

```
firstScript
```

Writing Scripts

Inside your script, write your sequence of commands as if you are writing them using the interactive mode inside the command window. However, we usually prefer to do the following:

- use the semicolon (;) to suppress the output of commands. We are usually interested in the **final result**, not the output of the intermediate steps.
- Use the % to insert comments to explain the inputs, outputs, functionality, *etc.*

Sometimes, you need to ask the user for input; you can use the **`input`** command to prompt users for input through the command window, and specify whether the required input is to be stored as a *numeric* value or a *string*. Don't forget to suppress the statements by a semicolon at the end.

```
x = input ('Please Enter your Name: ', 's') % Add 's' at the end of the input command to specify a string input
```

```
x = 'Ashraf'
```

```
y = input ('Please Enter your Age: ')  % By default, inputs accepts numerical values
```

```
y = 35
```

To display text messages, or the value of variables to the user on the command window, as usual, use the **`disp`** command:

```
disp('We are becoming good users of MATLAB!')
```

```
We are becoming good users of MATLAB!
```

Let us update our ***`firstScript.m`*** file that we have created inside the *myMATLAB* folder by doing something fun!

Suppose we want to plot the following equations on the Cartesian Coordinate system. Initially, we want to generate the **(x,y)** pairs using this equation in order to plot it, how would we approach this problem?

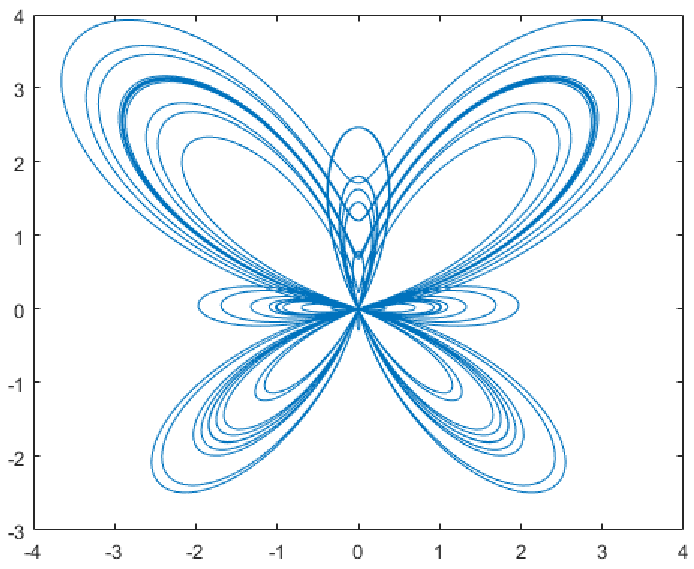
$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \sin(t) \cdot \left(\left(e^{\cos(t)} - 2\cos(4t) - \left(\sin\left(\frac{t}{12}\right)^5 \right) \right) \right) \\ \cos(t) \cdot \left(e^{\cos(t)} - 2\cos(4t) - \left(\sin\left(\frac{t}{12}\right)^5 \right) \right) \end{bmatrix}$$

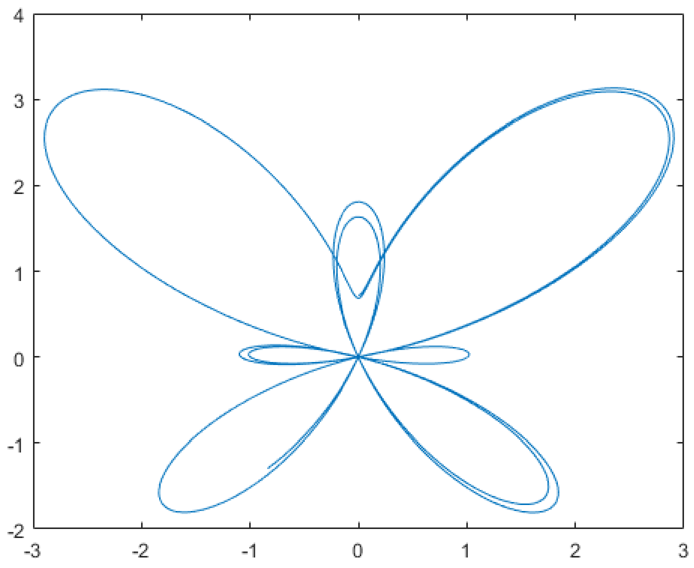
1. In this example. we need to generate the values of **x**, and the associated values of **y**. Both **x** and **y** equations are given in terms of **t**. If you use one value for **t**, this means you will only get one value for **x**, and another one for **y**, so you end up with only one pair of **(x,y)**, which is basically a dot!
2. We conclude that we need to generate numerous pairs of **(x,y)** such that we end up with a meaningful plot. To do so, we need to generate numerous values of **t** to substitute in the equations above.
3. We already learnt that we can generate a vector of values either using the **colon** expression or the **linspace** command. But we need to know where to stop? In our case, we don't really know how many pairs are enough, so lets ask the user to provide the end point.
4. The following sequence of MATLAB commands corresponds to the steps mentioned above. Copy/Paste this sequence into ***firstScript.m*** file after deleting its old content.

```
endpoint = input('Please provide a value of t: ');
t = 0:0.01:endpoint;
x = sin(t).*(exp(cos(t)) - 2.*cos(4*t) - (sin(t/12)).^5);
y = cos(t).*(exp(cos(t)) - 2.*cos(4*t) - (sin(t/12)).^5);
plot(x, y)
```

Notice that we have suppressed the output of each step and only visualized the plot. To execute the script, simply write its name. As long as the script location can be found in MATLAB's path, MATLAB will find it and run it. Run the script a couple of times times. Try the value 100 and 10 for **t**.

```
firstScript
```





Functions and Subfunctions

In MATLAB, aside from the ability to write scripts, you can write another type of files called function files. A function file is a special type of an `.m` file that contains one **primary** function and the possibility of having one or more subfunctions. Subfunctions are optional. You can use them if you have a very large primary function that you wish to divide into smaller modular functions. These are some important rules when writing function files:

1. The **primary** function is always the **first** function in your file.
2. Your filename and primary function name must be **identical**.
3. Valid function names begin with an alphabetic character, and can contain letters, numbers, or underscores.

4. You can call the function from the command window or any other **.m** file, as long as the file is in MATLAB's path. Use **addpath** to add the folder containing the function file to MATLAB's path.
5. You can only call the primary function. You cannot call any of the subfunctions. They are not visible outside of the file. Only the main function can access them.
6. Any function can have zero or more inputs, and zero or more outputs.
7. All variables defined in the primary function or the subfunctions are **local** to that function. You cannot access them from outside and they get **erased** once the function finishes execution; unless it is the output variable!
8. In MATLAB 2016b or later, you can place functions at the end of a script file, but in this course, we will not cover this.

Defining MATLAB Functions

A MATLAB function has the following definition:

```
function [y1,...,yN] = myfun(x1,...,xM)
..
..
end
```

*"It declares a function named **myfun** that accepts inputs x1,...,xM and returns outputs y1,...,yN. This declaration statement must be the first executable line of the function".* You must name the file as **myfun.m**

Note that the output variables are enclosed in ***square brackets***. The input variables must be enclosed with ***parentheses***.

The following are valid function definitions:

Function definition	filename	Outputs	Inputs
function [area] = squareArea(*side*) function area = squareArea(*side*)	squareArea.*m*	area	*side*
function [volume] = boxVol(*height*, *width*, *length*) function volume = boxVol(*height*, *width*, *length*)	boxVol.*m*	volume	*height*, *width*, *length*
function [area, circumference] = circle(*radius*)	*circle.m*	area, circumference	*radius*
function sqplot(*side*)	*sqplot.m*		*side*

Creating MATLAB Functions

To create a MATLAB function, simply go to **New --> Function** and it will create a template that you can edit according to your requirements. The template will look like this:

```
function [outputArg1,outputArg2] = untitled3(inputArg1,inputArg2)
%UNTITLED3 Summary of this function goes here
%   Detailed explanation goes here
outputArg1 = inputArg1;
outputArg2 = inputArg2;
end
```

Start by first changing the function name from **`untitled`** to a meaningful name to call your function by. Don't forget to save your filename exactly as the function name. This first function is your primary function. Don't forget to add the folder this file exists in to MATLAB' path by using the command **`addpath`**.

Now, edit your input and output arguments and the body of your function as you need. For example, create a function called **`zedSquares`** that takes in the arguments **`x`** and **`y`** , and returns the sum of their squares in **`z`**. Always try to write meaningful comments detailing what the expected input is, and what does the function do and its output. Your function should look like this:

```
 function [z] = zedSquares(x, y)
 % zedSquares: This function comptes the sum of the sqaure of inputs x
and y and stores them in z.
 z = x^2 + y^2;
 end
```

Try calling your function as follows:

```
m = 7;
n = 9;
```

```
n = 9
```

```
z = zedSquares(m, n)
```

```
z = 130
```

What if the inputs to the function **zedSquares** were vectors or matrices? Our expectation is that it should perform the operation element-wise. But the function call will fail. Try it.

```
m = [7, 5, 9, 0, 1];
n = [2, 2, 3 8, 6 ];
z = zedSquares(m, n)
```

The reason behind this is we did not write our function body to be generic; that is, to take in the possibility that the user might enter a scalar, a vector, or a matrix. A simple fix would be to use element-wise operators. Update and save the function **zedSquares** with the following body**:**

```
z = x.^2 + y.^2;
```

then run the code below again:

```
m = [7, 5, 9, 0, 1];
n = [2, 2, 3 8, 6 ];
z = zedSquares(m, n)
```

```
z = 1×5
    53    29    90    64    37
```

Creating MATLAB Subfunctions

MATLAB subfunctions are functions that are written after the primary function. They can't be called outside the file, and only called by the primary function or each other.

Let us create a new function file and call it ***circleParameters.m*** inside the folder *myMATLAB*. Copy the following code inside the newly created file:

```
% The primary function in the file circleParameters.m
function [area,circumference] = circleParameters(radius)
    area          = computeArea(radius) ;
    circumference = computeCircumference(radius) ;

% A subfunction to compute the area
function a = computeArea(r)
a = pi.*r.^2;

% A subfunction to compute the circumference
function c = computeCircumference(r)
c = 2*pi.*r;
```

In the above code, the primary function takes the radius as an input, then calls the subfunctions **computeArea** and **computeCircumference.** Each one of them computes its respective values which are stored eventually into **area** and **circumference**; the final outputs of the primary functions.

Note that when you call the primary function **circleParameters**, don't forget to store its output or otherwise you won't make good use of it in future operations.

```
radii = [4, 5, 9];
[areas, circumferences] = circleParameters(radii)
```

```
areas = 1×3
   50.2655   78.5398  254.4690
circumferences = 1×3
   25.1327   31.4159   56.5487
```

Global Variables

"*Ordinarily, each MATLAB function has its own local variables, which are separate from those of other functions and from those of the base workspace. However, if several functions all declare a particular variable name as **global**, then they all share a single copy of that variable. Any change of value to that variable, in any function, is visible to all the functions that declare it as global.*"

Let us create a new function file and call it ***testGlobals.m*** inside the folder *myMATLAB*. Copy the following code inside the newly created file:

```
function testGlobals (x, y)
global z
z = 10;
disp(z)
inner1 (y);
disp(z)
inner2 (x);
```

```
disp(z)
end

function inner1 (y)
global z
z = z - y;
end

function z = inner2 (x)
z = 5 + x;
end
```

Once you are done, call the primary function **`testGlobals:`**

```
testGlobals (2, 3)
```

```
    10

     7

     7
```

We observe that the variable **`z`** is a global variable in only the primary function and **`inner1`** function, but **NOT** **`inner2`** function where it is a local variable. So initially, **`z`** has a value of 10. When we call **`inner1`**, the value of **`z`** changes globally to become 7. However, when we call **`inner2`**, there **`z`** is a local variable, it becomes 4 and the function returns with the value 4, but since it is not stored anywhere, its value is lost. Global **`z`** value is not affected and remains 7.

Persistent Variables

*"Persistent variables are **local** to the function in which they are declared, yet their values are retained in memory between calls to the function"*. That is, when the function finishes execution, it does not clear the variable. Further, you cannot change the value of the persistent variable from MATLAB's command line or from within other functions. By default, persistent variables are initialized to an empty vector.

Let us create a new function file and call it ***`testPersistent.m`*** inside the folder *myMATLAB*. Copy the following code inside the newly created file:

```
function testPersistent()
persistent n
    if isempty(n)
        n = 0;
        disp(n)
    end
    n = n+1;
    disp(n)
end
```

Once you are done, call the primary function **`testPersistent`** few times:

```
testPersistent
```

```
        0

        1
```

```
testPersistent
```

```
        2
```

```
testPersistent
```

```
        3
```

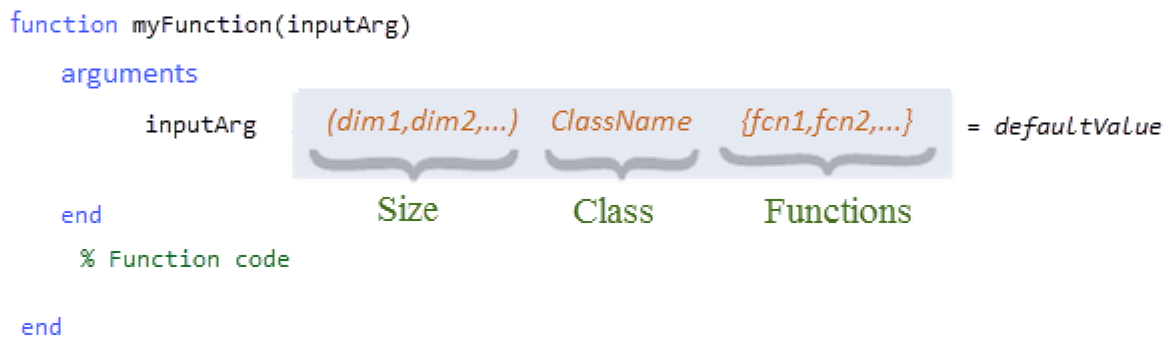
```
testPersistent
```

```
        4
```

Once declared, the persistent variable **`n`** is an empty vector. The **`if`** statement tests if **`n`** is empty and if so initializes it to 0, so it becomes a scalar with the value 0. Then, it gets incremented by 1 and the function exits. Upon the next call, the persistent variable **`n`** was not cleared, and it still retains its previous value of 1, so the function increments it to 2. In the third call, its value is updated to 3 and so on.

Function Arguments Validation

In many times, you want to make sure that your function accepts numeric values only (no strings, NaN, or Inf), or you want to make sure that it accepts vectors but not arrays, or accepts arrays of certain fixed dimensions, or that the values passed satisfy a certain criteria. You can make these validation checks at the beginning of your function using the **`arguments`** and **`end`** keywords. Function validation has the following syntax:



```
function myFunction(inputArg)
    arguments
        inputArg    (dim1,dim2,...) ClassName {fcn1,fcn2,...} = defaultValue
    end
     % Function code
 end
```

After you define your function, and before its main body you can add the checks necessary using the **`arguments`** keyword. The function argument declaration can include any of these kinds of restrictions:

- **Size**: The length of each dimension, enclosed in parentheses. For example (1, 1) means it accepts scalars, (1, :) can accept either vertical or horizontal vectors, (3:0) means the first dimension must be 3, and second dimension can be any value.
- **Class**: can be *char*, *double*, *string*, *etc*.
- **Validation Functions**: A comma-separated list of validation functions, enclosed in curly braces. You can choose your functions from the list below:

Name	Meaning
`mustBePositive(value)`	`value > 0`
`mustBeNonpositive(value)`	`value <= 0`
`mustBeFinite(value)`	`value` has no `NaN` and no `Inf` elements.
`mustBeNonNan(value)`	`value` has no `NaN` elements.
`mustBeNonnegative(value)`	`value >= 0`
`mustBeNegative(value)`	`value < 0`
`mustBeNonzero(value)`	`value ~= 0`
`mustBeGreaterThan(value,c)`	`value > c`
`mustBeLessThan(value,c)`	`value < c`
`mustBeGreaterThanOrEqual(value,c)`	`value >= c`
`mustBeLessThanOrEqual(value,c)`	`value <= c`
`mustBeNonempty(value)`	`value` is not empty.
`mustBeNonsparse(value)`	`value` has no sparse elements.
`mustBeNumeric(value)`	`value` is numeric.
`mustBeNumericOrLogical(value)`	`value` is numeric or logical.
`mustBeReal(value)`	`value` has no imaginary part.
`mustBeInteger(value)`	`value == floor(value)`
`mustBeMember(value,S)`	`value` is an exact match for a member of `S`.

Let us write a function that has three arguments **a**, **b** and **c**. We wish to restrict **a** to be a positive scalar, **b** to be a vector with no NaN or Inf values, and **c** to be an array whose elements are larger than 10. All variables must be of course numeric.

Let us create a new function file and call it ***testValidation.m*** inside the folder *myMATLAB*. Copy the following code inside the newly created file:

```
function testValidation (a, b, c)
    arguments
        a (1, 1) double  {mustBeNumeric, mustBePositive }
        b (1,:)  double  {mustBeNumeric, mustBeFinite}
        c (:,:)  double  {mustBeNumeric, mustBeGreaterThan(c, 10)}
    end

    disp(a)
    disp(b)
    disp(c)
end
```

To test how functions with arguments validation works, let us try few examples:

```
a = 7;
b = [12, 14, 17];
c = [12, 15,; 19, 11];
testValidation (a, b, c)
```

```
     7

    12    14    17

    12    15
    19    11
```

Suppose we change the value of **a** to -7:

```
a = -7;
b = [12, 14, 17];
c = [12, 15,; 19, 11];
testValidation (a, b, c)
```

You will get an error telling you the restriction that you have imposed:

```
Error using testValidation
Invalid argument at position 1. Value must be positive.
```

Similarly, if we change the value of **b** to have an **`Inf`**:

```
a = 7;
b = [12, 14, Inf];
c = [12, 15,; 19, 11];
testValidation (a, b, c)
```

You will get an error telling you the restriction that you have imposed:

```
Error using testValidation
Invalid argument at position 2. Value must be finite.
```

And finally, if we change the value of 11 into 9 in variable c:

```
a = 7;
b = [12, 14, 17];
c = [12, 15,; 19, 9];
testValidation (a, b, c)
```

You will get an error telling you the restriction that you have imposed:

```
Error using testValidation
Invalid argument at position 3. Value must be greater than 10.
```

Function Handles and Anonymous Functions

Function handles are created by simply preceding the function name by the **@** sign. So, the handle becomes like a pointer to that function which allows you to pass the function as an argument to other functions or create what we call a cell array of function handles.

A cell array is special type of MATLAB arrays that can be used to store elements of different types together, unlike numeric or string arrays, they can contain numbers, strings, or function handles, *etc.* We can create a cell array of trigonometric function handles as follows:

```
trigfun = {@sin, @cos, @tan}
```

```
trigfun = 1×3 cell
```

	1	2	3
1	1×1 function_handle	1×1 function_handle	1×1 function_handle

Note that we use ***curly brackets*** instead of square brackets to both create and call cell array elements.

We can use function handles to create anonymous functions according to this syntax:

```
functionName = @(input_arguments) body
```

To illustrate:

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

In this example, **sqr** is the function name, **@** is the function handle which specifies that it accepts one input **x**, followed by the body of the anonymous function.

To try it out:

```
x = 1:10;
sqr(x)
```

```
ans = 1×10
     1     4     9    16    25    36    49    64    81   100
```

You can create anonymous functions with more than one variable:

```
myfunction = @(x,y) (x.^2 + y.^2 + x.*y);
x = 1:10;
y = 2:2:20;
myfunction(x,y)
```

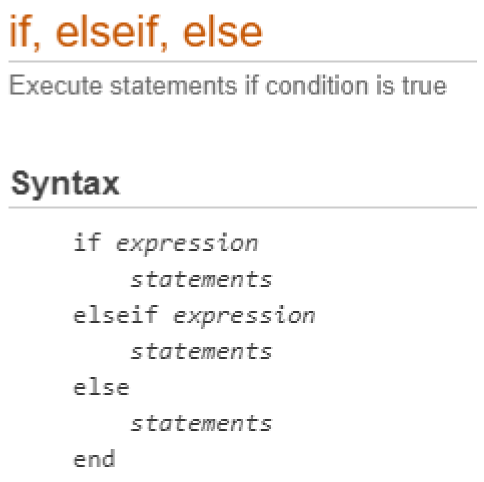
```
ans = 1×10
     7    28    63   112   175   252   343   448   567   700
```

Control Flow

In this section, we assume that you are familiar with all the control structures from previous programming courses. So, we will simply introduce their syntax and any worthy notes.

Conditionals

The syntax for the **`if`**, **`elseif`**, and **`else`** statements looks like this:



if, elseif, else

Execute statements if condition is true

Syntax

```
if expression
    statements
elseif expression
    statements
else
    statements
end
```

where the **`elseif`**, and **`else`** parts are *optional* and depend on your application. The expression of the **`if`** statement can be written *with* or *without* parenthesis. But we advise that once you start using compound logical expressions using the (**`&&, ||, !`**) that you enclose the expressions in parenthesis to improve readability and avoid logical errors. Review the following examples below. We advise you to refer to *Experiment 02 - MATLAB Fundamentals II - Vector and Matrix Logical Operations Section* to review the logical operators available for use in MATLAB.

```
% Example: One-Way If Statement
x = 10;
if x ~= 0
    disp('Nonzero value')
end
```

```
Nonzero value
```

```
% Example: Two-Way If Statement
x = -9;
if x > 0
    disp('Positive value')
else
    disp('Non-Positive Value')
end
```

```
Non-Positive Value
```

```
% Example: Nested If Statements
x = -10;
if x > 0
    disp('Positive value')
elseif x < 0
    disp('Negative Value')
elseif x ==0
    disp('Zero Value')
else
    disp('Fail')
end
```

```
Negative Value
```

```
% Example: Two-Way If Statement with Compound Expressions
x = -10;
if (x > 0) || (x < 0)
    disp('Non-Zero value')
else
    disp('Zero Value')
end
```

```
Non-Zero value
```

But, what if the input to the **`if`** statement was a vector or array instead of a scalar? Suppose we have this case:

```
x = [5, 0 , -5];
if x > 0
    disp('Positive value')
elseif x < 0
    disp('Negative Value')
elseif x == 0
    disp('Zero Value')
else
    disp('Fail')
end
```

```
Fail
```

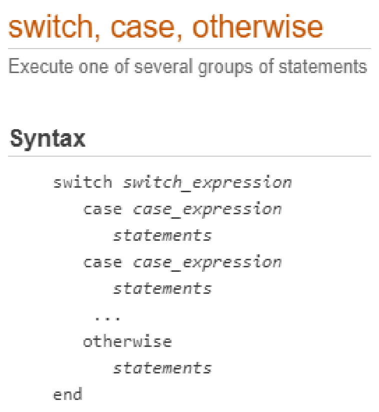
The above code will not go through the elements of **x** element-by-element. It will treat the vector or array as **one unit**. Either **all** of its elements satisfy one of the conditions or not. Here, each element in **x** satisfies a different case, so the vector as one unit won't match any case except the '**`Fail`**'. In contrast, in the code below, all the elements in **x** are positive, so the vector **x** as one unit is positive and thus it matches the first case.

```
x = [8, 24 , 55];
if x > 0
    disp('Positive value')
elseif x < 0
    disp('Negative Value')
elseif x == 0
    disp('Zero Value')
else
    disp('Fail')
end
```

```
Positive value
```

Switch Statement

The switch statement is similar to the one you have been introduced to before in C++ or Java. It has the following syntax:



switch, case, otherwise

Execute one of several groups of statements

Syntax

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

The major difference is that in MATLAB, you do **NOT** need a **`break`** statement between each **`case`** statement. Each sentence will end once the next **`case`** or **`otherwise`** statement begins. If no value matches any of the cases, then MATLAB will run the body of the **`otherwise`** statement. We will try the example below with the input **`-1`**

```
n = input('Enter one of the following numbers [-1, 0, 1]: ');
switch n
    case -1
        disp('negative one')
    case 0
        disp('zero')
    case 1
        disp('positive one')
    otherwise
        disp('other value')
end
```

```
negative one
```

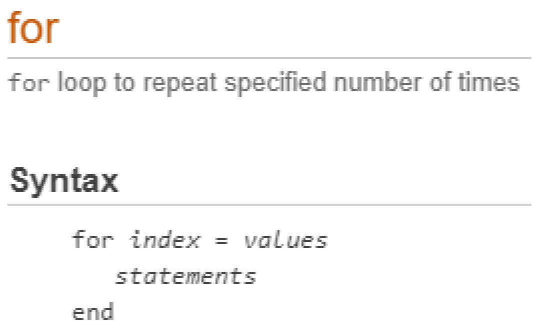
You can also compare characters or strings inside a MATLAB switch statement. We will try the example below with the input `g`:

```
Gender = input('Enter the gender of the newborn baby [b/g]: ', 's');
switch Gender
    case 'b'
        disp('Congratulations! It is a baby boy!')
    case 'g'
        disp('Congratulations! It is a baby girl!')
    otherwise
        disp('Alien baby?')
end
```

```
Congratulations! It is a baby girl!
```

Loops

The **`for`** loop is quite simple in MATLAB. It has the following syntax:



for

for loop to repeat specified number of times

Syntax

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

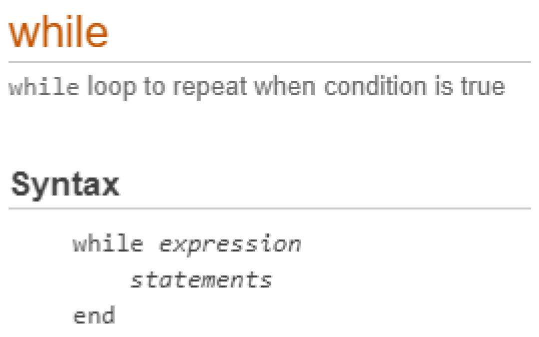
```
for v = 1.0:-0.2:0.0
   disp(v)
end
```

```
     1

    0.8000

    0.6000

    0.4000

    0.2000

     0
```

```
for v = [1 5 8 17]
   disp(v)
end
```

```
     1

     5

     8

    17
```

The **`while`** loop on the other hand keeps executing until the condition it is checking against becomes false. You must make sure the variable is updated inside the **`while`** loop so that you will not end with an infinite loop. The **`while`** loop has the following syntax:



while

while loop to repeat when condition is true

Syntax

```
while expression
   statements
end
```

For example, this is a **`while`** loop that computes the factorial of number n similar to the function factorial. We will try the example below with the input **`5.`**

```
n = input('Enter a value for n less than 10: ');
f = n;
while n > 1
    n = n-1;
    f = f*n;
end
disp(['n! = ' num2str(f)])
```

```
n! = 120
```

Remember that you can use the **`break`** keyword to exit the **`while`** or for **`loops`** at any time. Also, you can use the **`continue`** keyword, to only skip the current iteration and start the next one without fully exiting the loop.

Loop Vectorization

We often forget that MATLAB has been designed from the ground up to work easily with vectors or matrices. Anyone coming from C++ or Java programming background might use old techniques in problems MATLAB can handle quickly using its syntax and features.

Suppose you want to compute the sin of 1001 values ranging from 0 to 10 radian. You might naturally go with your first intuition and write the code as follows:

```
% Not Recommended
i = 0;
y = zeros (1,1001);
for t = 0:0.01:10
    i = i +1;
    y(i) = sin(t);
end
```

Whereas you forgot that MATLAB can readily do the same operation as:

```
% Recommended
t = 0:0.01:10;
y = sin(t);
```

The beauty of MATLAB is that it can work on entire vectors or arrays without having to loop over their elements one-by-one. The second code is much faster to run and is the only way we accept in this course. Vectorize whenever possible. Another example is if you have the following array and you want to find all the elements larger than or equal to zero.

```
A = [ 0, -1,  4; ...
    -14, 25,  9;...
    -34, 49, 64];
```

The wrong way to do it in MATLAB is:

```
% Not Recommended
[m, n] = size(A);
C = zeros([m,n]);
for K =  1:m*n
    if A(K) >=0
        C(K) = A(K);
    end
end
disp(C)
```

```
     0     0     4
     0    25     9
     0    49    64
```

Whereas you could have easily done the same thing by:

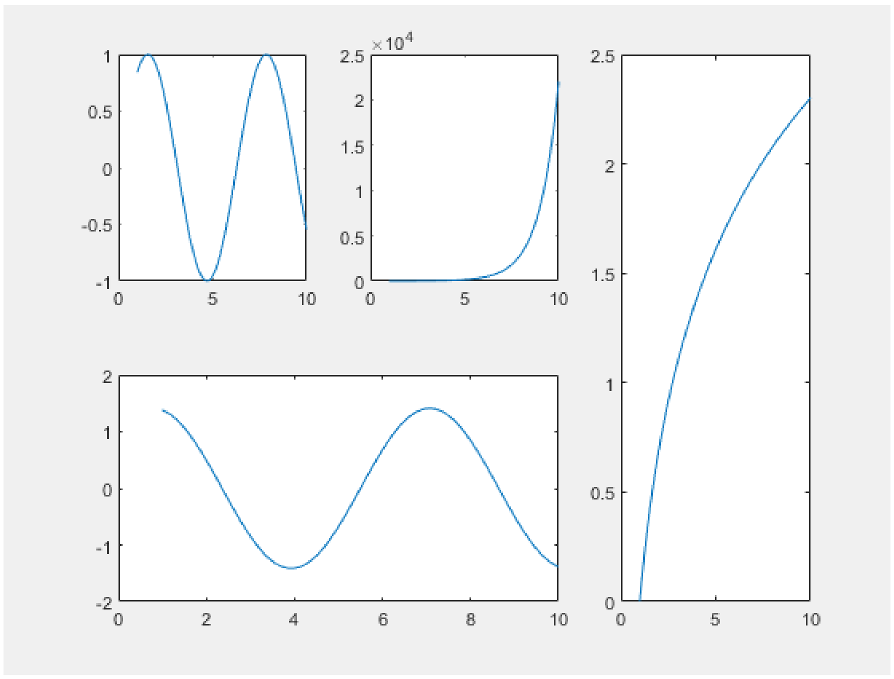
```
C = A;
C(C < 0) = 0
```

```
C = 3×3
     0     0     4
     0    25     9
     0    49    64
```

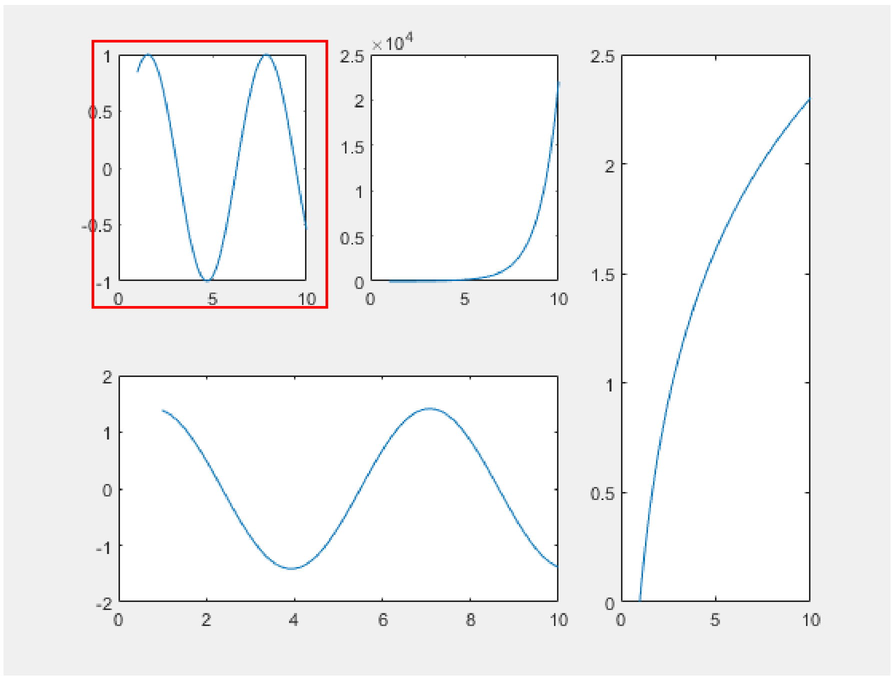
```
                                  Experiment version 1.00
                         Last Updated November 10th, 2020
                      Dr. Ashraf Suyyagh – All Rights Reserved
```

Quick Review on Plotting Tiled Layouts

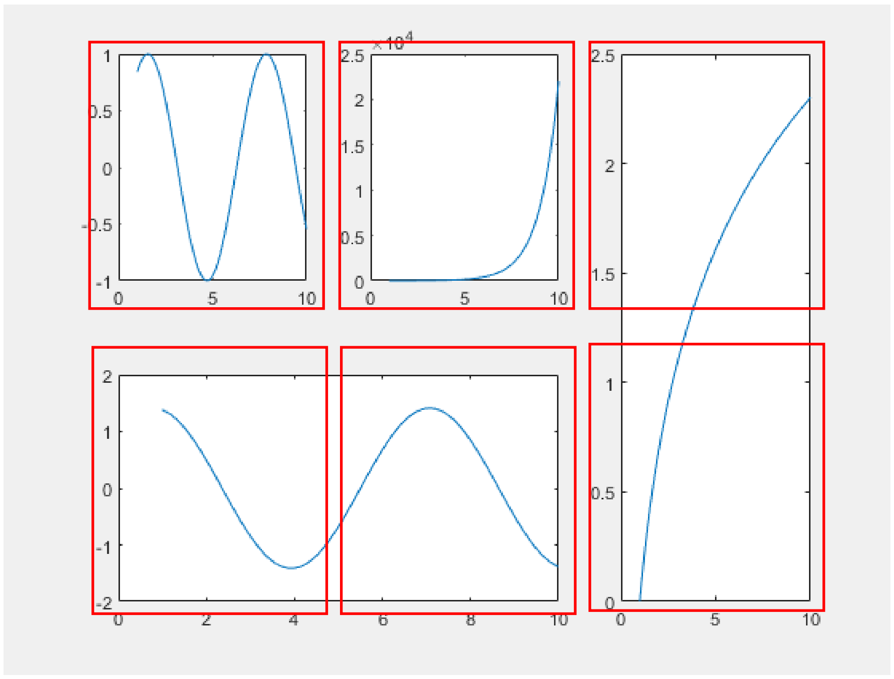
Suppose we want to plot the following signals in a tiled layout as shown in the figure below:



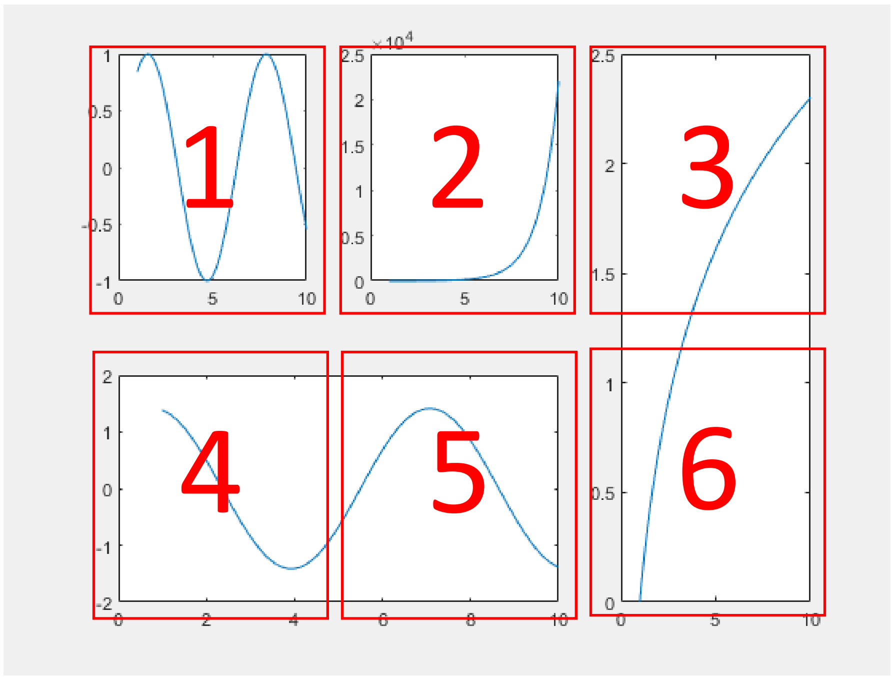
Step 1: Identify the smallest plot tile. In this case either one of the first two plots is the smallest.



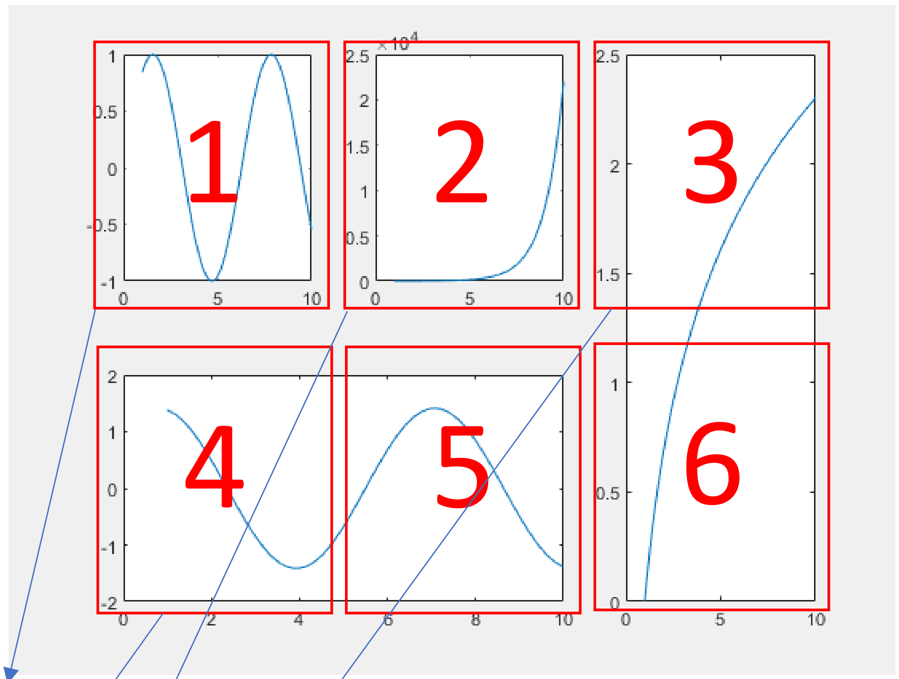
Step 2: Use the smallest tile as a scale to measure the other tiles:



Step 3: Number the tiles from left to right, then from up to down



Step 4: Use this numbering to identify the start of each plot, and the area it takes to fill this space in the tiled layout



nexttile (1)

nexttile (2)

nexttile(3, [2 1]) % Starts at 3, and takes a space of two rows, one column

nexttile (4, [1 2]) % Starts at 4, and takes a space of one row, two columns

Example Code:

```
x = 1:0.1:10;
y1 = sin(x);
y2 = exp(x);
y3 = log(x);
y4 = sin(x)+cos(x);

t = tiledlayout (2, 3);

nexttile (1)
plot(x, y1)

nexttile (2)
plot(x, y2)

nexttile(3, [2 1])
plot(x, y3)

nexttile(4, [1, 2])
plot(x, y4)
```

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 4 - MATLAB Plots

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

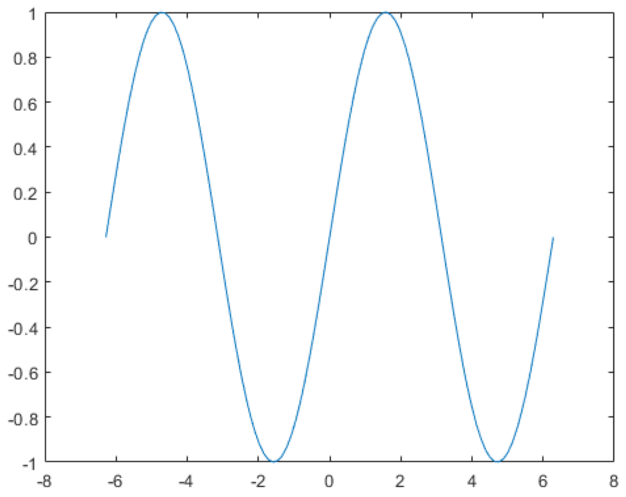
Introduction

Data visualization is important to analyse, understand, and make informed guesses or deductions about the underlying data. In the ribbon bar, and under the 'Plots' tab, MATLAB offers a large set of plot types. While it is easy to use these GUI plots readily, especially for one-time plots, they can be cumbersome to use for a large number of plots. If you want to use the GUI tool, you simply select the variables you want to plot from the Workspace window, then you click on the plot type applicable to this type of data. You can customize the graph afterwards. But if you want to repeat these steps for a larger number of files, the procedure will be repetitive and time-consuming. In this lab, we will learn how to write MATLAB commands that draw plots with different options and save them in any compatible output format. We can use the codes afterwards for any input data we want and provide customization options far more powerful than what you can do with the GUI plots (*e.g*. plot animation or stacked or tiled plots).

Simple Vector Plots on the Cartesian Axes

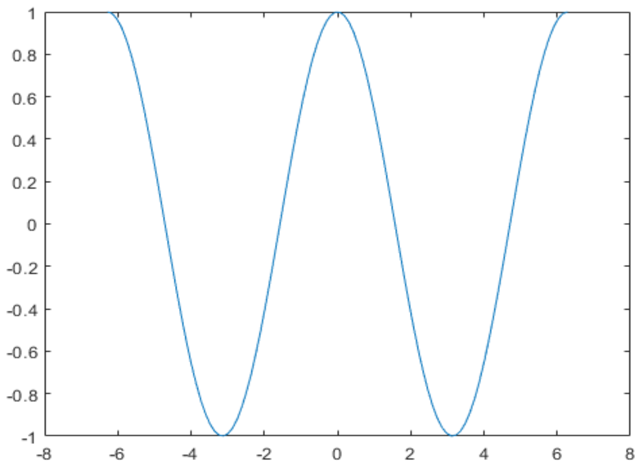
Often, we need to plot functions in relation to time $x(t)$, $y(t)$ or mathematical functions such as $y(x)$. In this type of graphs, it is only logical to have **equal-size** vectors associating the two variables together. To plot a simple sine wave from -2π to 2π, we can do the following:

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
```



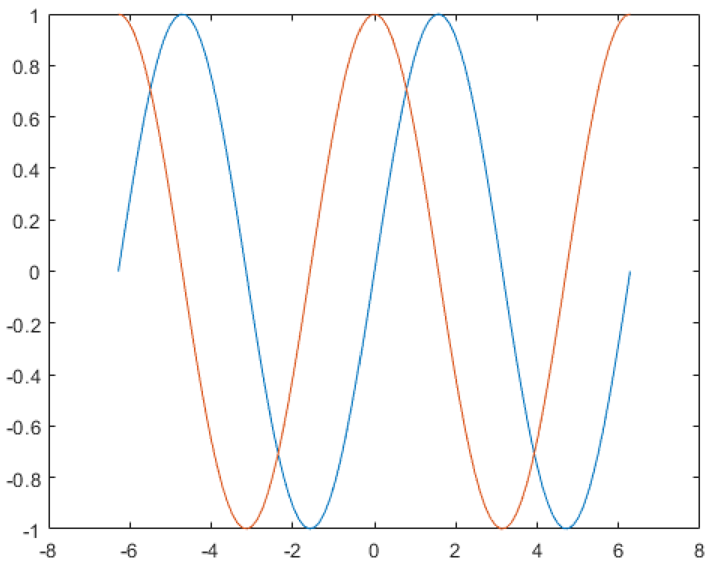
You will notice that a figure with the title **`Figure 1`** appears on the screen and the plot drawn on an axes inside. Now suppose you want to add another cosine wave to the same figure, if you try:

```
x2 = linspace(-2*pi,2*pi);
y2 = cos(x2);
plot(x2, y2);
```



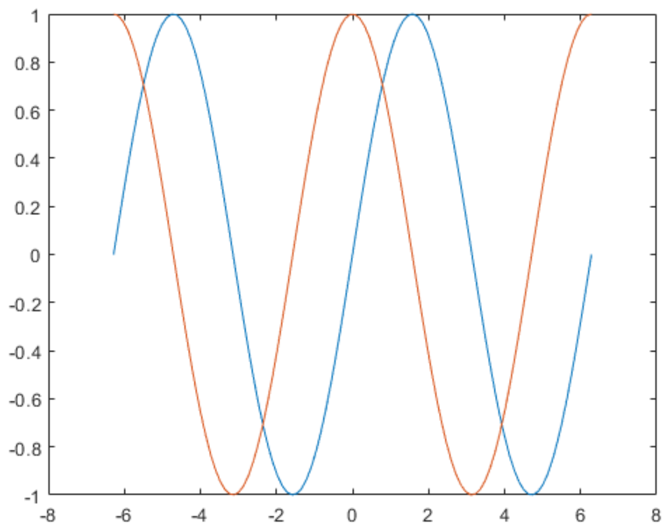
You will notice that the new figure will overwrite the previous figure! In order to keep the previous sine wave and add the cosine wave to it (superimpose the new plot over the old plot). You can expand the plot command such that it plots both the sine wave and the cosine wave in the same plot:

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
y2 = cos(x1);
plot(x1, y1, x2, y2);  % Notice we provided the two graphs together
```



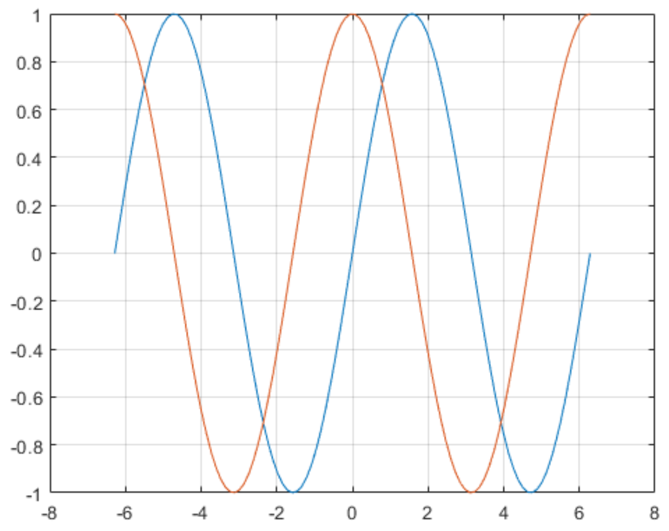
When you have lots of functions to plot on the same figure, the previous method can get a bit confusing (too long) or not visually appealing. Instead, we use the **`hold`** command which instructs MATLAB to hold (keep) the previous figure while we plot another on the same canvas. You can turn the hold **`on`** and **`off`** as suits your needs.

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
hold on
y2 = cos(x1);
plot(x1, y2);
hold off
```



Similarly, we can use the **`grid`** command to enable or disable drawing a grid in our figure. Grids can be useful to read the figure easily. However, you *enable* or *disable* the grid **after** you plot your functions.

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
hold on
y2 = cos(x1);
plot(x1, y2);
hold off
grid on
```

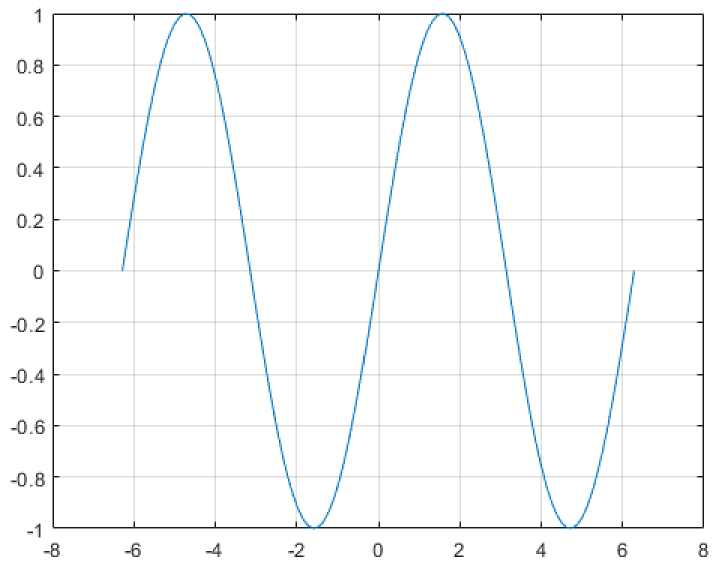


But what if you want to plot the two trigonometric functions on two separate figures instead of one overwriting one another, or having them on the same canvas? In this case, use the figure command to start a new canvas:

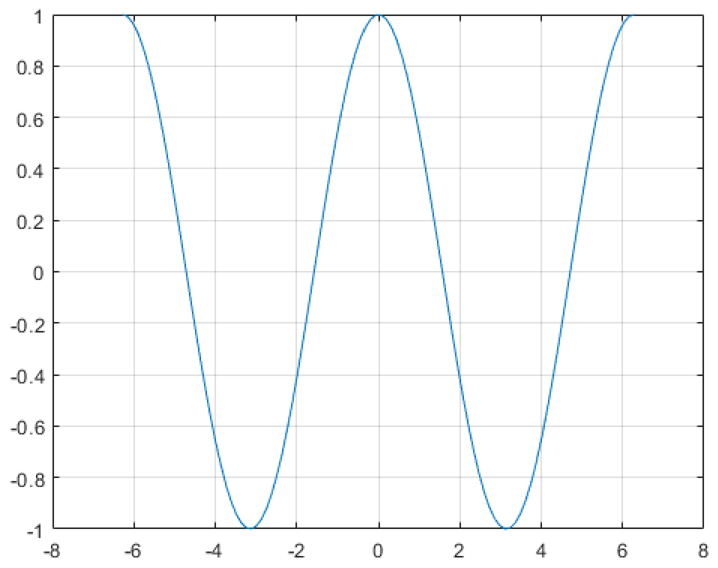
```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
figure
y2 = cos(x1);
plot(x1, y2);
grid on
```

In the above example, note that the grid was only shown for the second figure but not the first! It is important to note that all plot commands affect the most recent figure drawn, so to show the grid for each figure, you must do the following:

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
grid on
```



```
figure
y2 = cos(x1);
plot(x1, y2);
grid on
```



It will be extremely advantageous to be able to return to a certain figure and do some modifications. Since all edits by default are applicable to the most recent figure drawn, we can overcome this limitation by storing each plot canvas (figure) and its plot axes in two objects so that we can refer to them later and modify their properties by name pairs, if needed:

```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
h1 = figure;
p1 = plot(x1, y1);   % Save first plot in object h1
grid on
figure
y2 = cos(x1);        % Save second plot in object h2
h2 = figure;
p2 = plot(x1, y2);
grid on
```

Plotting a Matrix against a Vector using different Plot Commands

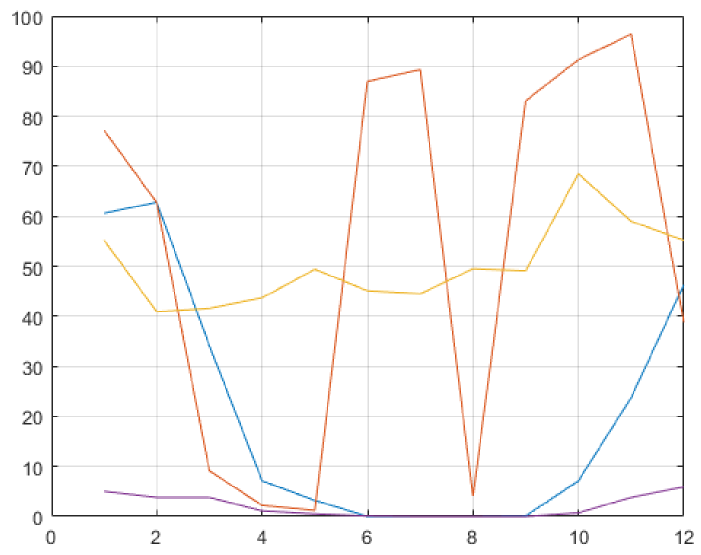
In the examples above, we have plotted a vector against another vector of equal size. MATLAB allows plotting a vector against a matrix, or a matrix against another matrix. Plotting a vector against a vector is simply a special case of plotting matrices against each other. As an example, suppose we have collected the average annual precipitation from four world cities in a matrix, and we want to plot the amount of rainfall (in mm) against each month. Matrix **`rainfall`** has the precipitation values for Amman, Montreal, London, and Cairo (Source: Wikipedia):

Table 1: Average Annual Precipitation (in mm)

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Amman	60.6	62.8	34.1	7.1	3.2	0	0	0	0.1	7.1	23.7	46.3
Montreal	77.2	62.7	9.1	2.2	1.2	87	89.3	4.1	83.1	91.3	96.4	38.8
London	55.2	40.9	41.6	43.7	49.4	45.1	44.5	49.5	49.1	68.5	59	55.2
Cairo	5	3.8	3.8	1.1	0.5	0.1	0	0	0	0.7	3.8	5.9

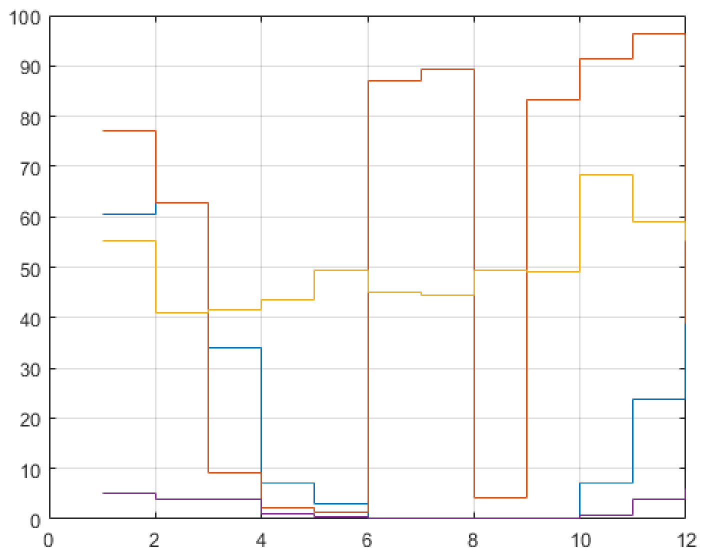
One condition when plotting matrices against vectors is that they must have one dimension of the same length. It is apparent that we need to plot against the months of the year, and we must have a value for each month. Our table satisfies this condition. MATLAB plots columns of a matrix, not its rows, so we must transpose the matrix before plotting it. Either enter it as in Table 1 above then transpose it or write it in transpose form.

```
months = 1:12;
rainfall = [60.6        77.2        55.2        5; ...
            62.8        62.7        40.9        3.8; ...
            34.1        9.1         41.6        3.8; ...
            7.1         2.2         43.7        1.1; ...
            3.2         1.2         49.4        0.5; ...
            0           87          45.1        0.1; ...
            0           89.3        44.5        0; ...
            0           4.1         49.5        0; ...
            0.1         83.1        49.1        0; ...
            7.1         91.3        68.5        0.7; ...
            23.7        96.4        59          3.8; ...
            46.3        38.8        55.2        5.9];
plot(months, rainfall)
grid on
```



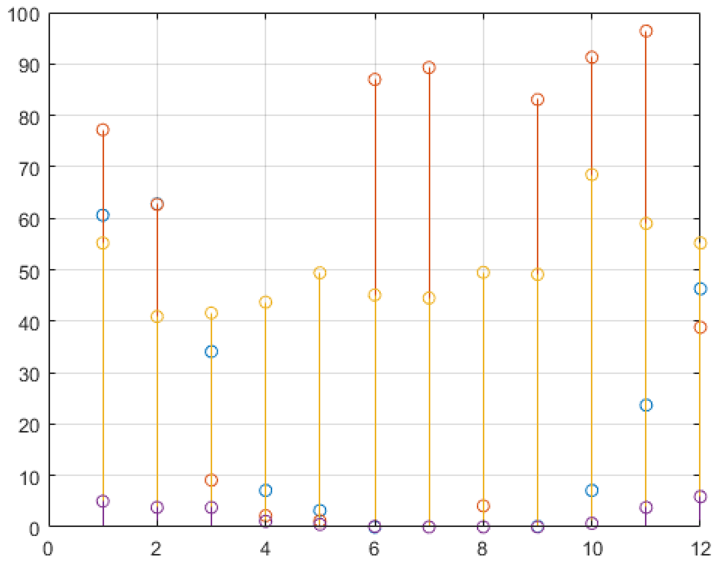
However; one can note that we are trying to plot discrete time data using a continuous plot. This might lead to false deductions; that the rainfall linearly increases or decreases between each month! Perhaps it is better to convey this data using a better graph, for example, lets try the **`stairs`** plot:

```
stairs(months, rainfall)
grid on
```



In our case, the **`stairs`** plot is definitely an improvement over the continuous **`plot`** function, but what if we try the **`stem`** plot?

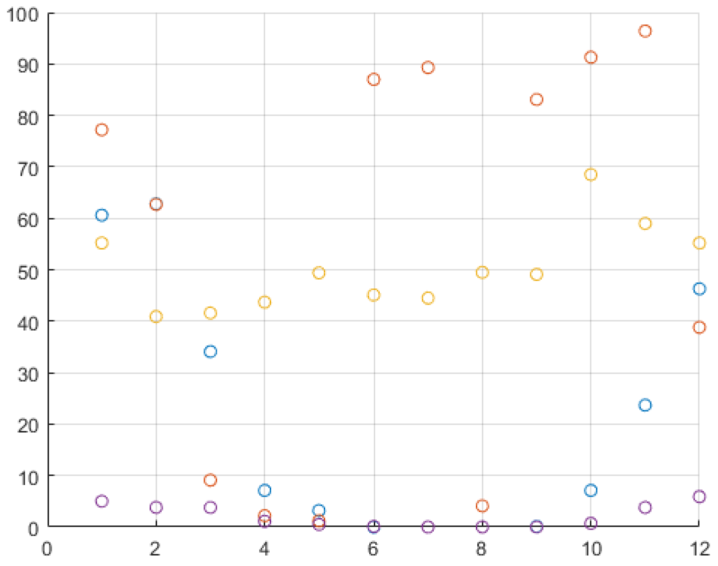
```
stem(months, rainfall)
grid on
```



It seems like the **stem** plot is indeed a better visual plot for our type of discrete data points.

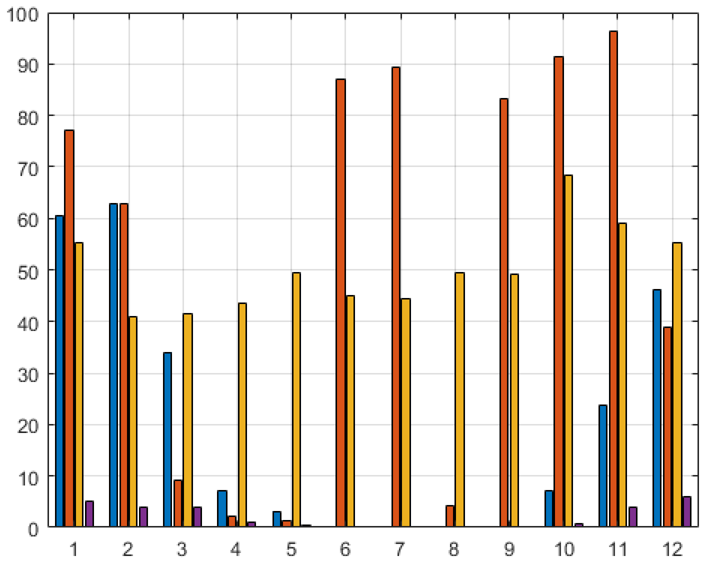
But, can we for example plot the datapoints only? without any lines connecting them? For this type of graph, we can use the **`scatter`** plot. Unfortunately, the **`scatter`** plot **cannot** handle matrices, so we need to divide the matrix into columns, and use the loop to go over and plot the matrix column-by-column while also having the **`hold`** command enabled to keep drawing on the same figure canvas:

```
[m, n] = size(rainfall);
for i = 1:n
    scatter(months', rainfall(:, i))
    hold on
end
grid on
hold off
```



We can also draw the data as bars using the **bar** command:

```
bar(months, rainfall)
grid on
```



Plotting a Matrix against another Matrix

Suppose we have the three functions:

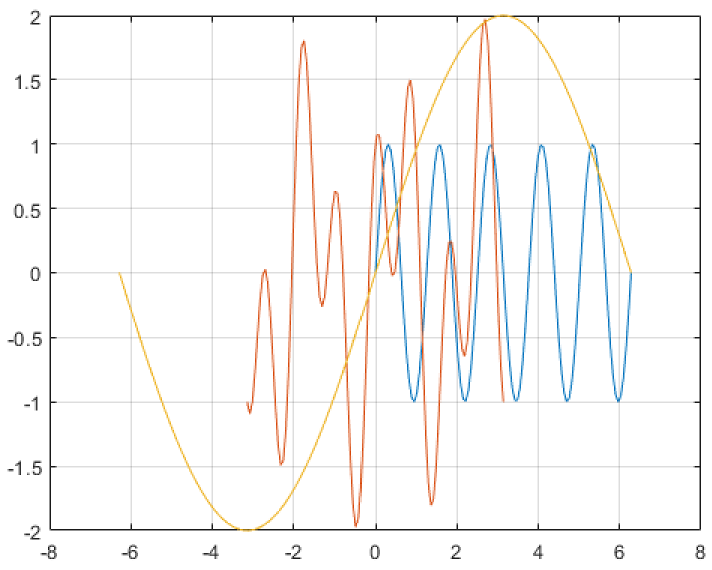
$y_1(t) = sin(5t)$,

$y_2(t) = sin(3t) + cos(7t)$, and

$y_3(t) = 2sin(0.5t)$

and you want to plot the three functions, with the same number of points of t (say 100) but you evaluate t at different points. We can group the t samples in a matrix of size `3x100,` and evaluate the functions in another matrix, then plot the matrices against each other. But remember when plotting matrices, MATLAB plots a column against a column, so we should make sure we prepare our data in this way to make sure we draw the desired plot:

```
t1 = linspace(0, 2*pi, 100)';
t2 = linspace(-pi, pi, 100)';
t3 = linspace(-2*pi, 2*pi, 100)';
t = [t1, t2, t3];
trigo = [sin(5*t1), ...
         sin(3*t2) + cos(7*t2), ...
         2*sin(0.5*t3)];
plot(t, trigo)
grid on
```



Plotting Different Data Against a Common Axis

Table 2 has more detailed meteorological data on the city of Amman. It has information about the average high and average low temperatures, as well as the average monthly sunshine hours, and the number of rainy days in each month. This table has one common base for all different data: the months. Yet, some data are temperatures in Celsius, others are in hours, while the last data is number of days. One way to plot such data is as follows:

Table 2: Amman Meteorological Data

Amman	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
Average High (C°)	2.7	13.9	17.6	23.3	27.9	30.9	32.5	32.7	30.8	26.8	20.1	14.6
Average Low (C°)	4.2	4.8	7.2	10.9	14.8	18.3	20.5	20.4	18.3	15.1	9.8	5.8
Precipitation Days	11	10.9	8	4	1.6	0.1	0	0	0.1	2.3	5.3	8.4
Average Monthly Sunshine Hours	79.8	182	226.3	266.6	328.6	369	387.5	365.8	312	275.9	225	179.8

Initially, we store this table inside the matrix **ammanMeteo:**

```
months = [1:12]';
ammanMeteo = [  2.7     4.2         11          79.8;  ...
                13.9    4.8         10.9        182;   ...
                17.6    7.2         8           226.3; ...
                23.3    10.9        4           266.6; ...
                27.9    14.8        1.6         328.6; ...
                30.9    18.3        0.1         369;   ...
                32.5    20.5        0           387.5; ...
                32.7    20.4        0           365.8; ...
                30.8    18.3        0.1         312;   ...
                26.8    15.1        2.3         275.9; ...
                20.1    9.8         5.3         225;   ...
                14.6    5.8         8.4         179.8];
```

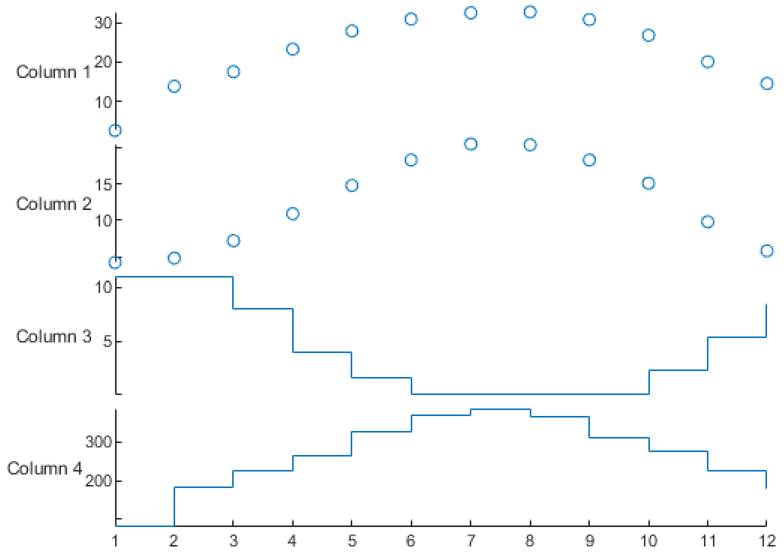
Then, we use a special type of plot command called **`stackedplot:`**

```
  h3 = figure;
  p3 = stackedplot(months, ammanMeteo)
```

As you can see, all four variables have been stacked on top of each other. Each has its own y-axis, and they share a common **months** axis. But the issue now, is that all our plots so far need annotation. That is to specify plot titles, axis names, legends and so on. We will see how to do this later.

By default, the stacked plots are continuous, so what if we want to change some or all of the plot types to scatter or stairs (Only these two types are supported in stackedplots)? In this case, after you create the stackedplot. you can edit the type as needed:

```
    p3.LineProperties(1).PlotType = 'scatter';
    p3.LineProperties(2).PlotType = 'scatter';
    p3.LineProperties(3).PlotType = 'stairs';
    p3.LineProperties(4).PlotType = 'stairs';
```

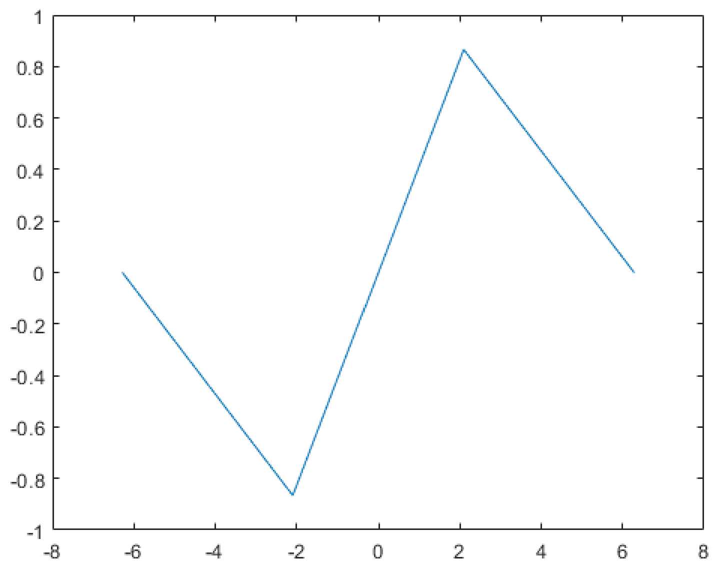


Tricky Function Plots

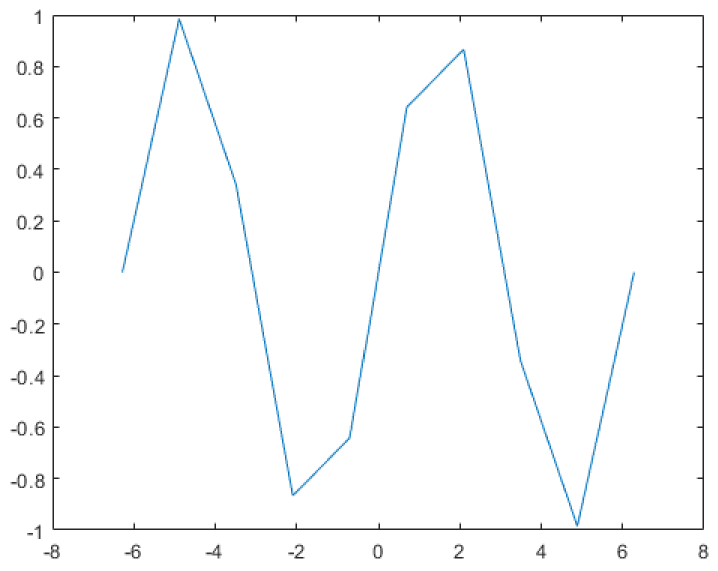
When we plot functions such as $y(x)$ or $x(t)$, our approach is to create a vector of inputs x or t then substitute them into the associated function. We are creating discrete points that MATLAB plots. Do not be misled to believe that what you see is an actual continuous function. MATLAB has connected the discrete data points (samples) and gave you the illusion that the function is continuous .

In fact, MATLAB is good at approximating the function shape depending on how many sample data points you provide it with. Few samples might not capture the actual shape and characteristics of the underlying function, while too much data points are unnecessary, time-consuming and slow your code. In the code below, we will attempt to draw the sine function in four separate figures over the entire range of -2π to 2π using different vector sizes:

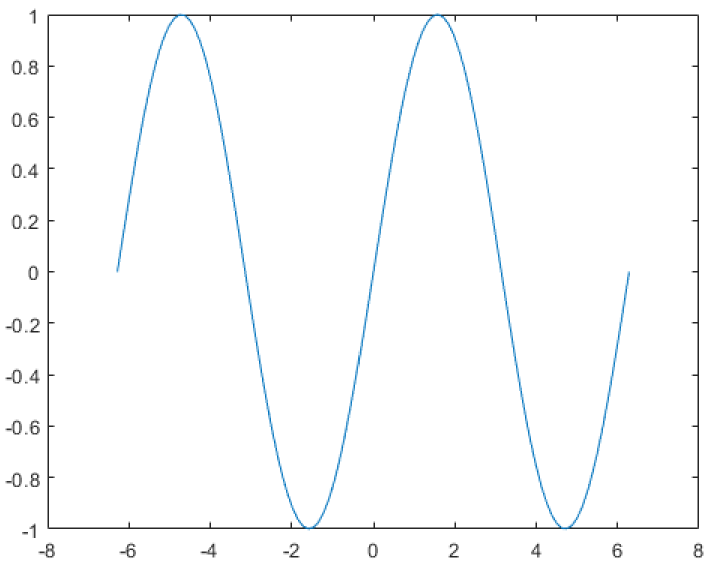
```
  t = linspace(-2*pi, 2*pi, 4);
  y = sin(t);
  plot(t,y)
```



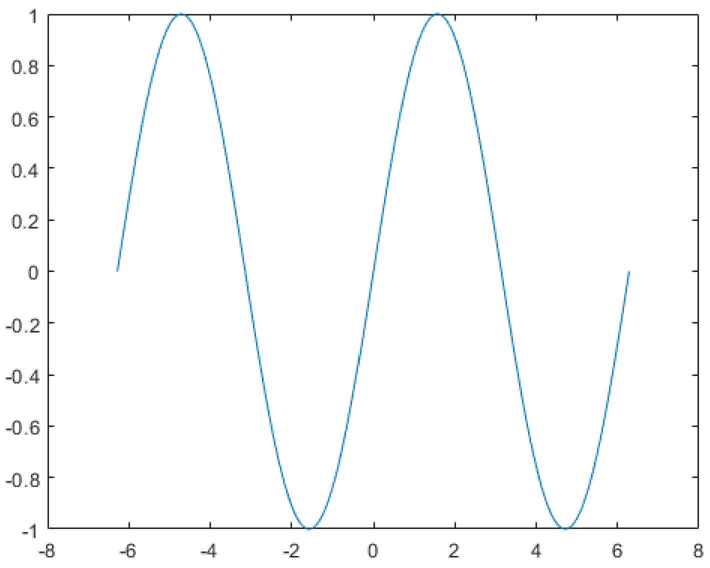
```
figure
t = linspace(-2*pi, 2*pi, 10);
y = sin(t);
plot(t,y)
```



```
figure
t = linspace(-2*pi, 2*pi, 100);
y = sin(t);
plot(t,y)
```



```
figure
t = linspace(-2*pi, 2*pi, 1000);
y = sin(t);
plot(t,y)
```

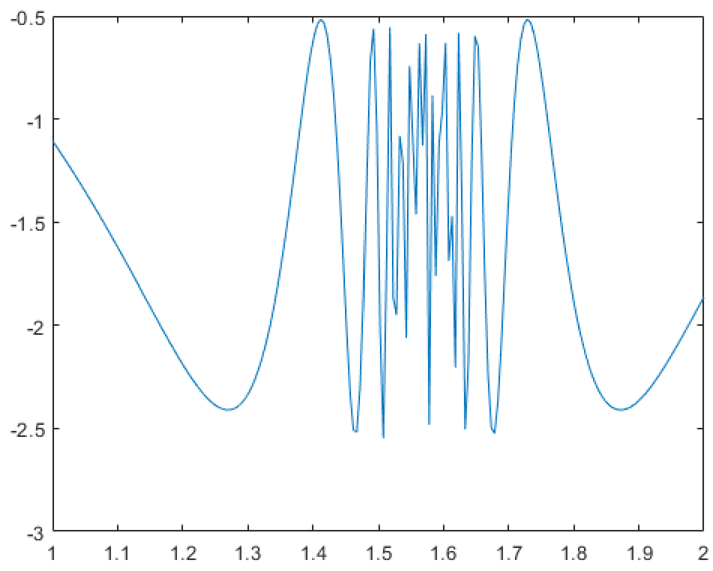


Notice how smaller vectors yielded different shapes that the expected sine function. As we increased the number of sample points, the shape did indeed turn into a sine. There is no observable difference between using 100 or 1000 samples.

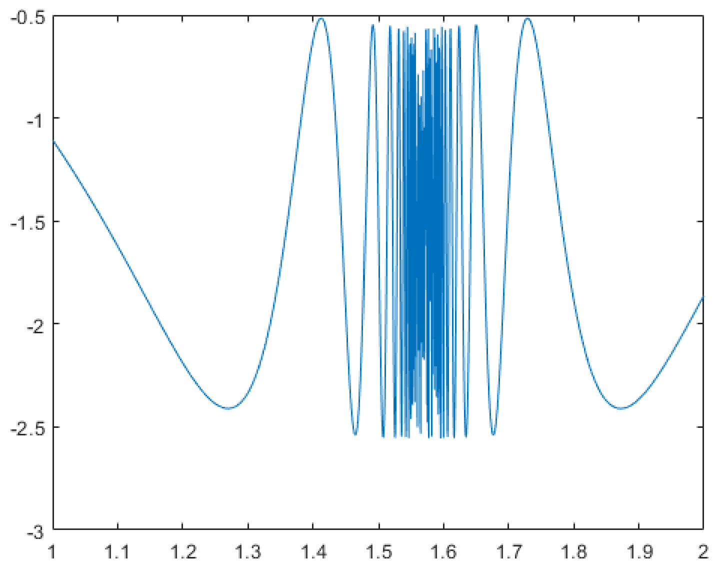
In the previous case, we could easily determine that 100 was enough because we know what to expect when drawing a sine wave. But what if we are working with a new function whose shape we do not know, how can we guess that the number of samples we choose is indeed enough?

Suppose we want to plot the function $y(x) = cos(tan(x)) - tan(sin(x))$ over the range of 1 to 2. Normally, we would create a large vector of x then substitute it in the function equation:

```
x = linspace(1, 2, 200);
y = cos(tan(x)) - tan(sin(x));
plot(x, y)
```



```
figure
x = linspace(1, 2, 2000);
y = cos(tan(x)) - tan(sin(x));
plot(x, y)
```



As you can see, plotting over large samples 200 and 2000 gave different plot shapes? Do we even know if 2000 are enough now?

MATLAB has a smart function plot command called **`fplot`** which takes as input the function as an anonymous function, and the range we want to plot the function over. The **`fplot`** command internally determines the correct number of samples it needs and then plots the function accordingly. The default interval for the fplot command is from -5 to 5. To draw the previous function using **`flpot`** and over the range from 1 to 2, one can write:

```
fplot(@(x) cos(tan(x)) - tan(sin(x)), [1 2]  )
```

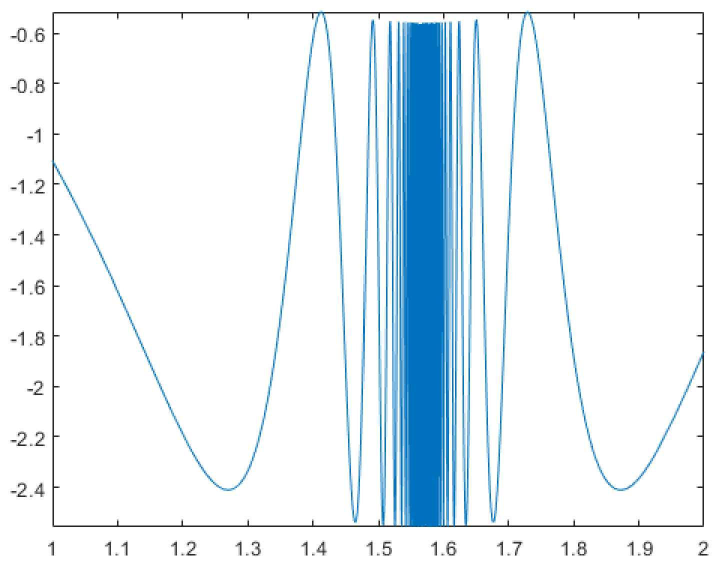
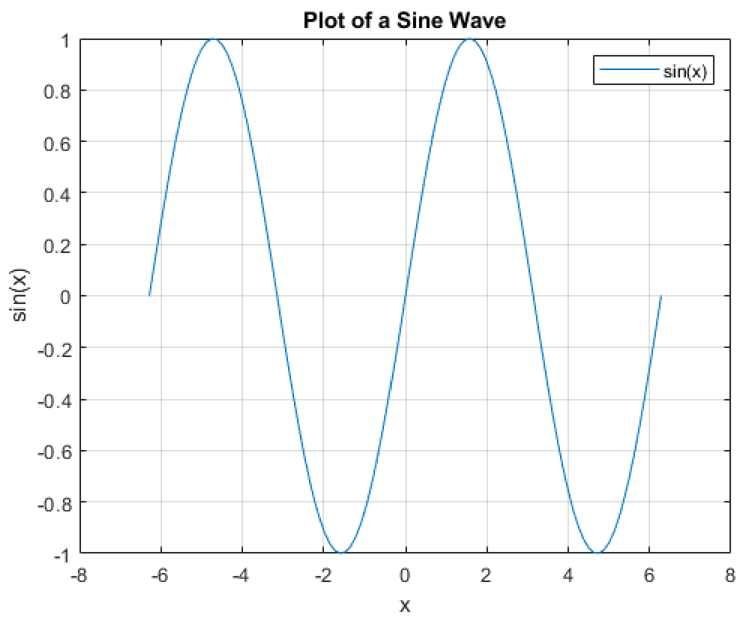


Figure Annotation and Options

Professional plots need not only display accurate representation of the underlying data, but also make the reader understand what they represent. Plots must have titles, proper axes titles, and legends. Let's start over with our first plot, and edit the code with proper annotation. In the first instance, we will assume that we did not save the plot in an object, so by default, all subsequent plot commands will default to the most recent plotting canvas. To add titles, names for any axis, or a legend, we use the **title**, **xlabel**, **ylabel**, and **legend** commands. We pass a string for each that would appear into its respective location.

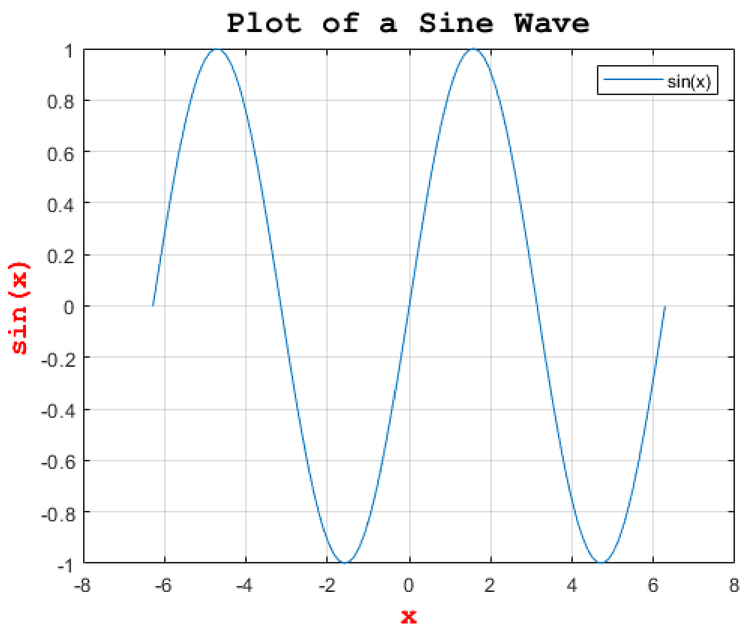
```
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
title('Plot of a Sine Wave')
xlabel('x')
ylabel('sin(x)')
legend('sin(x)')
grid on
```



You can also change the font size, type, and colour of the figure title or labels. You can do so by using name-pair arguments. That is, you specify what you want to change followed by its value. The available options are:

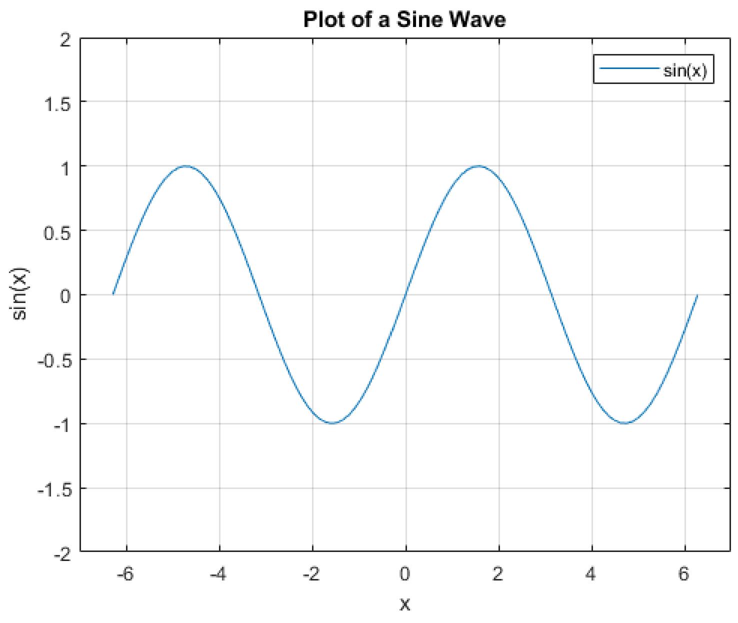
- `'FontSize':` Default (11), you can set the size to 12, 14, *etc.*
- `'FontWeight'` Default ('normal') can be either `'normal'` or `'bold'`
- `'FontName'` Default ('FixedWidth') or the name of a font installed and supported by your OS
- `'Color'` MATLAB assigns default colours, you can change them to any colour as specified in the customization section (later).

```
 x1 = linspace(-2*pi,2*pi);
 y1 = sin(x1);
 plot(x1, y1);
 title('Plot of a Sine Wave', 'FontSize', 16, 'FontWeight', 'bold', "FontName",
'FixedWidth' )
 xlabel('x', 'FontSize', 14, 'FontWeight', 'bold', "FontName", 'FixedWidth',
'Color', 'red')
 ylabel('sin(x)', 'FontSize', 14, 'FontWeight', 'bold', "FontName",
'FixedWidth','Color', 'red' )
 legend('sin(x)')
 grid on
```



You can also change the span of the x-axis and y-axis, this does not affect the range of the function, only the axis:

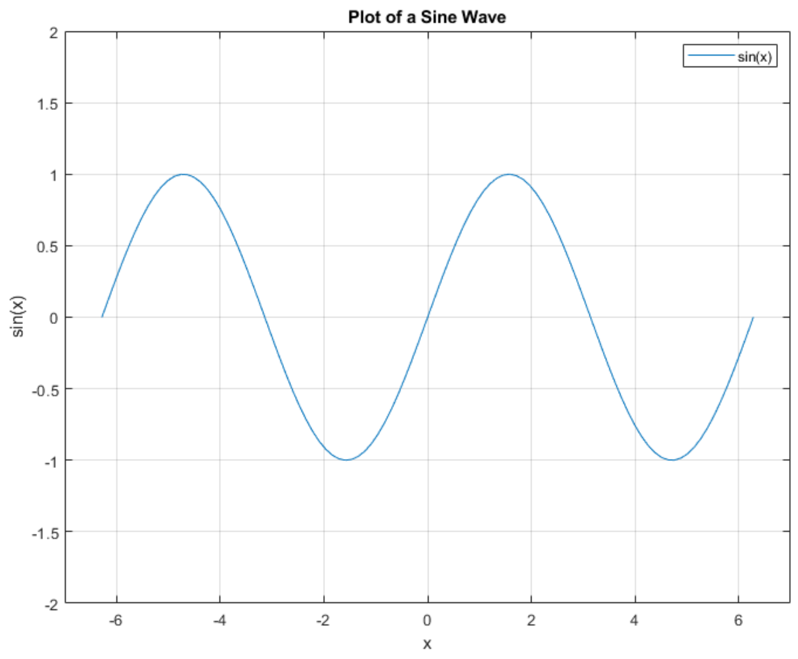
```
 x1 = linspace(-2*pi,2*pi);
 y1 = sin(x1);
 plot(x1, y1);
 title('Plot of a Sine Wave')
 xlabel('x')
 ylabel('sin(x)')
 legend('sin(x)')
 axis ([-7 7 -2 2])  % Insert the range of the x-axis and y-axis in order
 grid on
```



If you want to specify a size and location for your figure on the screen, you need first to set the units ('`pixels`' (default), '', '`centimeters`', '`inches`'), then you specify the following vector [left bottom width height] in the unit you have chosen.

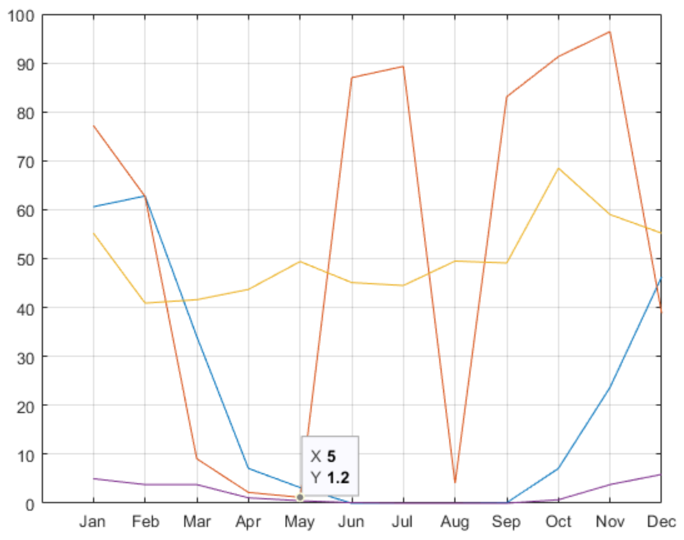
So, in the next example, we are specifying that the figure is to be 300 pixels away from the bottom left corner of the screen, and that it has a size of 640×480 pixels.

```
figure("Units","pixels", "Position", [300 300 640 480])
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
title('Plot of a Sine Wave')
xlabel('x')
ylabel('sin(x)')
legend('sin(x)')
axis ([-7 7 -2 2])  % Insert the range of the x-axis and y-axis in order
grid on
```



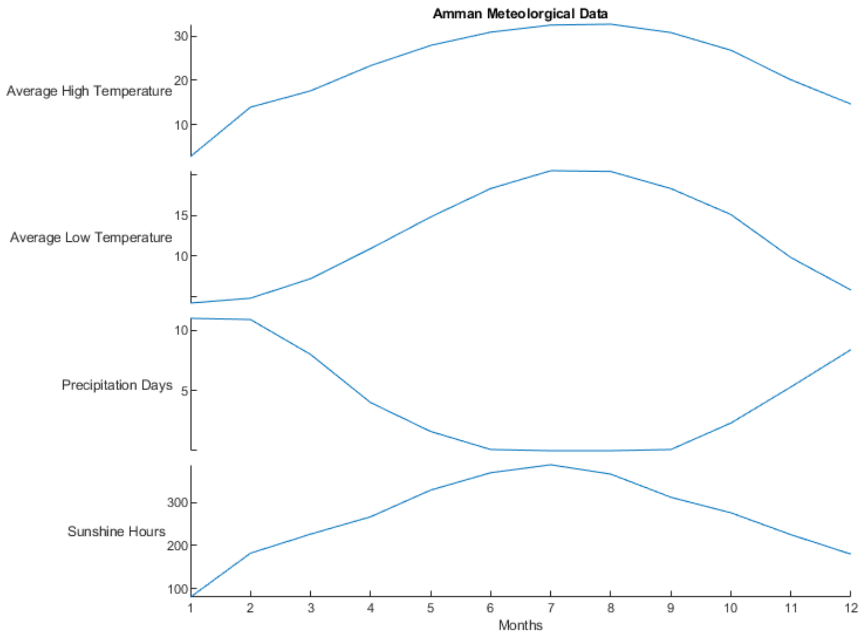
If you need to overwrite the labels used on the x-axis or y-axis, you need to pass the new labels as a string and specify which axis you are changing (`xticklabels`). For example, we can replace the months name in the previous plot with their name:

```
 plot(months, rainfall)
 xticks([1:12])
 xticklabels({'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
'Oct', 'Nov', 'Dec'})
 grid on
```



To annotate stacked plots like the one we created for Amman's metrological data, we need to specify multiple y-axis labels for each stacked plot. Create a string cell for each stacked plot in the order they appear inside the matrix, and pass it to the **`stackedplot`** command:

```
 figure("Units","pixels", "Position", [300 300 800 600])
 newYlabels = {'Average High Temperature', 'Average Low Temperature',
'Precipitation Days', 'Sunshine Hours'};    % Note we create a cell array by
using curly brackets {}
 stackedplot(months, ammanMeteo, 'DisplayLabels',newYlabels)
 title('Amman Meteolorgical Data')
 xlabel('Months')
```



Tiled Plots

Sometimes we need to display four, six, or nine plots on the same figure. We can do this using tiled layout. Let us define and draw four sine waves at different frequencies and draw each in a different tile in a **`4x4`** grid:

```
 x = linspace(0,30);
 y1 = sin(x);
 y2 = sin(x/2);
 y3 = sin(x/3);
 y4 = sin(x/4);
 t = tiledlayout(2,2);
```

So far, the figure is empty, to start adding plots, we need to use the **`nexttile`** command and any plot command of our choice. You can give titles to your tiles as you go:

```
% Tile 1
nexttile
plot(x,y1)
title('F = 1')

% Tile 2
nexttile
plot(x,y2)
title('F =  1/2')

% Tile 3
nexttile
plot(x,y3)
title('F = 1/3')

% Tile 4
nexttile
plot(x,y4)
title('F =  1/4')
```

You can change the spacing between the tiles by using one of these three options ('`normal`', '`compact`', 'none'):

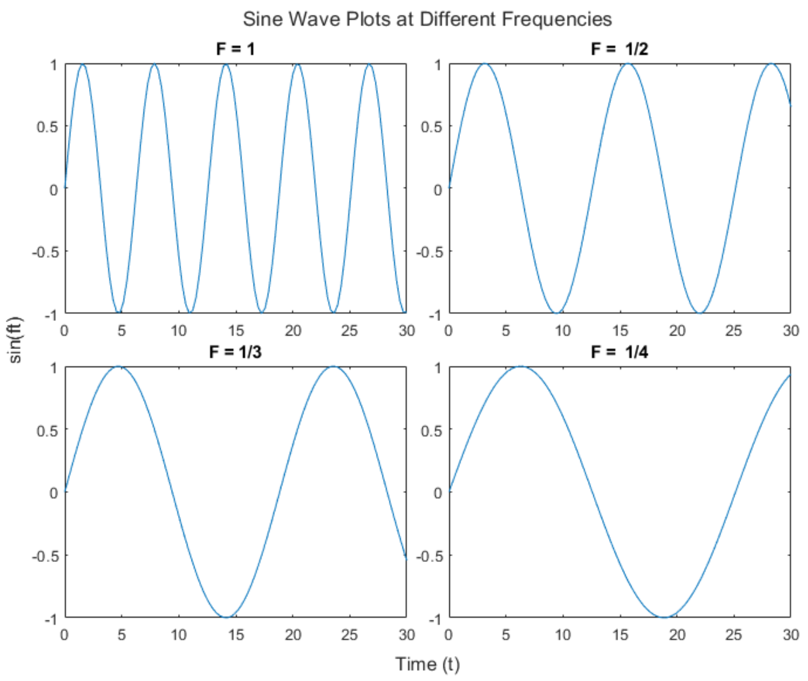
```
t.TileSpacing = 'compact';
```

or

```
t.TileSpacing = 'none';
```

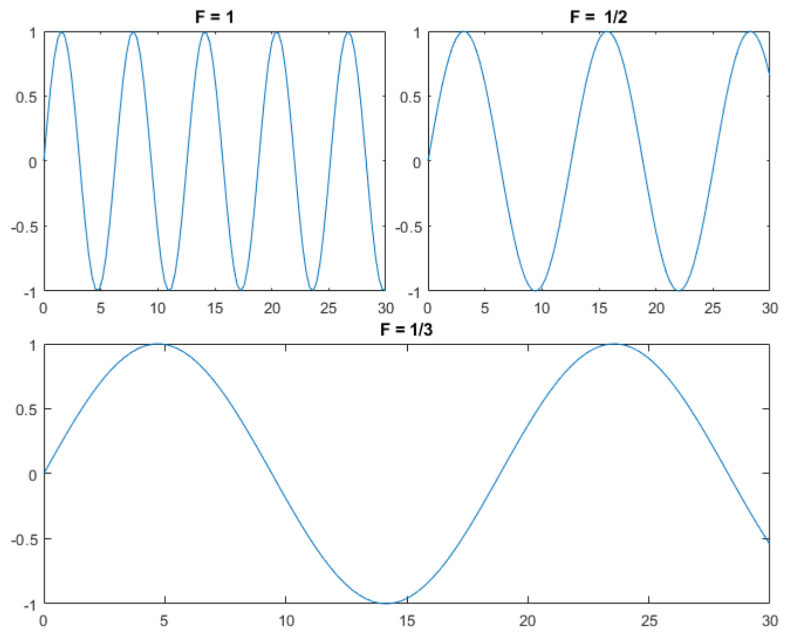
At the end, you can create a shared title and common axes using what we have already learnt:

```
title(t,'Sine Wave Plots at Different Frequencies')
xlabel(t,'Time (t)')
ylabel(t,'sin(ft)')
```



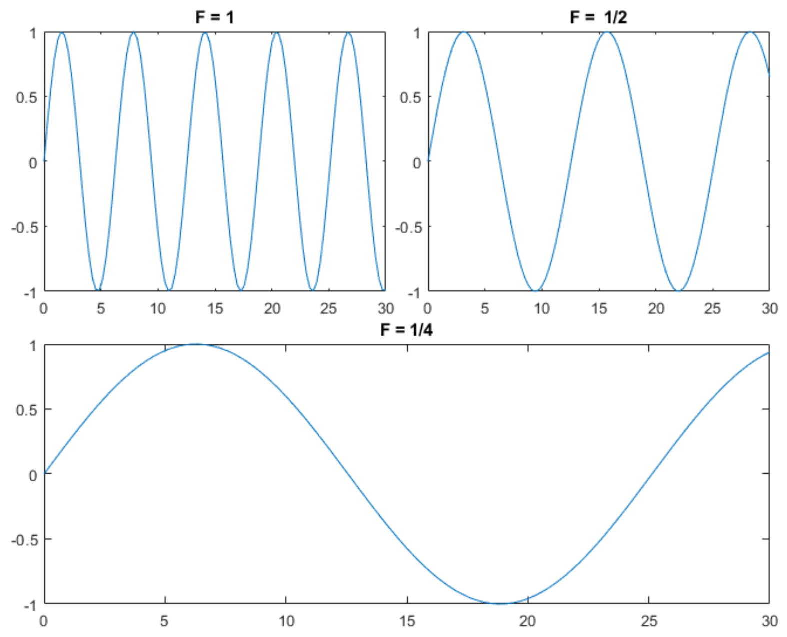
We can make one tile span multiple tiles as follows:

```
t2 = tiledlayout(2,2);

% Tile 1
nexttile
plot(x,y1)
title('F = 1')

% Tile 2
nexttile
plot(x,y2)
title('F =  1/2')

% Tile 3
nexttile([1 2])   % Here we are specifying that this tile will take the space
of 1 row, 2 columns
plot(x,y3)
title('F = 1/3')

t2.TileSpacing = 'none';
```



Suppose you want to modify a certain tile after plotting it. For example, in the previous example, to plot $sin(\frac{1}{4}t)$ instead of $sin(\frac{1}{3}t)$, all you need to do is specify the tile number:

```
nexttile(3)
plot(x,y4)
title('F = 1/4')
```



Plot Customization

You can change the colour of your plot, the line shape, width, and even add markers at measurement (sample) points by using name-value pair arguments. Each of these pairs is **optional**. So you can use any combination of them, or none. You start by writing the option name you would like to change, then assign to it any of the applicable options. The main properties which you can change are:

- **`'Color':`** MATLAB can accept colors either as an RGB values triplet, or their equivalent color Hexadecimal code.

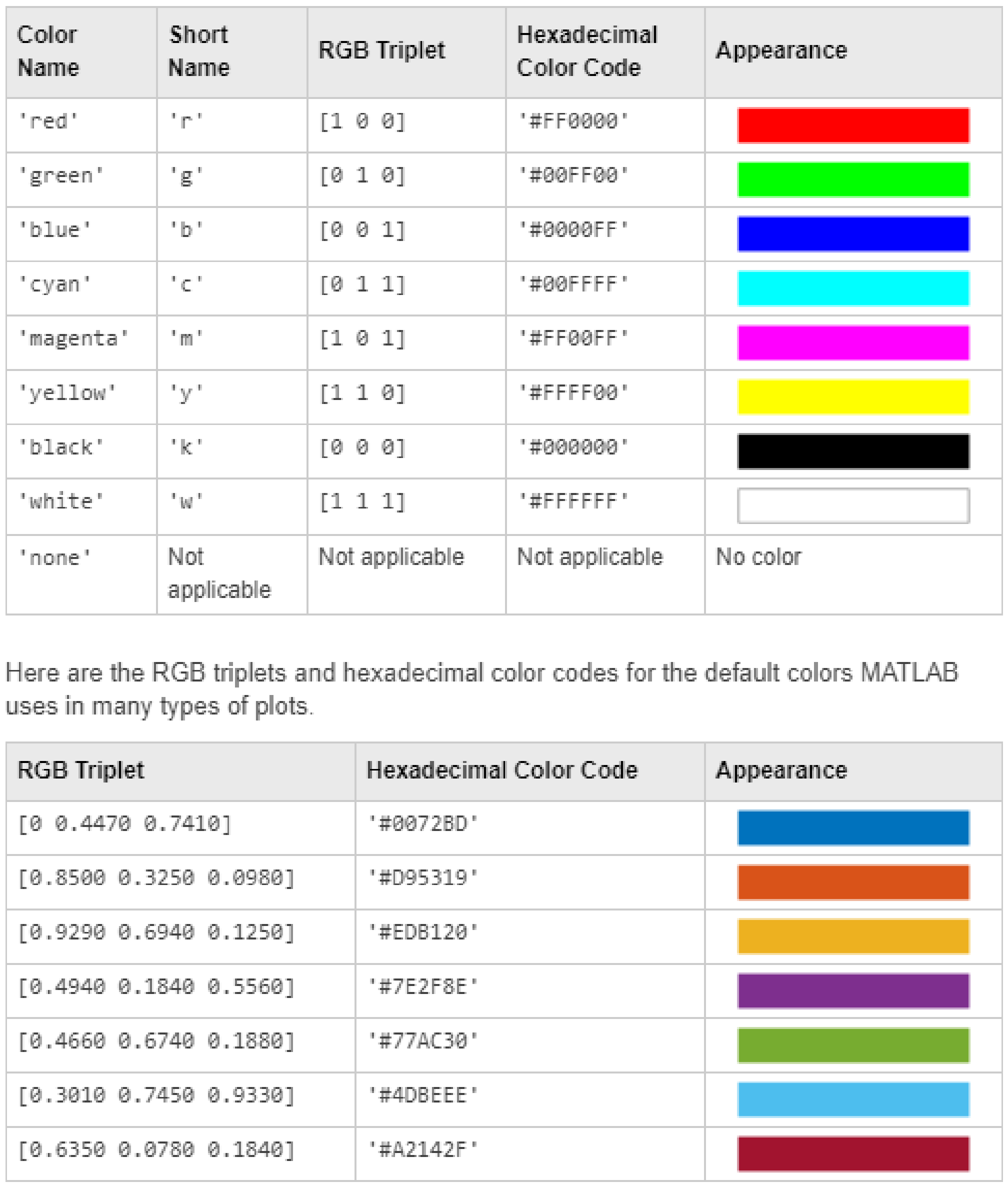
An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range [0,1]; for example, [0.4 0.6 0.7]. On the Internet, you will find RGB values from the range of 0 to 255. In order to use them in MATLAB, you need to divide them by 255.

A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from `0` to `FF each (e.g. '#FF8800'.` It is basically writing the RGB codes in hexadecimal next to each other).

You can use any of the many available websites to find beautiful colour palettes for your plots. These websites will provide you with the RGB and Hex codes:

- https://htmlcolorcodes.com/
- https://www.color-hex.com/
- https://en.wikipedia.org/wiki/Web_colors

For few select colours, MATLAB has assigned names that you can use instead of their RGB or hexadecimal notation.



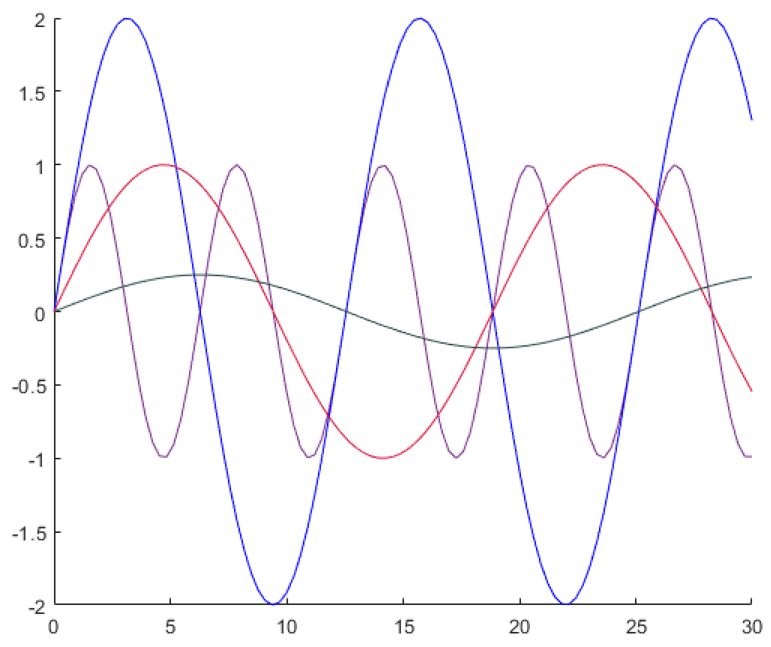
Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	🟥
'green'	'g'	[0 1 0]	'#00FF00'	🟩
'blue'	'b'	[0 0 1]	'#0000FF'	🟦
'cyan'	'c'	[0 1 1]	'#00FFFF'	🟦
'magenta'	'm'	[1 0 1]	'#FF00FF'	🟪
'yellow'	'y'	[1 1 0]	'#FFFF00'	🟨
'black'	'k'	[0 0 0]	'#000000'	⬛
'white'	'w'	[1 1 1]	'#FFFFFF'	⬜
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	🟦
[0.8500 0.3250 0.0980]	'#D95319'	🟧
[0.9290 0.6940 0.1250]	'#EDB120'	🟨
[0.4940 0.1840 0.5560]	'#7E2F8E'	🟪
[0.4660 0.6740 0.1880]	'#77AC30'	🟩
[0.3010 0.7450 0.9330]	'#4DBEEE'	🟦
[0.6350 0.0780 0.1840]	'#A2142F'	🟥

Let's try few of these colours:

```
x = linspace(0,30);
y1 = sin(x);
y2 = 2*sin(x/2);
y3 = sin(x/3);
y4 = 0.25*sin(x/4);
figure("Units","pixels", "Position", [300 300 600 480])
hold on
plot(x,y1, "color", '#7E2F8E')
plot(x,y2, "color", 'blue')
plot(x,y3, "color", '#DC143C')  % Crimson
plot(x,y4, "color", [0.1843, 0.3098, 0.3098])  % DarkSlateGray RGB = 47  79  79
```



- **`'LineStyle'`** You can use it to change the shape of your line from solid lines to dashed or dotted. MATLAB has these options available:

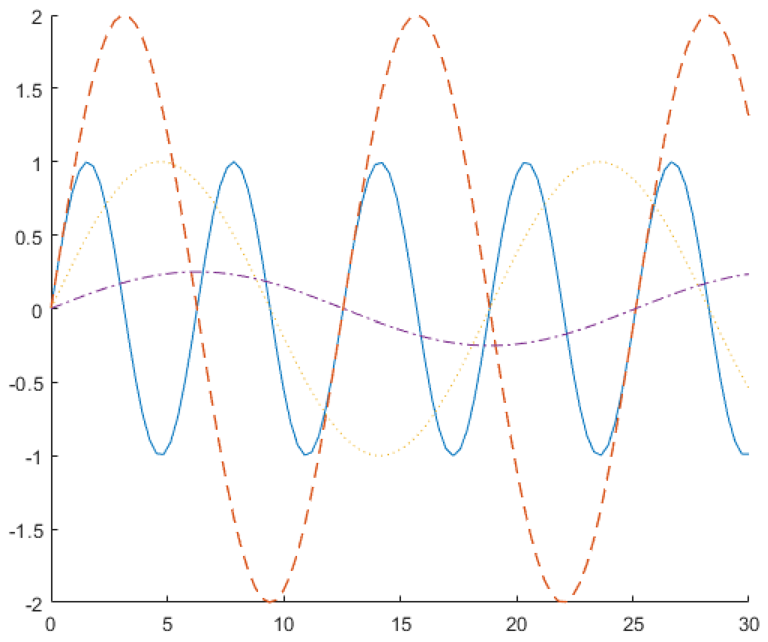
Line Style	Description	Resulting Line
`'-'`	Solid line	————
`'--'`	Dashed line	- — — — —
`':'`	Dotted line	....
`'-.'`	Dash-dotted line	—·—·—·—·-
`'none'`	No line	No line

- **`'LineWidth'`** Default (0.5) You can change the thickness of your line using this option:

Let us modify our code above to change the line type and widths:

```
x = linspace(0,30);
y1 = sin(x);
y2 = 2*sin(x/2);
y3 = sin(x/3);
y4 = 0.25*sin(x/4);
figure("Units","pixels", "Position", [300 300 600 480])
hold on
```

```
plot(x,y1,  'LineStyle', '-' , 'LineWidth', 0.75)
plot(x,y2,  'LineStyle', '--', 'LineWidth', 1)
plot(x,y3,  'LineStyle', ':' , 'LineWidth', 0.25)
plot(x,y4,  'LineStyle', '-.')
```

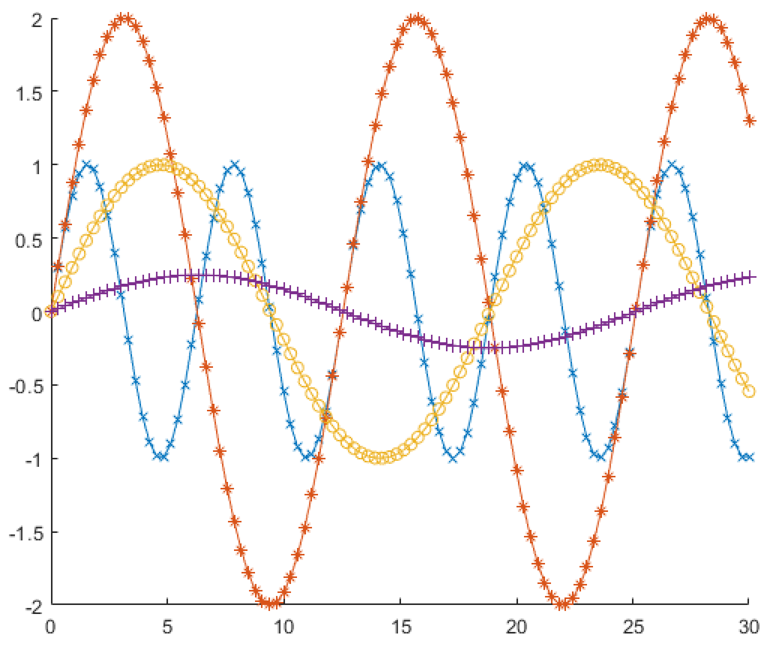


- `'Marker'` By default, MATLAB shows no markers on the line. However, if you need to, you can display markers on all or some of the samples in the plot. Markers come in different shapes. The options available in MATLAB are:

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle
'pentagram' or 'p'	Five-pointed star (pentagram)
'hexagram' or 'h'	Six-pointed star (hexagram)
'none'	No markers

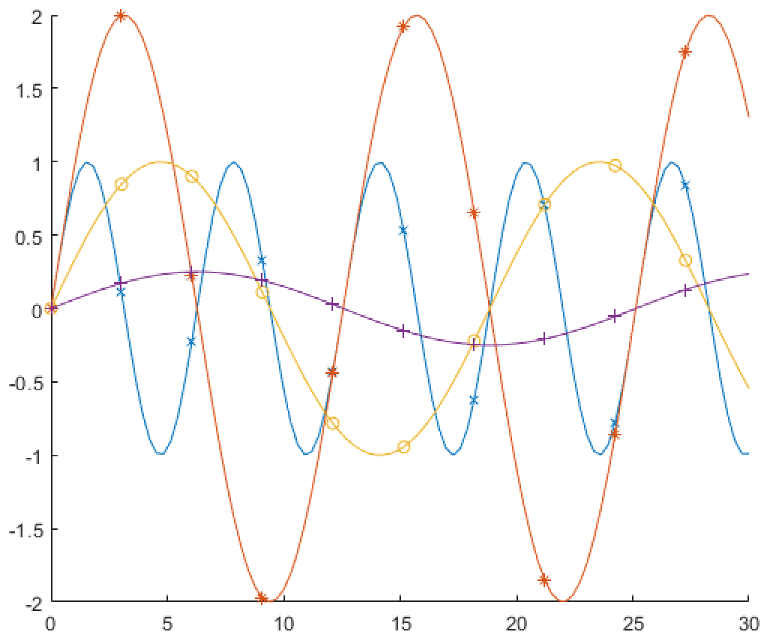
In this example, we have plotted the sine waves over 100 samples, so we would expect to see 100 markers for each plot:

```
x = linspace(0,30, 100);
y1 = sin(x);
y2 = 2*sin(x/2);
y3 = sin(x/3);
y4 = 0.25*sin(x/4);
figure("Units","pixels", "Position", [300 300 600 480])
hold on
plot(x,y1, 'Marker', 'x')
plot(x,y2, 'Marker', '*')
plot(x,y3, 'Marker', 'o')
plot(x,y4, 'Marker', '+')
```



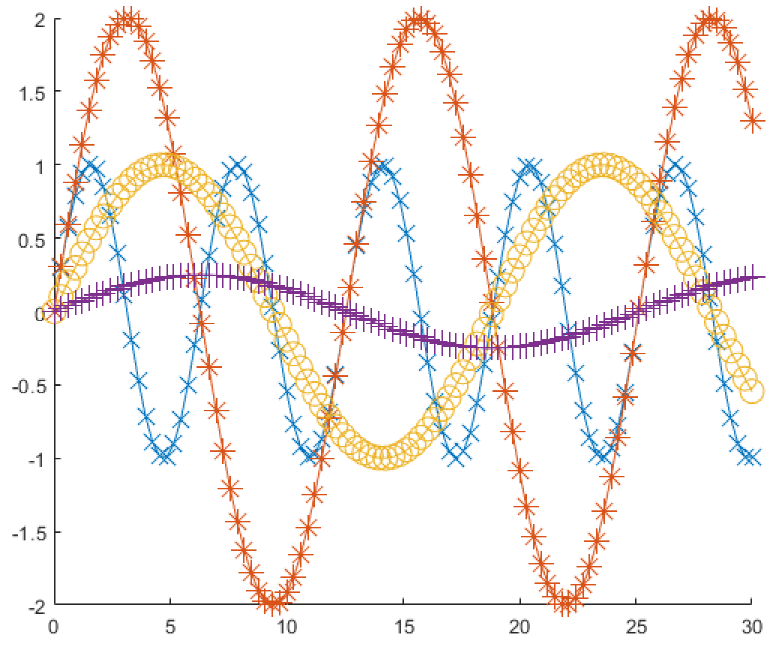
- **`'MarkerIndices':`** You can use this to only print the markers on a subset of the samples.

```
x = linspace(0,30);
y1 = sin(x);
y2 = 2*sin(x/2);
y3 = sin(x/3);
y4 = 0.25*sin(x/4);
subsetIndices = [1:10:100];
figure("Units","pixels", "Position", [300 300 600 480])
hold on
plot(x,y1, 'Marker', 'x', 'MarkerIndices', subsetIndices)
plot(x,y2, 'Marker', '*', 'MarkerIndices', subsetIndices)
plot(x,y3, 'Marker', 'o', 'MarkerIndices', subsetIndices)
plot(x,y4, 'Marker', '+', 'MarkerIndices', subsetIndices)
```



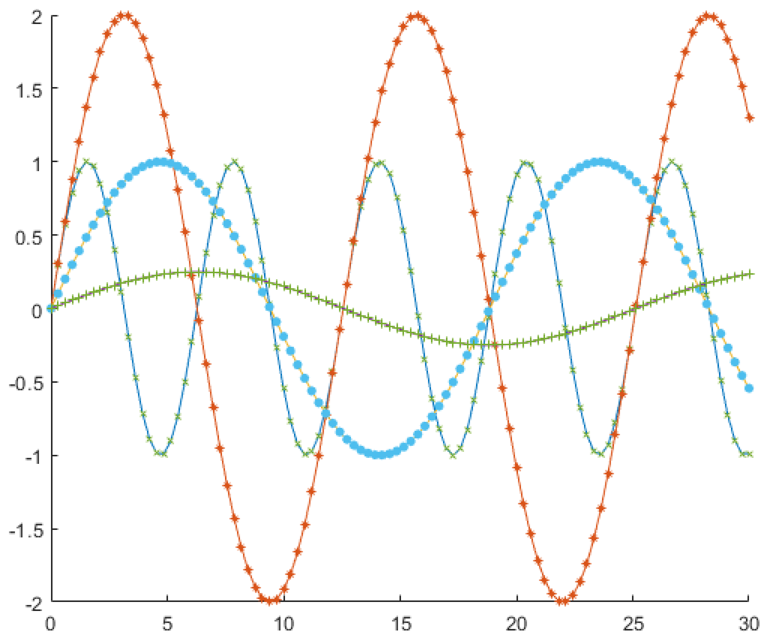
- **`'MarkerSize'`** by default, markers have a size of 6 points. Use this property to change its size.

```
x = linspace(0,30);
y1 = sin(x);
y2 = 2*sin(x/2);
y3 = sin(x/3);
y4 = 0.25*sin(x/4);
figure("Units","pixels", "Position", [300 300 600 480])
hold on
plot(x,y1, 'Marker', 'x', 'MarkerSize', 12)
plot(x,y2, 'Marker', '*', 'MarkerSize', 12)
plot(x,y3, 'Marker', 'o', 'MarkerSize', 12)
plot(x,y4, 'Marker', '+', 'MarkerSize', 12)
```



- **`'MarkerEdgeColor'`** and **`'MarkerFaceColor'`** are used to give markers a different color than the plot. The former changes the colour of the markers edge, the latter, it fills it with a colour as well.

```
 x = linspace(0,30);
 y1 = sin(x);
 y2 = 2*sin(x/2);
 y3 = sin(x/3);
 y4 = 0.25*sin(x/4);
 figure("Units","pixels", "Position", [300 300 600 480])
 hold on
 plot(x,y1, 'Marker', 'x', 'MarkerSize', 4, 'MarkerEdgeColor', '#77AC30')
 plot(x,y2, 'Marker', '*', 'MarkerSize', 4, 'MarkerEdgeColor', '#D95319')
 plot(x,y3, 'Marker', 'o', 'MarkerSize', 4, 'MarkerEdgeColor', '#4DBEEE',
'MarkerFaceColor', '#4DBEEE')
 plot(x,y4, 'Marker', '+', 'MarkerSize', 4, 'MarkerEdgeColor', '#77AC30')
```

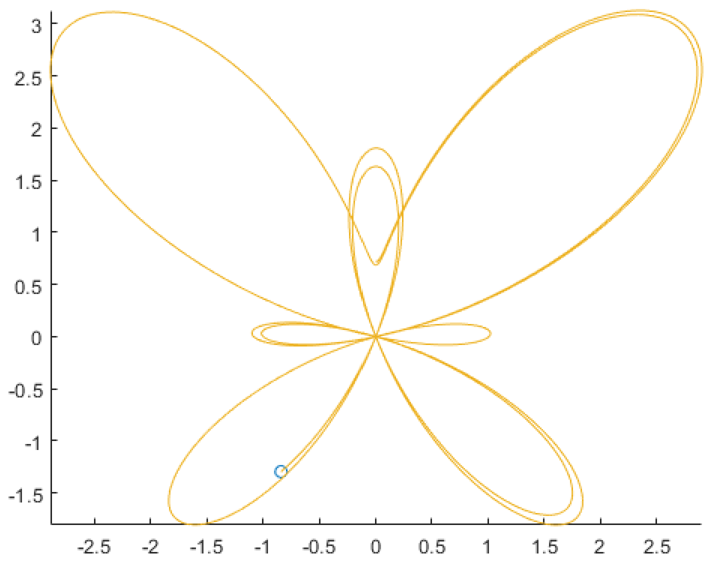


Comet Plot Animation

Sometimes, during presentations or classes, you might need to animate your drawings. MATLAB allows you to animate your plot using many techniques. Here, will only present the simplest one: **`comet`** animation

The comet animation traces and plots the function from beginning to end. The following example plots the butterfly shape as an animated comet.

```
figure
t = 0:0.01:10;
x = sin(t).*(exp(cos(t)) - 2.*cos(4*t) - (sin(t/12)).^5);
y = cos(t).*(exp(cos(t)) - 2.*cos(4*t) - (sin(t/12)).^5);
comet(x, y)
```



3D Plots

In Engineering applications, we often need to visualize functions in the form $z = f(x, y)$ MATLAB offers plenty of functions to plot 3D graphs. You can apply many of the plot options like coloring and titles as we used for 2D plots.

To start, let us plot the function:

$$z = f(x, y) = xe^{(x-y^2)^2+y^2}$$

Obviously, we need to apply the function on every possible value of the pair (x, y), but this is impossible because both would extend to from $-\infty$ to $-\infty$. So we need to specify a range for this function on the xy-plane both for the x-axis and the y-axis. Then we need to create all the possible pairs within this range. This would have been cumbersome, but MATLAB provides the **`meshgrid`** command that would do exactly the same thing in one step.

Let us assume that we want to plot the above function within the range of $[-3, 3]$ in steps of 0.1 for the x-axis, and within the range $[-2, 2]$ in steps of 0.1 for the y-axis, to do so, simply write:

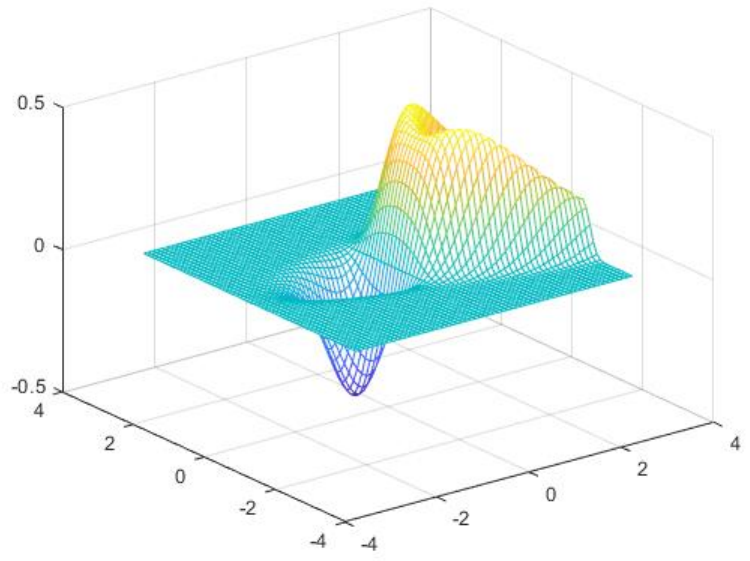
```
[x, y] = meshgrid (-3:0.1:3, -2:0.1:2);
```

But if we want to draw both the x-axis and y-axis to extend to the same range and same increment, you can use:

```
[x, y] = meshgrid (-3:0.1:3);
```

All that remains is to write the function and plot it. MATLAB offers the **`mesh`** command that draws 3D functions.

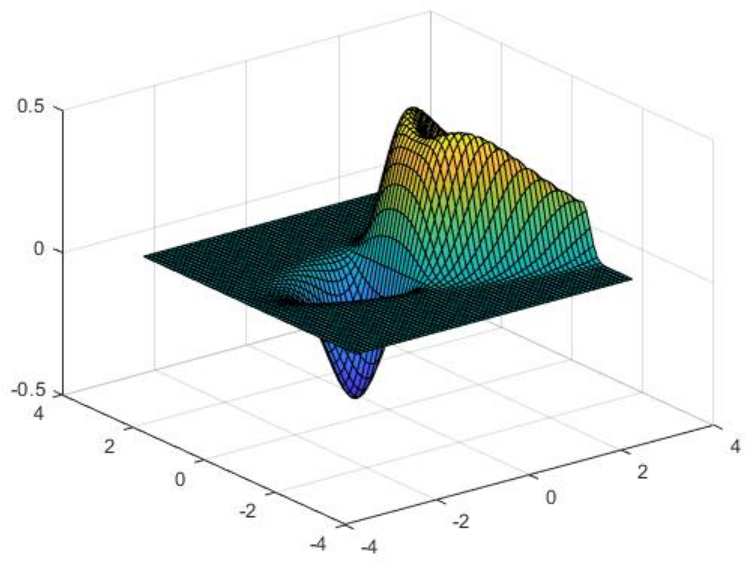
```
figure
z = x.*exp(-1*((x-y.^2).^2 + y.^2));
mesh(x,y,z)
grid on
```



The varying colours illustrate the hills and valleys of the 3D function (local and global minima and maxima points) and they change colour moving toward the peaks and lows of the function.

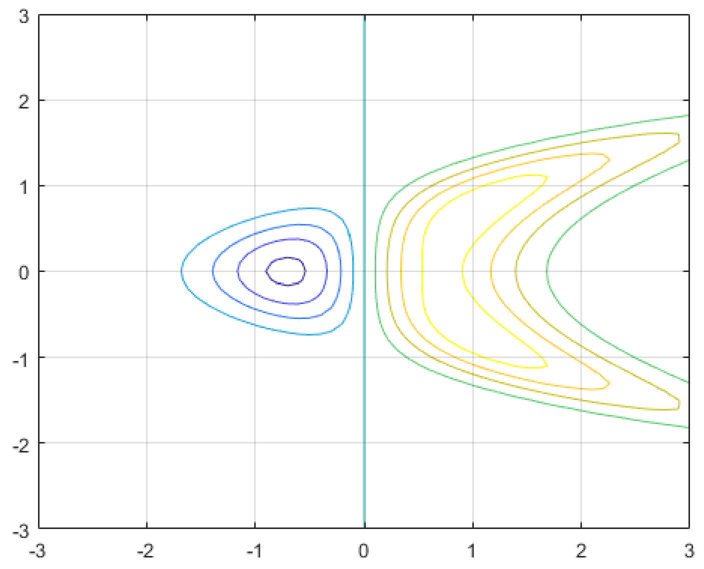
We can redraw this function by plotting the surface of the 3D function using the **`surface`** command:

```
figure
surf(x,y,z);
grid on
```



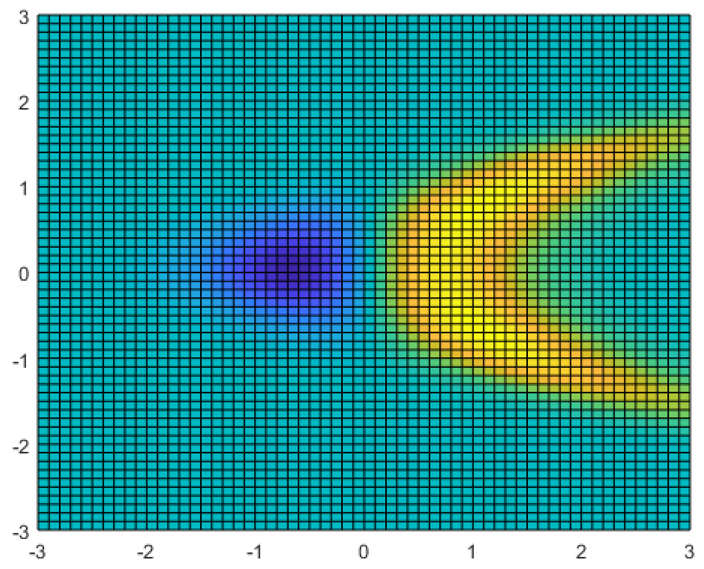
You can draw the function by looking at it orthogonally from the z-axis, and then you see the projection of the values of the function onto the plane:

```
figure
contour(x,y,z);
grid on
```



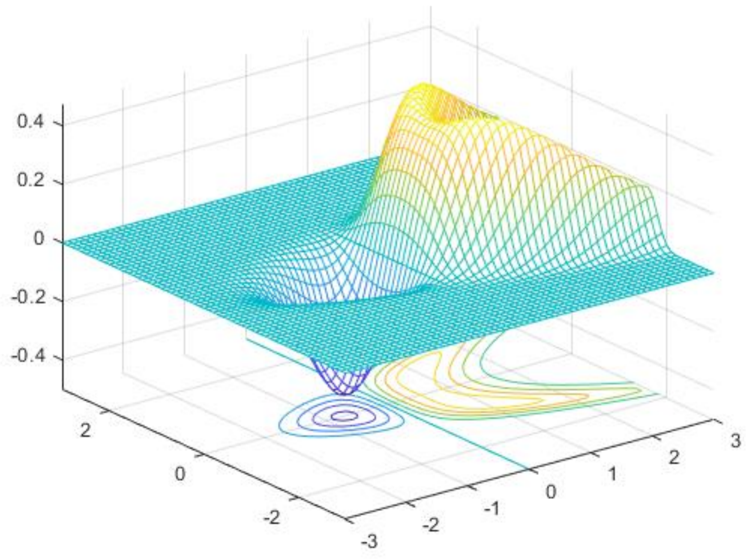
Use the surface command to combine both the surface and contour commands:

```
figure
surface(x,y,z);
```



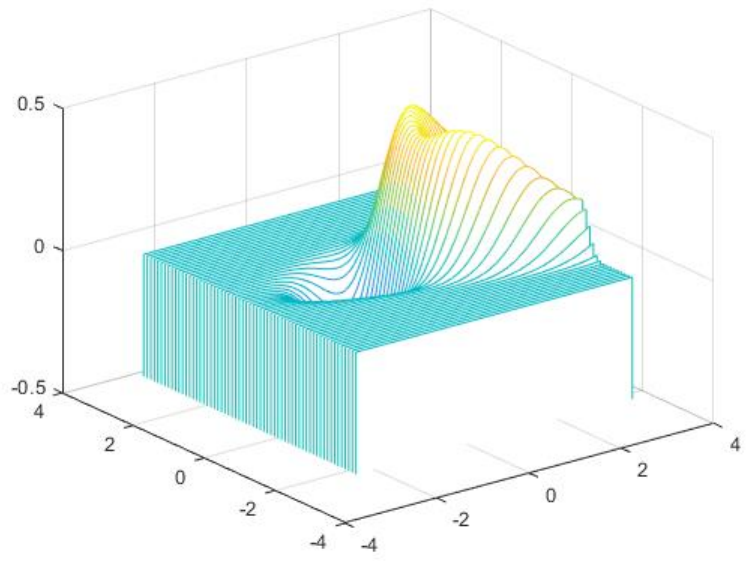
You can also combine the mesh and contour commands in one command:

```
figure
meshc(x,y,z);
grid on
```



And finally, you can use the **`waterfall`** command to draw a 3D plot that looks like a waterfall:

```
figure
waterfall(x,y,z);
grid on
```



Storing MATLAB Figures

Using the **`savefig`** command, you can easily save your figure in a **`.fig`** format that you can open in MATLAB and edit using the plot GUI. You can use **`savefig`** to save the most recent figure or by passing the figure object handle. The **`gcf`** object variable holds the most recent figure drawn.

```
figure
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
savefig(gcf, 'myImage.fig')
```

You can save your figure as an image in numerous image formats using the **`saveas`** command. You can save your plot in **`.jpg`**, **`.png`**, **`.bmp`**, **`.tif`** or vector graphics formats such as **`.pdf`** and **`.eps.`** The **`gcf`** object variable holds the most recent figure drawn.

```
figure
x1 = linspace(-2*pi,2*pi);
y1 = sin(x1);
plot(x1, y1);
saveas(gcf, 'myImage.jpg')
```

Summary for Creating Professional Plots

1. Each axis must be labelled with the name of the quantity being plotted *and its units!*
2. Each axis should have regularly spaced tick marks at convenient intervals —not too sparse, but not too dense—with a spacing that is easy to interpret and interpolate. For example, use 0.1, 0.2, and so on, rather than 0.13, 0.26, and so on.
3. If you are plotting more than one curve or data set, label each on its plot or use a legend to distinguish them.
4. If you are preparing multiple plots of a similar type or if the axes‟ labels cannot convey enough information, use a title.
5. If you are plotting measured data, plot each data point with a symbol such as a circle, square, or cross (use the same symbol for every point in the same data set). If there are many data points, plot them using the dot symbol.
6. Sometimes data symbols are connected by lines to help the viewer visualize the data, especially if there are few data points. However, connecting the data points, especially with a solid line, might be interpreted to imply knowledge of what occurs between the data points. Thus, you should be careful to prevent such misinterpretation.
7. If you are plotting points generated by evaluating a function (as opposed to measured data), do *not use a symbol to plot the points. Instead, be sure to generate many points, and connect the points with solid lines.*

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 5 - Statistics and Probability

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Basic Statistical and Probabilistic Analysis

MATLAB offers numerous functions and tools for statistical and probabilistic processing of data. In this section, we will present the basic functions that MATLAB offers. To start, we will define a vector and an array to work with:

```
A = [1, 5, 7, 8.5, 8, 3.5 , 1.25, 5, 4, 4, 6, 7.75, 1.4, 10, 10, 9.7];
B = [ 4, 5,  5, 7; ...
     10, 0, 12, 4; ...
      2, 1,  4, 9; ...
      5, 6, 10, 7; ...
     12, 7,  8, 0 ];
```

The **min** and **max** commands return the minimum or maximum value in a vector, or the minimum or maximum value of each **column** in a matrix:

```
min(A)
```

```
ans = 1
```

```
max(A)
```

```
ans = 10
```

```
min(B)
```

```
ans = 1×4
     2     0     4     0
```

```
max(B)
```

```
ans = 1×4
    12     7    12     9
```

We use the average (mean) and median as two different measures of the central tendency (central position) for a set of data.

To compute the average of collected observed data in MATLAB, we use the **mean** function.

```
mean(A)
```

```
ans = 5.7563
```

If the input to the mean function is a matrix, the average is by default computed column-by-column:

```
mean(B)
```

```
ans = 1×4
    6.6000    3.8000    7.8000    5.4000
```

Many times, the median (average) does not necessarily convey the correct picture of the underlying data due to data outliers. The fact that the average is susceptible to the influence of outliers is a major disadvantage. Data outliers are the values (observations) that lie an abnormal distance from the other values (observations). This could be due to an actual observation, or sometimes a glitch in measuring the observation. For example, assume we have a company where the annual wages of the 15 employees (security, janitor, cleaning personnel, HR, engineers) and the CEO, CTO, CFO are as follows (in thousands of JODs):

```
annual_salary = [3.6, 3.8, 3.4, 6, 6.5, 6.3, 6.2, 7, 7.5, 7.8, 7.2, 8, 8.2, 8,
8, 40, 36, 36 ];
```

Computing the mean in this case will give us a truly misleading representation! The mean salary will be around 11,639 JODs, where in fact almost all employees get much less income. Do not trust the average, ever!

```
mean(annual_salary)
```

```
ans = 11.6389
```

Another example for outliers is if we have a class of 20 students taking an exam; eighteen students showed up to the exam while two students missed it. The grading system automatically granted zeros to these students. Taking the average might give misleading info about the actual performance of the students. The two zeros are not related to the actual performance of the absent students, it simply means they did not attend. It is up to the data analyst to examine the data before applying any statistical processes. Should you for example consider these outliers as abnormal or exclude them? Or are they expected albeit rare and thus should be considered in the analysis?

The median is the value at which half of the data falls below, and the other half falls above. It is less affected by outliers and skewed data, and thus; in many cases can provide a better way to understand the data. We use the MATLAB function **`median`** to compute the median. So, in our case for the company employees, the median is 7,350 JODs.

```
median(annual_salary)
```

```
ans = 7.3500
```

The median operates on matrices column-wise:

```
median(B)
```

```
ans = 1×4
     5     5     8     7
```

We can get the most frequently occurring value in a set by using the **`mode`** command. For example, the annual income 8,000 JOD is the most frequent salary in the set **`annual_salary.`**

```
mode(annual_salary)
```

```
ans = 8
```

As most MATLAB functions, the mode command operates on vectors or matrices in a column-wise fashion.

```
mode(A)
```

```
ans = 4
```

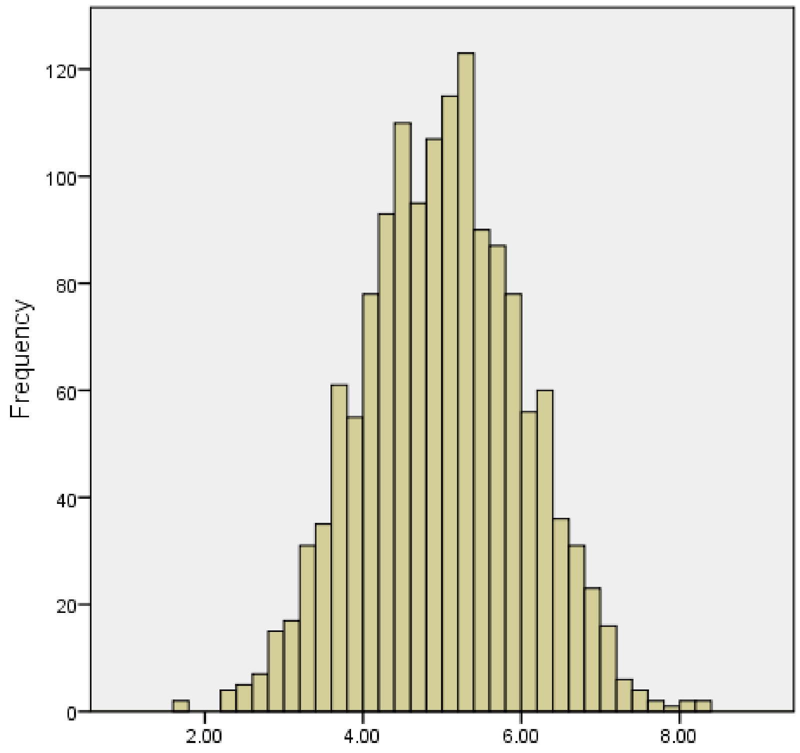
When there are multiple values occurring equally frequently, **`mode`** returns the smallest of those values. For example, in matrix B, each value occurs exactly once in each column, so the **`mode`** function returns the smallest value in each column.

```
mode(B)
```

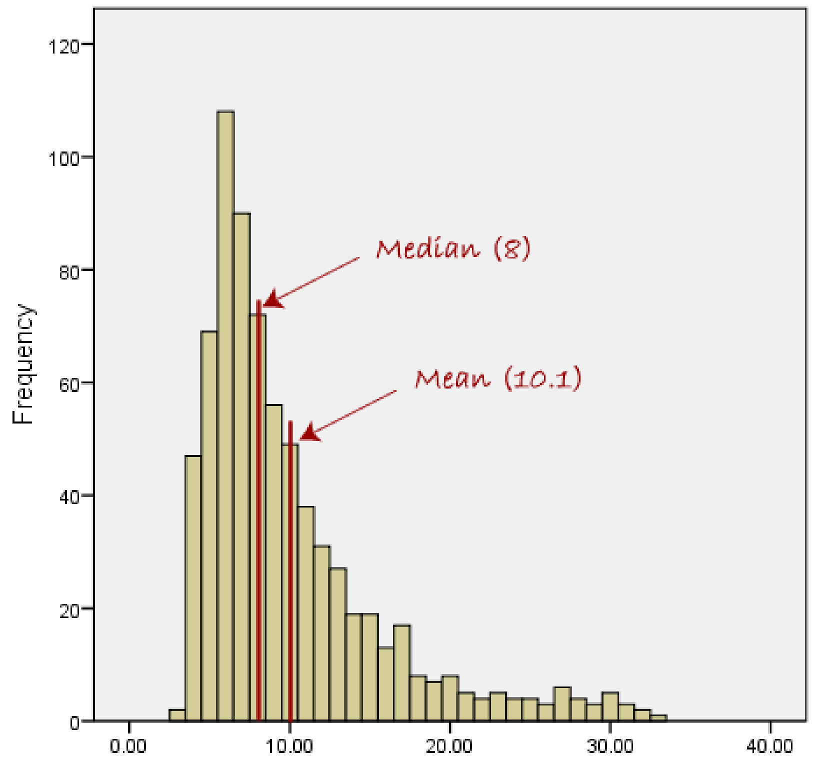
```
ans = 1×4
     2     0     4     7
```

Skewed Distributions and the Mean and Median

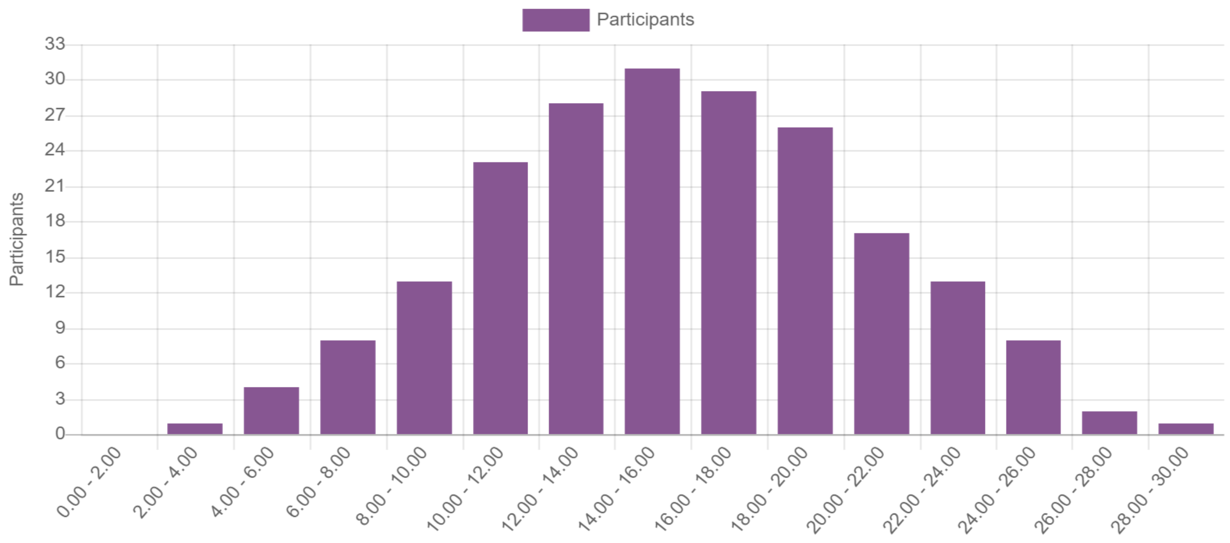
When we have lots of data samples, we often divide them into equal ranges, and count how many samples occur within this range. Basically, creating a histogram (we will learn how to draw histograms later). Understanding a histogram helps us determine which statistical tool is better to use given our data set: the mean or median. If we have a distribution that looks like a normal (Gaussian) distribution, then we can use either the mean or median to present our data. However, in this case, the mean is widely preferred.



However, if the data is skewed towards the right or left, then the values for the median and mean will start to vary. In these cases, the median is generally considered to be the best representative of the central location of the data. The more skewed the distribution, the greater the difference between the median and mean.



The next figure shows the actual distribution of the grades of the course CPE101 taken by 204 students at the University of Jordan (Fall 2020). We can easily observe that the distribution is Gaussian (normal) and almost symmetrical. If the distribution is symmetric, then the mean will in fact equal the median and will be around half the full range. In this particular case, we expect that the mean and median are extremely close, which is in fact the case: 15.6 and 15.7, respectively.



The Standard Deviation and Variance

The standard deviation (σ) provides a measure of how widely or narrowly the values (observations) are away from the mean. It is a measure of the dispersion (spread) of a set of values. We know from the statistics course that around 68.2% of the observations must be between $\pm\sigma$, and that 95.4% of the values must be between $\pm 2\sigma$, and 99.6% of the values must be between $\pm 3\sigma$, and finally 99.8% of the values must be between $\pm 4\sigma$. In our grades example above, the standard deviation of our student grades was 5.4, which means that 68.2% of the students have a grade between 10.2 and 21, and that 95.4% of the students have a grade between 4.8 and 26.4.

In MATLAB, we use the **`std`** command to compute the standard deviation. Similar to the mean and median, it operates on vectors and columns of matrices:

```
std(A)
```

```
ans = 3.0986
```

```
std(B)
```

```
ans = 1×4
    4.2190    3.1145    3.3466    3.5071
```

However, note that the standard deviation provides useful insights when the underlying set is indeed normally distributed. If the data is skewed, then the standard deviation provides little to no information about the underlying data! For example, lets apply the **`std`** command on the **`annual_salary`** data which gives a σ = 11,947 JOD.

```
std(annual_salary)
```

```
ans = 11.9464
```

We know that the mean itself (11,639 JOD) was not reliable in this skewed set in the first place, but if we momentarily ignore this and attempt to apply what we know of the standard deviation, then we can see that 68.2% of the employees will be getting between -308 JOD to 23,586 JOD. Clearly, this is wrong (negative salary), and we know that 15/18 = 83.33% of the employees get below 8,200 JOD. The morale of the story, know when to use these functions and do not just apply them universally for all cases!

We also know from statistics courses that the variance is the square of the standard deviation (σ^2). Similar to the standard deviation, it measures the spread of the data from their mean. In MATLAB, the command var works on vectors or matrices column-wise:

```
var(A)
```

```
ans = 9.6016
```

```
var(B)
```

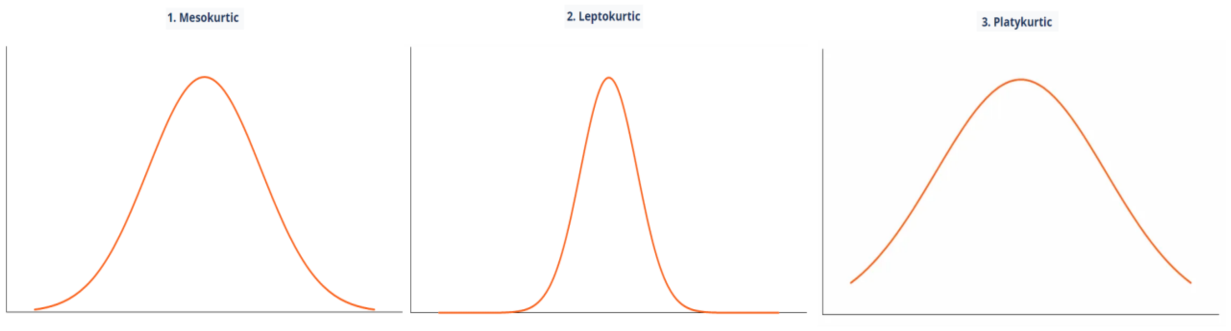
```
ans = 1×4
   17.8000    9.7000   11.2000   12.3000
```

Kurtosis defines how the tails of a distribution differ from the tails of a normal distribution. Kurtosis identifies whether the tails of a given distribution contain extreme values, and determines the heaviness of the tails! The kurtosis of a Gaussian (normal) distribution is 3. The excess kurtosis is defined as:

$$Excess\ \ Kurtosis = Kurtosis - 3$$

There are three types of tail heaviness:

1. **Mesokurtic:** If the excess kurtosis is equal or **<u>very close to zero</u>**, this means that the data follows a normal distribution.
2. **Leptokurtic:** If the excess kurtosis is positive, this means the distribution tails are heavy on either side. This usually means there are large data outliers (extreme positive or negative events). The larger the value means that there are heavier tails and more extreme values
3. **Platykurtic**: If the excess kurtosis is negative, this means the distribution tails are lighter on either side. This usually means there are small data outliers (fewer extreme positive or negative events).



MATLAB has the command **`kurtosis`** that you need to extract three from it to get the excess kurtosis value.

```
kurtosis(A)
```

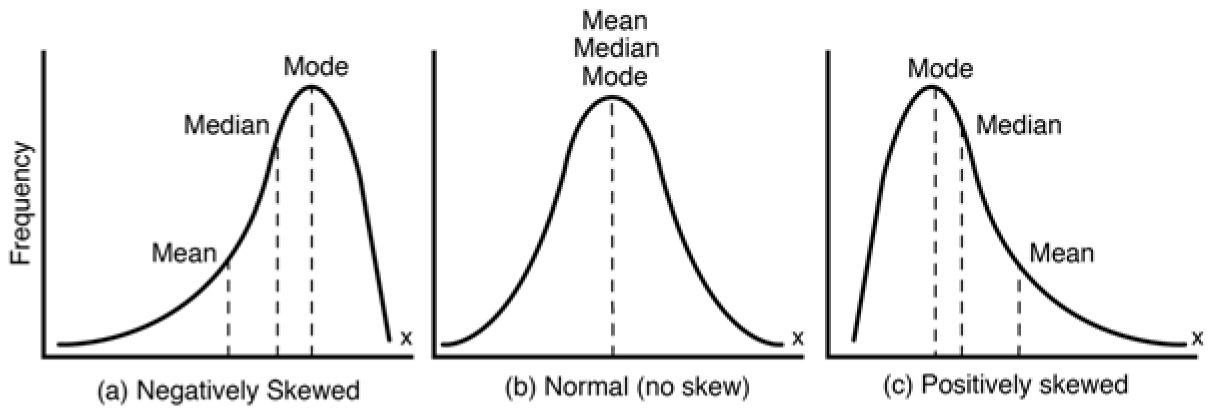
```
ans = 1.7900
```

Analysing Skewed Data

Skewness is a measure of the asymmetry of the distribution of a real-valued data (observations) about their mean. It gives an idea where the peak is located.

1. If the distribution of the dataset is symmetric and Gaussian, then the mean equals the median and also equals the mode. The two tails of the distribution are equal. In this case the skewness is zero.
2. If the distribution of the dataset leans towards the right, then in this case the mode > median > mean, and the skewness is negative indicating that the majority of the values fall to the right and that the distribution tail is to the left.

3. If the distribution of the dataset leans towards the left, then in this case the mean > median > mode, and the skewness is positive indicating that the majority of the values fall to the left and that the distribution tail is to the right.



In MATLAB, we use the command **`skewness`** to understand the underlying distribution. Let's try this:

```
skewness(annual_salary)
```

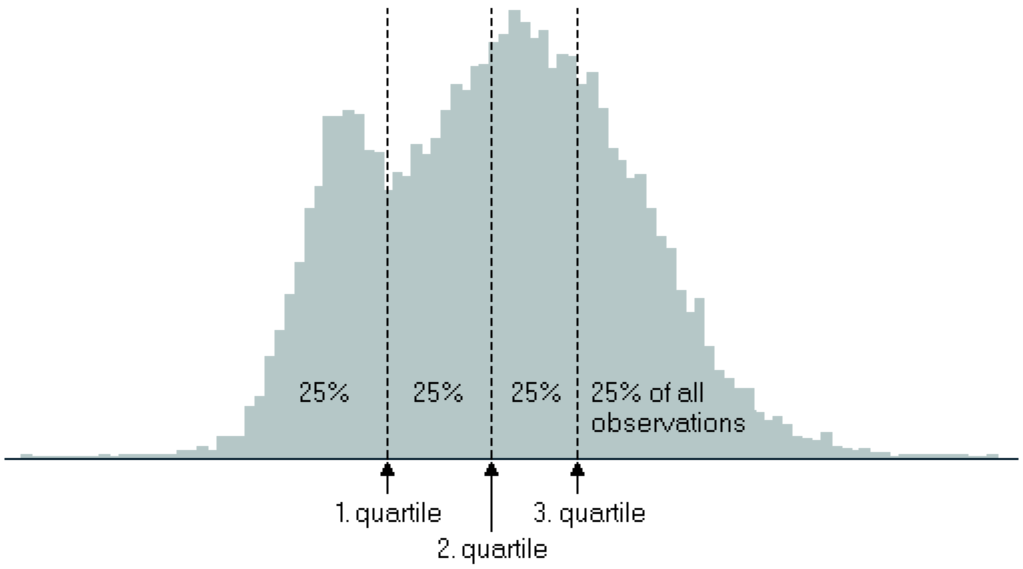
```
ans = 1.7410
```

The positive result indicates that the majority of the data leans to the right, and only few data points (the tail) fall to the right. This is indeed true, for the majority of salaries are well below 8,000 JOD, and the three salaries of the CEO, CTO, and CFO are towards the right!

To better understand what skewness means given the magnitude of its value, we can interpret it as [1]:

- If skewness is less than −1 or greater than +1, the distribution is **highly skewed**.
- If skewness is between −1 and −½ or between +½ and +1, the distribution is **moderately skewed**.
- If skewness is between −½ and +½, the distribution is **approximately symmetric**.

In statistics, a **quartile** divides the number of data points into four parts, or four *quarters*, of more-or-less equal size. Basically, it finds the data points which divides the distribution of data into three points, where 25%, 50%, and 75% of the data are below this point. These points are usually referred to as $Q_1,\ Q_2,\ Q_3$ This metric can be used with both normally distributed and skewed datasets. The median and Q_2 are the same since both cut the data into two halves. The quartile points are not necessarily evenly spaced; for example, more data points can be concentrated in the second and third quarters than the first or fourth quarters. Along with the minimum and maximum of the data, the quartile points provide what is called in statistics the *five-number summary.*



In MATLAB, the **prctile** command (percentile) calculates the points for Q_1, Q_2, Q_3 by passing the 25th, 50th, and 75th percentages:

```
prctile(annual_salary, [25, 50, 75])
```

```
ans = 1×3
    6.2000    7.3500    8.0000
```

In fact, the percentile function **prctile** is generic. We can use it to get the point at which 42% of the data fall below:

```
prctile(annual_salary, 42)
```

```
ans = 7.0120
```

The distance between the third quartile Q_3 and the first quartile Q_1 is called the interquartile range (IQR). MATLAB has the command **iqr** to calculate this range:

```
iqr(A)
```

```
ans = 4.5000
```

```
iqr(B)
```

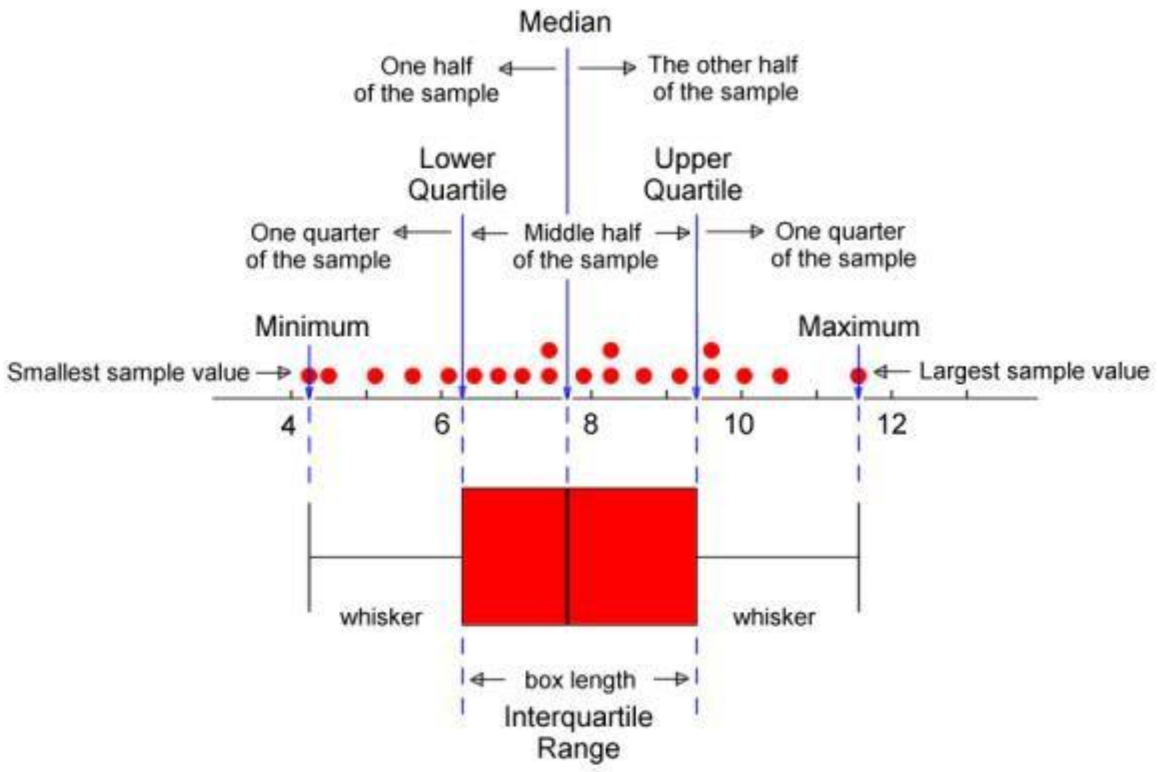
```
ans = 1×4
    7.0000    5.5000    5.7500    4.5000
```

```
iqr(annual_salary)
```

```
ans = 1.8000
```

Understanding and Plotting Boxplots

A **`boxplot`** is a standardized way of displaying the distribution of data based on the five-number summary (“minimum”, first quartile (Q1), median, third quartile (Q3), and “maximum”). An example boxplot is shown below:



The box illustrates the IQR (the data distribution between Q_1 and Q_3). The line in the middle of the box corresponds to the median or the Q_2 point. Here the "minimum" and "maximum" do not necessarily mean the true or actual minimum or maximum. In fact, these two values are sometimes referred to as **whiskers**. They are computed as:

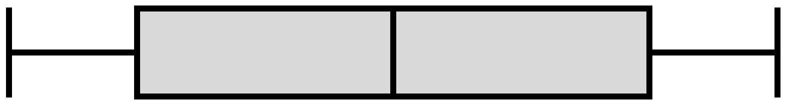
- From above the upper quartile, a distance of 1.5 times the IQR is measured out and a whisker is drawn up to the largest observed point from the dataset that falls within this distance.
- Similarly, a distance of 1.5 times the IQR is measured out below the lower quartile and a whisker is drawn up to the lower observed point from the dataset that falls within this distance.

All values that occur outside this range are considered outliers, or abnormal!

Boxplots visualise the underlying data and help us infer some useful information. For example, the shape of the inner box tells us if the distribution is symmetric or skewed:

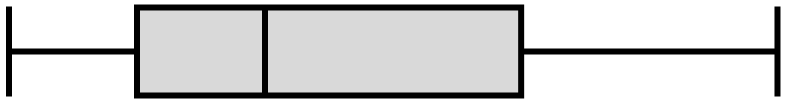
Normal Distribution

(Quartile 3 - Quartile 2) = (Quartile 2 - Quartile 1)



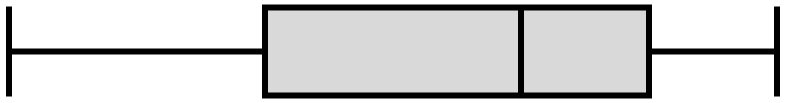
Positive Skew

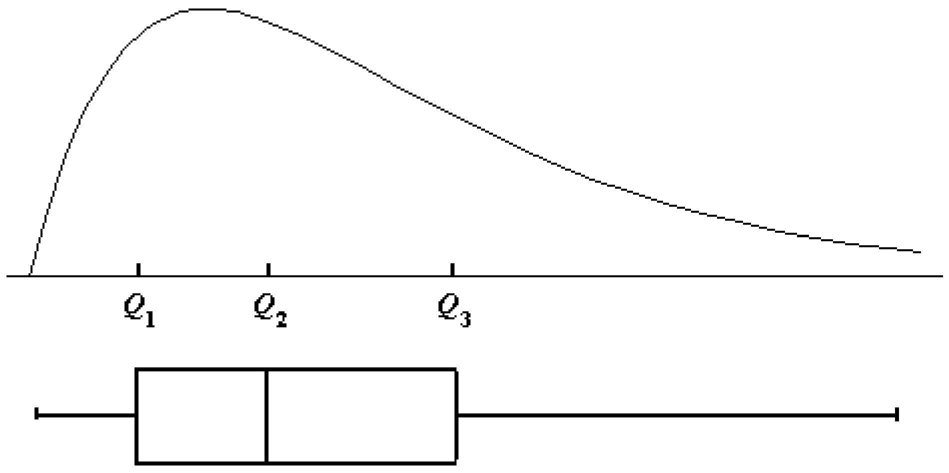
(Quartile 3 - Quartile 2) > (Quartile 2 - Quartile 1)



Negative Skew

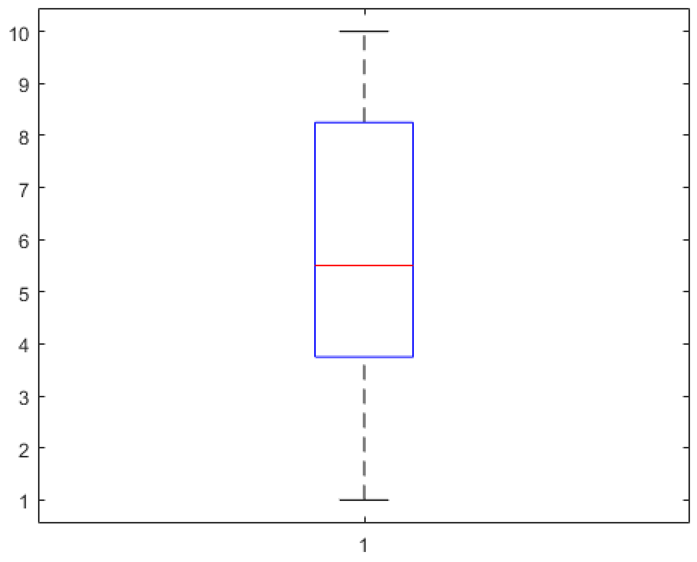
(Quartile 3 - Quartile 2) < (Quartile 2 - Quartile 1)





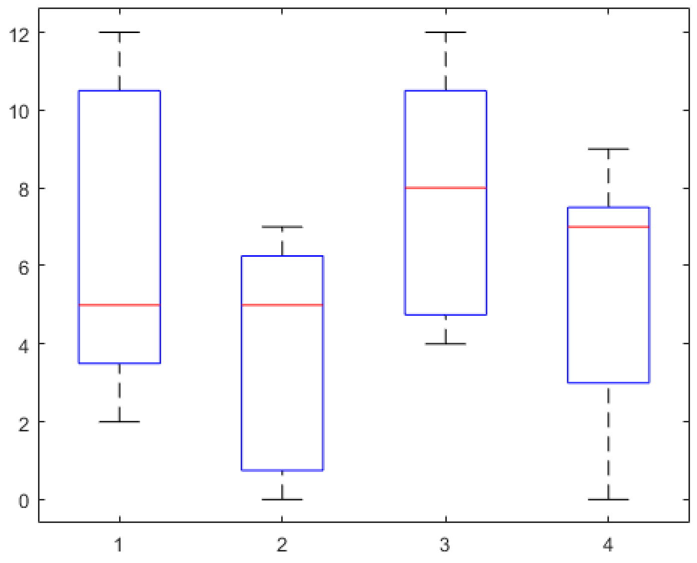
To generate the boxplot in MATLAB simply use the **`boxplot`** command:

```
boxplot(A)
```



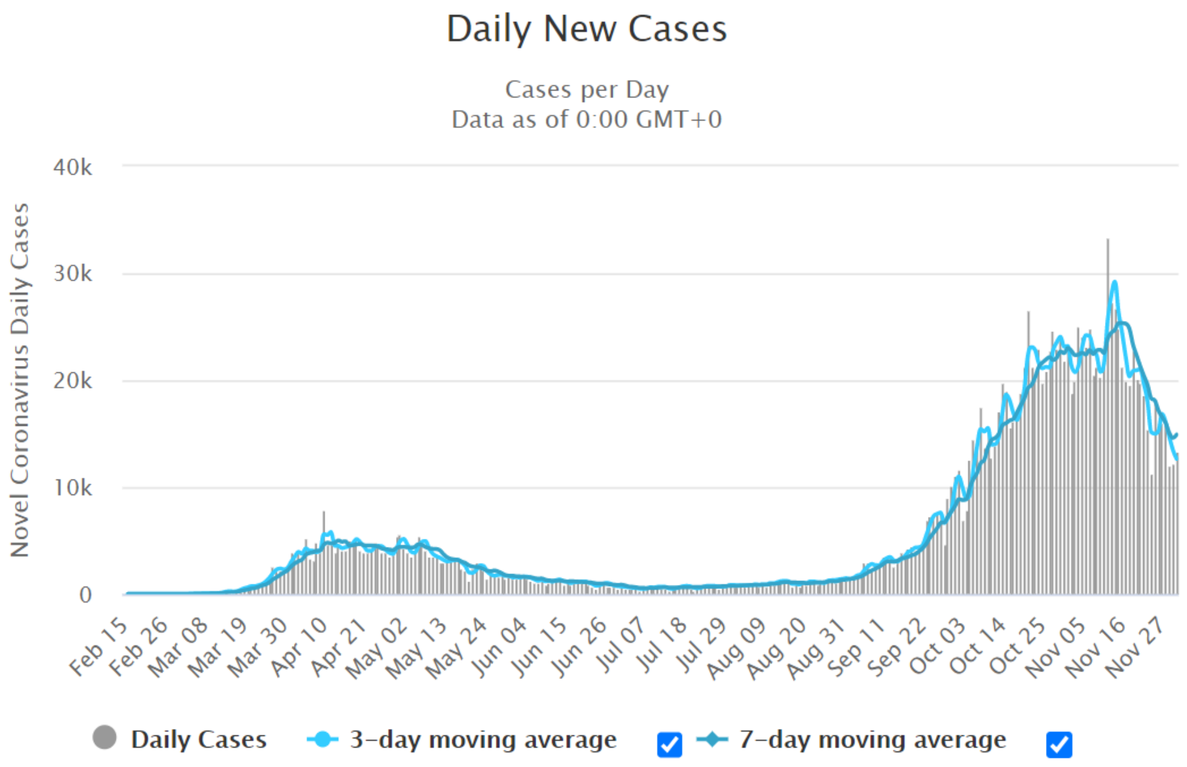
You can draw multiple boxplots for a matrix at once, one for each column:

```
boxplot(B)
```



The Moving Average

The moving average is an extremely powerful tool that extends beyond statistics. For example, it is used to design a special type of filters called MAF (moving average filter) that is widely used in control and embedded systems. The basic principles are the same whether we use the moving average in filtering sensory data or smoothing our data. Probably you have seen during the COVID-19 crisis lots of graphs that visualize the number of infected people on a daily basis. Daily numbers fluctuate and do not easily convey trends of what is going on. Many times, you would see figures showing a 3-Day average, or a 7-Day average of infections. Each day, the average is computed over the last three or seven days, so the average keeps moving with the data. The oldest day data is dropped and the new day data is used instead. The graph below shows the number of daily infections as well as the smoothed 3-Day and 7-Day average for the UK:



MATLAB offers the command **`movsum`** which takes the data and the window length we want to use to sum all the data in.

```
movsum(A, 3)
```

```
ans = 1×16
     6.0000    13.0000    20.5000    23.5000    20.0000    12.7500     9.7500 ⋯
```

In the above example, MATLAB sums the first three values, then it sums the 2nd, 3rd and 4th values, then it sums the 3rd, 4th, and 5th values and so on moving in a window of size three. If you want to calculate the moving average, simply divide the results by the window length.

```
movsum(A, 3) / 3
```

```
ans = 1×16
    2.0000    4.3333    6.8333    7.8333    6.6667    4.2500    3.2500 ⋯
```

```
movsum(A, 7) / 7
```

```
ans = 1×16
    3.0714    4.2143    4.7143    4.8929    5.4643    5.3214    4.8929 ⋯
```

Drawing Histograms

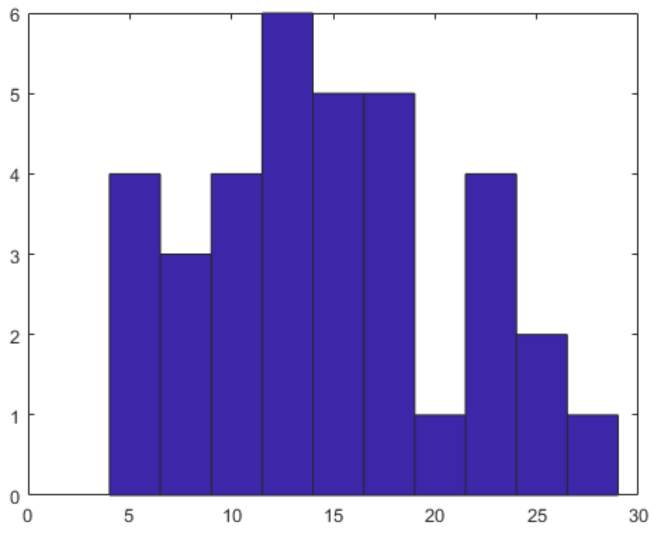
If you have collected a set of data values (student grades, company salaries, sensor values related to processor internal temperature over time), you need to visualize this data to better understand it and make good use of it. A histogram is a display of statistical information that uses rectangles to show the frequency of data items in successive numerical intervals of equal size. Histograms are useful in studying data properties and distributions as they can be used to approximate the probability function.

Suppose a class of 35 students have the grades:

```
grades = [18, 16, 16, 17, 20, 22, 15, 15, 24, 14, 13, 12, 11, 11, 10, 24, 8, 4,
4, 5, 17, 22, 7 , 16, 17, 17, 25, 7 , 13, 12, 13, 11, 26, 29, 6];
```

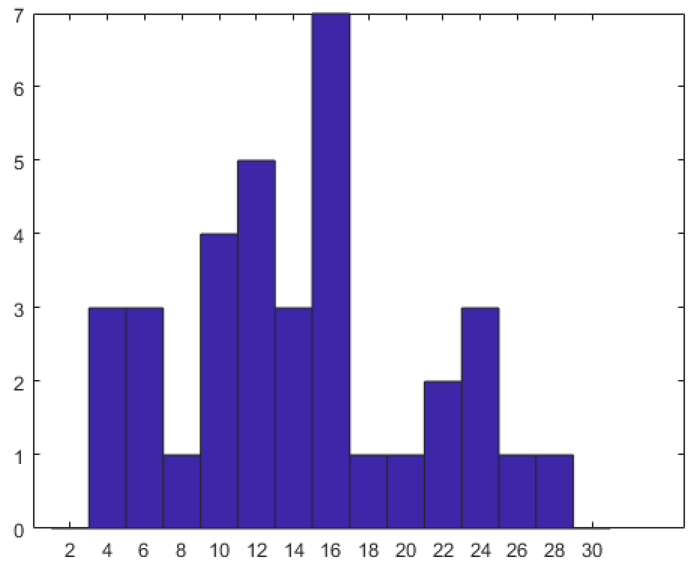
To create a histogram of this data in MATLAB, use the **`hist`** command. By default, MATLAB creates a histogram with 10 bars distributed evenly between the maximum and minimum values. In this case, the range for the histogram is [4, 29] and the width of each bin is ((29-4) / 10) = 2.5. Therefore, each bar is centered at 4 + 1.25 = 5.25 (inital point) and then each successive bar center is 2.5 apart from the others.

```
hist(grades)
```



We can manually specify the number and centre of each bar in the histogram by passing the centres to the **hist** command:

```
hist(grades, [2:2:30])
```



In the previous example, we created 15 bins cantered at 2, 4, 6, ... 30.

If we want to get the frequency count and the location of each bin centre, we simply have to save them as:

```
[count, centers] = hist(grades, [2:2:30])
```

```
count = 1×15
     0     3     3     1     4     5     3     7     1     1     2     3
1 ⋯
centers = 1×15
     2     4     6     8    10    12    14    16    18    20    22    24
26 ⋯
```

So, count basically tells us how many elements are in each bar; the frequency of elements occurring within the plotted bar range.

If instead you already have the frequency of the data instead of the actual data, you can plot a histogram using the **bar** plot. Suppose we have the following grade distribution for a certain course:

Grades	No. Students
45 - 49	10
50 – 54	12
55 - 59	13
60 - 64	15
65 - 69	18
70 - 74	19
75 - 79	14
80 - 84	10
85 - 89	6
90 – 94	1
95 - 100	0

To plot the data as a histogram, we can do the following:

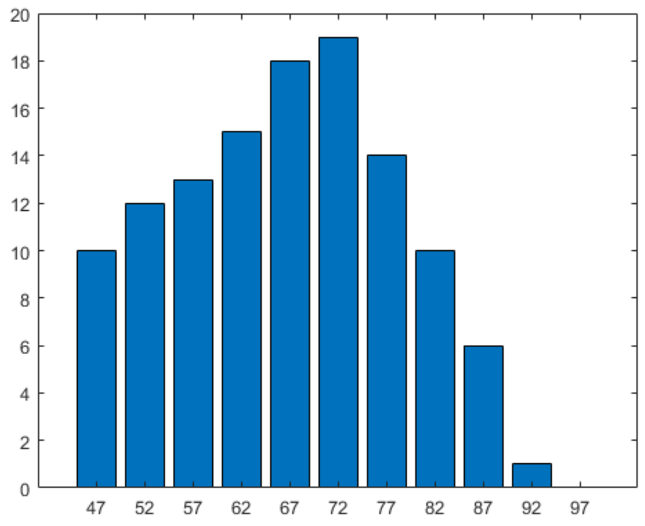
```
y = [10, 12, 13, 15, 18, 19, 14, 10, 6, 1, 0]
```

```
y = 1×11
    10    12    13    15    18    19    14    10     6     1     0
```

```
x = [47:5:97]
```

```
x = 1×11
    47    52    57    62    67    72    77    82    87    92    97
```

```
bar(x,y)
```



Random Number Generation (RNG)

In many engineering and scientific experiments, we need to generate random data to work with. MATLAB provides several functions to generate random data according to different specifications.

Uniformly Distributed Numbers

The **`rand`** command generates random numbers in the interval (0,1) from a uniform distribution. That is, each number has an equal chance of showing up in the sequence like the other.

```
x = rand
```

```
x = 0.0975
```

To generate a square array of uniformly distributed random numbers, use **`rand(n)`**:

```
x = rand(3)
```

```
x = 3×3
    0.2785    0.9649    0.9572
    0.5469    0.1576    0.4854
    0.9575    0.9706    0.8003
```

To generate an array of uniformly distributed random numbers of any size $m \times n$, use **`rand(m, n)`**:

```
x = rand(2, 4)
```

```
x = 2×4
    0.1419    0.9157    0.9595    0.0357
    0.4218    0.7922    0.6557    0.8491
```

If you close MATLAB, and run the above commands again, you will notice that you will get the same numbers over and over. This is because MATLAB uses the same seed to its random number generator algorithms. In some cases, this might be useful if you are still debugging or developing codes, but most of the time, you need truly random numbers to appear each time your codes run. You must instruct MATLAB to change its random number generator algorithm seed and make it rely on a new value each time. To do so, at the beginning of your programs, use the command:

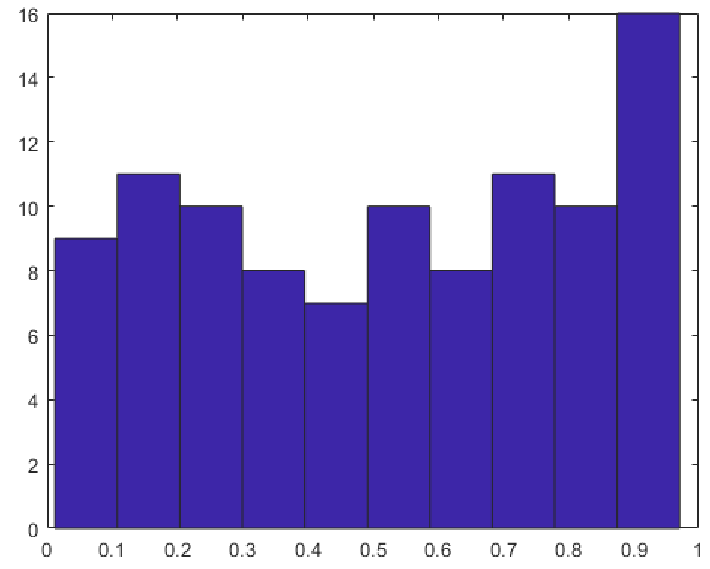
```
rng('shuffle')
```

If you want to fix the seed and start over with the same sequence over and over, use a fixed number:

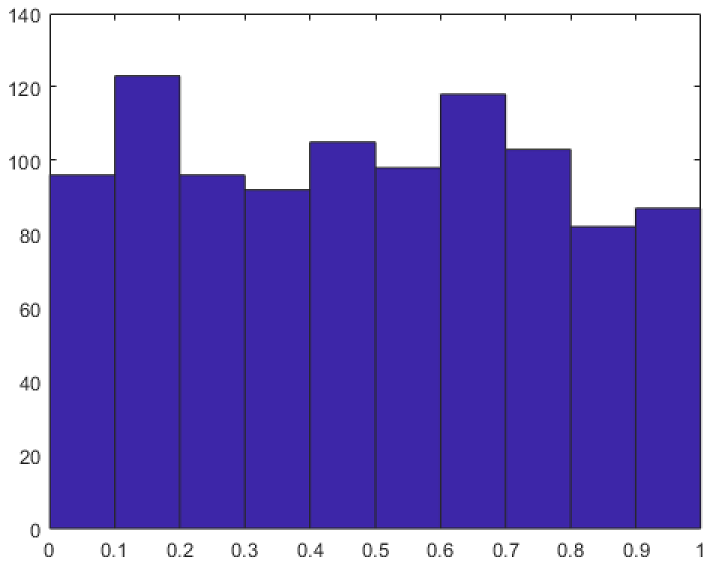
```
rng(0)
```

To understand what a uniformly distributed number means, lets try to visualize the generated random numbers. We will create 100, 10000, and 10,000 uniformly random numbers and plot them using a histogram plot:

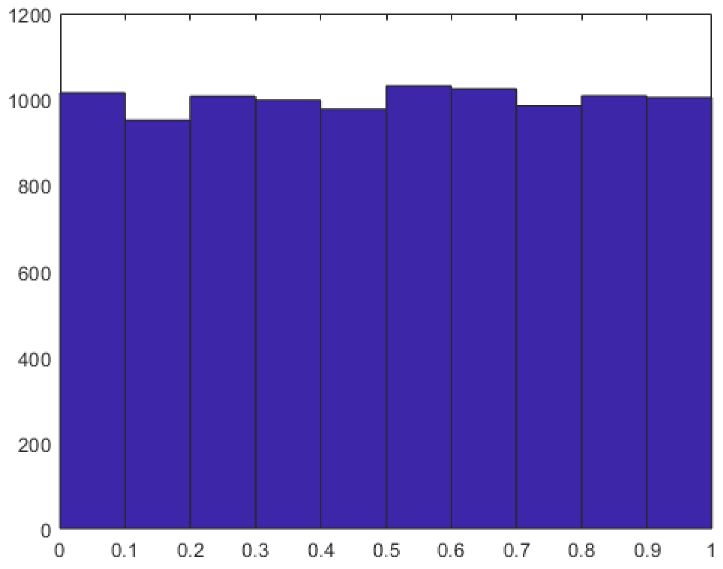
```
hist(rand(1, 100))
```



```
hist(rand(1, 1000))
```



```
hist(rand(1, 10000))
```



You will notice that more-or-less, the same number of values appear in each range. The more data we generate and plot, the closer the frequencies are each to each other and the more uniform the plot looks.

Uniformly Distributed Pseudorandom Integers

The function **`randi(n)`** returns a pseudorandom scalar integer between 1 and n.

```
x = randi(50)
```

```
x = 38
```

To generate a square array of uniformly distributed pseudorandom integers between 1 and n, use **`randi(n, a)`** where **`a`** is the size of the square array:

```
x = randi(50, 3)
```

```
x = 3×3
    41    43    43
    17    30    35
    29    26     6
```

To generate an array of uniformly distributed pseudorandom numbers between 1 and n of size $a \times b$, use **`randi(n, a, b)`**:

```
x = randi(50, 2, 4)
```

```
x = 2×4
    26    23     3    10
     1    46    48    19
```

if you want to change the interval such that it starts from a different value than 1, then simply specify the interval as:

```
x = randi([-10, 10], 2, 4)
```

```
x = 2×4
    -6    -1    -8    -5
    -8    -5     0    -3
```

Normally (Gaussian) Distributed Random Numbers

The **`randn`** command generates random numbers in the interval (0,1) from a Gaussian distribution. That is, the probability of each number showing up in the sequence follows a normal distribution probability:

```
x = randn
```

```
x = 0.1242
```

To generate a square array of normally distributed random numbers, use **`randn(n)`**:

```
x = randn(3)
```

```
x = 3×3
    0.1644   -0.3978   -1.3075
   -0.3501   -0.2564   -1.1253
   -0.2853   -0.9355    0.5279
```

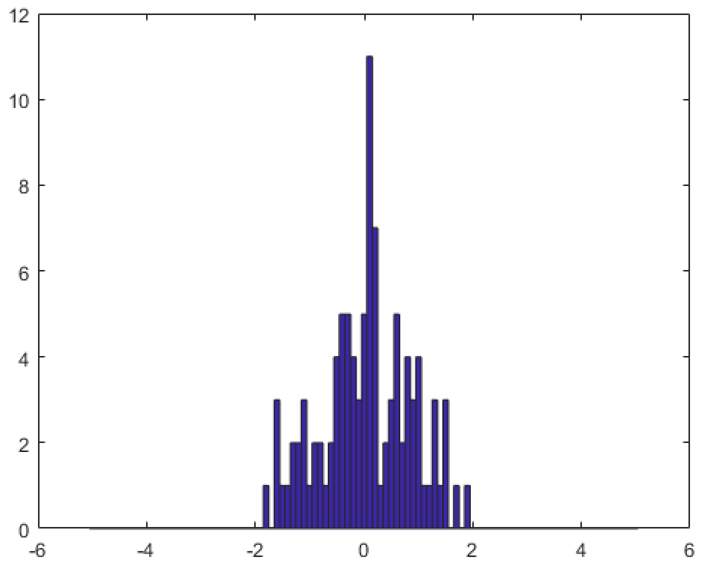
To generate an array of normally distributed random numbers of any size $m \times n$, use **`randn(m, n)`**:

```
x = randn(2, 4)
```

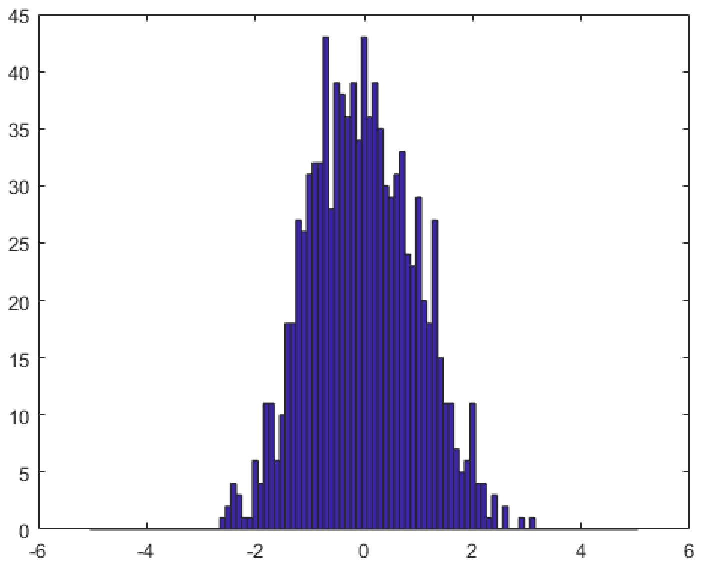
```
x = 2×4
    0.0054    1.1800   -1.0567   -0.2111
    0.8999   -0.7637   -1.8606    0.6913
```

To understand what a normally (Gaussian) distributed number means, lets try to visualize the generated random numbers. We will create 100, 10000, and 10,000 normally distributed random numbers and plot them using a histogram plot:

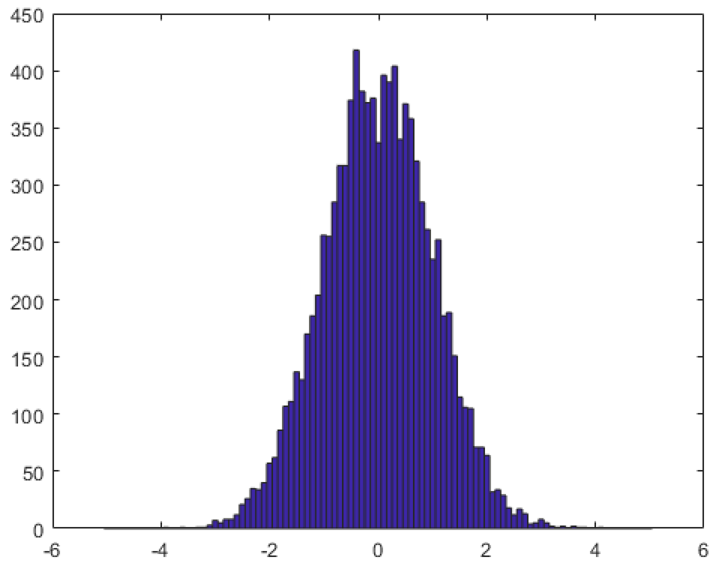
```
hist(randn(1, 100), [-5:0.1:5])
```



```
hist(randn(1, 1000), [-5:0.1:5])
```



```
hist(randn(1, 10000), [-5:0.1:5])
```



Random Permutations of Integers

The function **randperm(n)** returns a row vector containing a random permutation of the integers from 1 to **n** without repeating elements. So, in order to create 10 random permutations of the numbers 1 to 6 containing all the numbers from 1 to 6 once each:

```
for i = 1:10
    randperm(6)
end
```

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

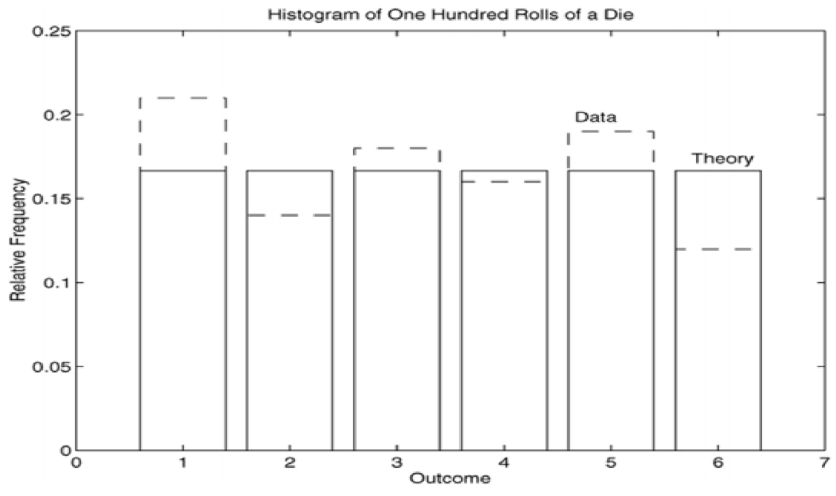
The function **`randperm(n, k)`** returns a row vector containing **k** unique integers selected randomly from 1 to n. So, in order to create 10 random permutations of the numbers 1 to 6 containing numbers from 1 to 6 with each vector having a size of 4:

```
for i = 1:10
    randperm(6, 4)
end
```

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

Probability

In any experiment or phenomena, the probability of an event is a number between 0 and 1 that indicates the likelihood of that event to occur if the experiment is repeated infinitely. For example, if we roll the dice an infinite number of times, theoretically, each side has an equal chance of showing up which is 1/6. However, if we repeat rolling the dice hundreds or thousands of times, record the data, and draw a histogram, the values will be close to 1/6 but not necessarily 1/6.



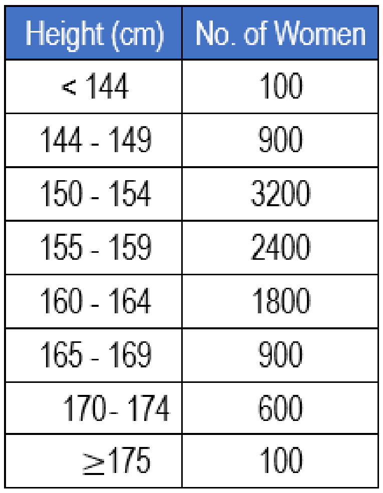
In the course Probability and Random Variables, you have learnt that a random variable is the term used to express the outcome of an experiment, so in the rolling dice experiment, the random variable X denoting the outcome will take any of the values 1, 2, 3, 4, 5, 6. In this experiment, the random variable X is discrete. Some other experiments have random variables which can take on continuous range.

Random variables are usually associated with functions or distributions to characterize the experiment they come from. For example, in the rolling dice example, since the probability of each one of the six numbers showing up is equal, then the random variable X can be characterized by a uniform distribution.

Depending on the experiment, the outputs, and their frequencies, the distributions or functions used to characterise the variables differ. These functions are called probability density functions (pdf).

Probability Density Functions (PDF)

Suppose we have collected the height of 10,000 Jordanian women over the age of 18 and the results were as shown in the table below:



Height (cm)	No. of Women
< 144	100
144 - 149	900
150 - 154	3200
155 - 159	2400
160 - 164	1800
165 - 169	900
170- 174	600
≥175	100

In terms of probability, we can use this table to calculate the probability that a Jordanian woman is shorter than 144cm as 100/10,000 = 1%, while the probability of a Jordanian woman to be between 155 to 159 cm is 2400 / 10,000 = 24%. Remember that the sum of all probabilities must equal 1, because the area under the pdf curve must be equal to 1.

We can visualize this data and create a *pdf* function describing the probabilities as follows:

```
h = [100, 900, 3200, 2400, 1800, 900, 600, 100]
```

```
h = 1×8
          100          900         3200         2400         1800          900 ⋯
```

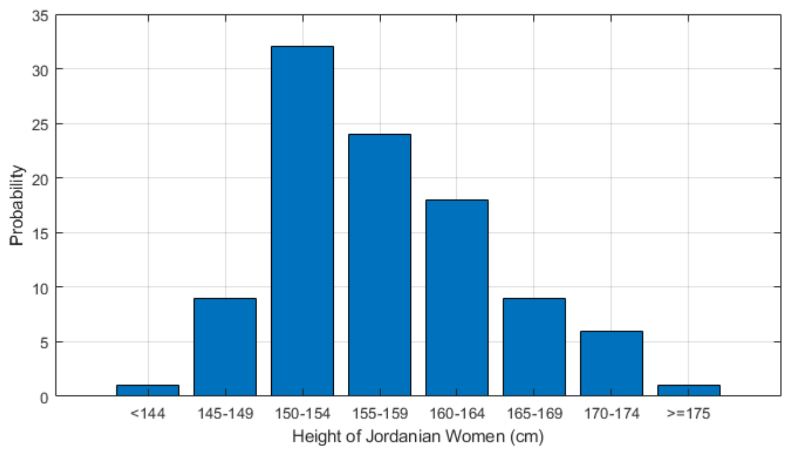
To generate the probabilities, we simply divide the data by the total samples and multiply by 100%:

```
h_pdf = (h / 10000) * 100
```

```
h_pdf = 1×8
      1      9     32     24     18      9      6      1
```

If we plot the resulting data, we will have the probability density function from which we can readily compute the probability of any Jordanian woman being of a certain height, Notice that a *pdf* is a normalized histogram; that is; a histogram whose data is divided by the total number of samples.

```
figure("Units","pixels", "Position", [300 300 800 400])
x = 1:8;
bar (x, h_pdf)
grid on
xticklabels({'<144', '145-149', '150-154', '155-159', '160-164', '165-169',
'170-174', '>=175'})
xlabel("Height of Jordanian Women (cm)")
ylabel("Probability")
```



Cumulative Density Functions

What if we want to know the probability of a Jordanian woman being of height equal or less than 159 cm?

This means we have to add the probability that she might be less than 144cm, or between 145 - 149, or between 150 - 154, or between 155 - 15; that is 0.01 + 0.09 + 0.32 + 0.24 = 0.66 (66%).

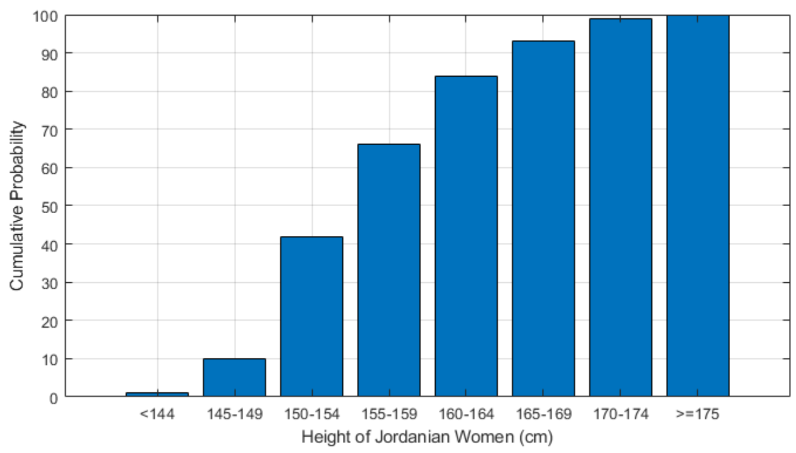
But is there an easier way? We can compute the probability of a woman being of less than or equal a certain height by using the cumulative density function (*cdf*). Remember that in *cdf*, each probability is added to the cumulative sum before it. In MATLAB, we have the function **`cumsum`** which we can use to generate a *cdf* from a *pdf*:

```
h_cdf = cumsum(h_pdf)
```

```
h_cdf = 1×8
      1     10     42     66     84     93     99    100
```

To plot the *cdf* function, we can do the same thing:

```
 figure("Units","pixels", "Position", [300 300 800 400])
 x = 1:8;
 bar (x, h_cdf)
 grid on
 xticklabels({'<144', '145-149', '150-154', '155-159', '160-164', '165-169', 
'170-174', '>=175'})
 xlabel("Height of Jordanian Women (cm)")
 ylabel("Cumulative Probability")
```



Notice how the *cdf* function steadily approaches one. Also notice the cumulative probabilities are shown on the x-axis. You can easily see that the probability of a Jordanian woman being of height less than or equal to 159 is 66%.

References:

[1] Bulmer, M. G. 1979.*Principles of Statistics*. Dover.

Photo References:

https://statistics.laerd.com/statistical-guides/measures-central-tendency-mean-mode-median.php

https://en.wikipedia.org/wiki/Skewness

Experiment version 1.01
Last Updated December 3rd, 2021
Dr. Ashraf Suyyagh – All Rights Reserved

Revision History

Ver. 1.01

- For the Mesokurtic property, I added that a value very close to zero is practically considered Mesokurtic

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 6 - Solving Linear Equations, Basics of Linear Regression and Curve Fitting, and Interpolation

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Solving Linear Equations in Matrix Form

A linear equation can be represented as a vector made of the coefficients of its terms. A system of linear equations can be represented by stacking the vectors of each linear equation on top of each other forming a matrix.

Representing Linear Equations in MATLAB

A linear equation $f_n(x) = a_n x^n + a_{n-1}x^{n-1} + \dots + a_2x^2 + a_1x + a_0$ can be presented in vector form as the coefficients vector $[a_n,\ a_{n-1},\ a_{n-2}, \dots, a_2,\ a_1,\ a_0]$.

So, the function $f_1(x) = x^5 - 3x^4 + 2x^3 - x^2 + x + 2$ can be expressed in MATLAB as:

```
c1 = [1, -3, 2, -1, 1, 2];
```

while the function $f_2(x) = 3x^2 + x^4 - 5$ can be expressed in MATLAB as:

```
c2 = [1, 0, 3, 0, -5];
```

Notice that we first reordered the terms to start from the highest order $x^4 + 3x^2 - 5$, and the coefficients for the missing terms x^3 and x are written as 0.

If we know the coefficients of each term, we can evaluate the function $f(x)$ for any x using the MATLAB command **`polyval`**. The **`polyval`** command takes as input the coefficients vector and the x at which we want to evaluate $f(x)$. Suppose you want to compute $f_1(2.5)$ in the previous function. If you already have the coefficients vector, you can simply write:

```
y = polyval(c1, 2.5)
```

```
y = 9.9688
```

System of Linear Equations

A system of m linear equations with n variables can be expressed as:

$$\begin{aligned}
&a_n x + a_{n-1} y + ... + a_2 z + a_1 w = a_0 \\
&b_n x + b_{n-1} y + ... + b_2 z + b_1 w = b_0 \\
&. \\
&c_n x + c_{n-1} y + ... + c_2 z + c_1 w = c_0 \\
&d_n x + d_{n-1} y + ... + d_2 z + d_1 w = d_0
\end{aligned} \tag{1}$$

The above system can be expressed in generic matrix form as a matrix holding the coefficients of the equations (the left-hand side), a vector of the unknowns $x,\ y,\ ...\ z,\ w$, and a vector that holds the right hand-side. Notice that we moved all literals $a_0,\ b_0,\ ...\ c_0,\ d_0$ to the right-hand side, so only the unknowns remain on the left-hand side before we transformed the equations into matrix form.

$$\begin{bmatrix} a_n & a_{n-1} & \cdots & a_1 \\ b_n & b_{n-1} & \cdots & b_1 \\ . & . & . & . \\ c_n & c_{n-1} & \cdots & c_1 \\ d_n & d_{n-1} & \cdots & d_1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ . \\ z \\ w \end{bmatrix} = \begin{bmatrix} a_0 \\ b_0 \\ . \\ c_0 \\ d_0 \end{bmatrix}$$

In order to understand the notation better, let us write a numeric example. Suppose we have a system of three equations and three unknowns $x,\ y,\ z$ like this:

$$\begin{aligned}
&x - 2y + 3z = 9 \\
&3y - x - z + 6 = 0 \\
&5z + 2x - 5y = 17
\end{aligned}$$

Before expressing the system in matrix form correctly, we need to reorder the terms in all three equations so that the variables are aligned. That is, the order of the variables $x, \ y, \ z$ must be the same in all three equations. Also, all literals must be moved to the right-hand part:

$$x - 2y + 3z = 9$$
$$-x + 3y - z = -6$$
$$2x - 5y + 5z = 17$$

We can express it in matrix form as:

$$\begin{bmatrix} 1 & -2 & 3 \\ -1 & 3 & -1 \\ 2 & -5 & 5 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 9 \\ -6 \\ 17 \end{bmatrix}$$

Notice that we end up with two numeric matrices, one to the left holding the coefficients, and one to the right holding the literals. In MATLAB, we write them as:

```
coef = [ 1 -2  3;  ...
     -1  3 -1; ...
      2 -5  5];
literals = [9;  -6; 17];
```

To find the values of the three unknowns and solve the system, there are two methods; the first is by using the inverse (the order is important to satisfy the matrix equations requirements):

```
b = inv(coef)*literals
```

```
b = 3×1
     1
    -1
     2
```

The other method is simply by using the ***backward*** division:

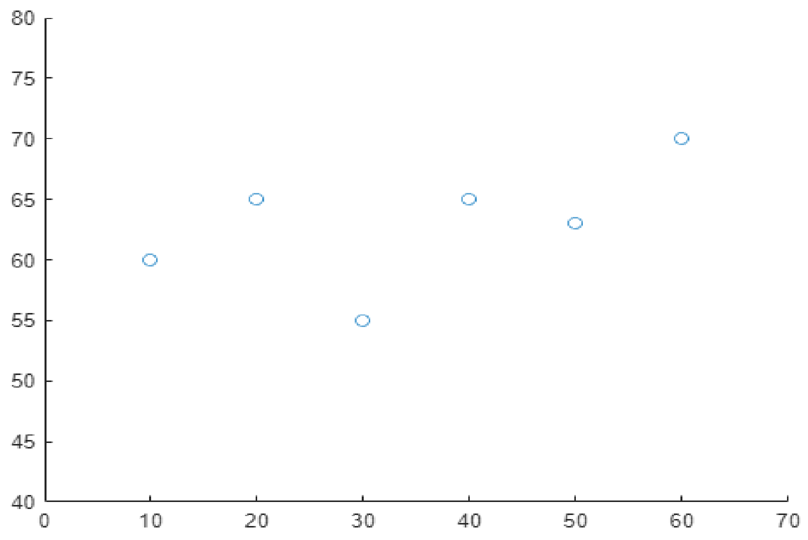
```
b = coef\literals
```

```
b = 3×1
     1
    -1
     2
```

Linear Regression

Suppose that we have collected some measurements in the form of $(x_1, y_1), (x_2, y_2), (x_3, y_3) \ldots (x_m, y_m)$ These measurements could be coming of an engineering application like measuring the speed of a car every 10 seconds. By observing the scatter plot of the discrete measurements, we might notice that their shape can be approximated by a linear equation. We want to fit a straight line to this set of paired measurements.

```
figure
x = 10:10:60;
y = [60, 65, 55, 65, 63, 70];
scatter(x,y)
axis ([0 70 40 80])
```

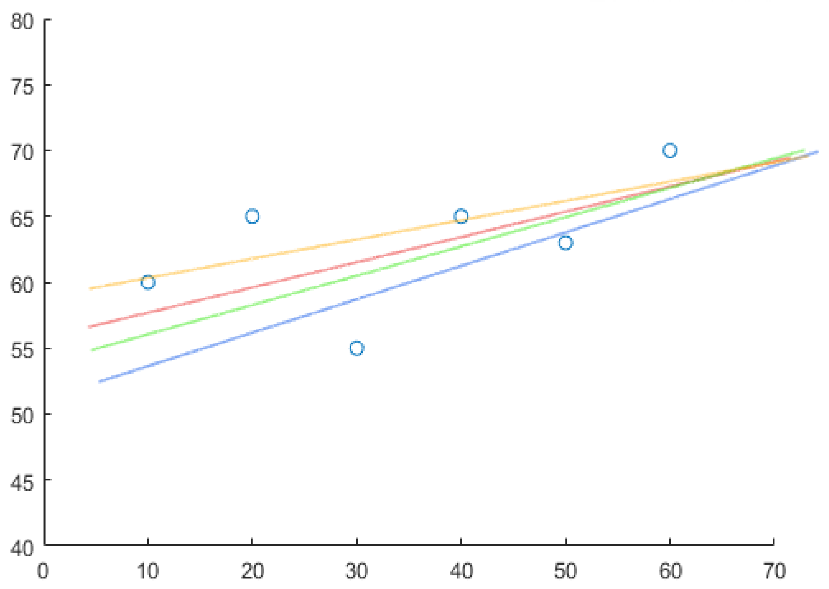


We know that the equation of a straight line is:

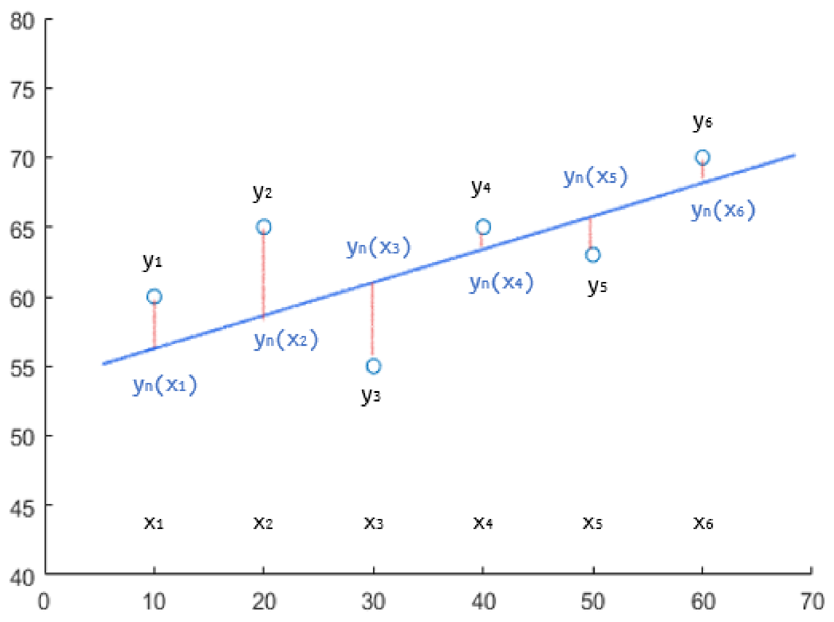
$$y_n(x) = a_1 x + a_0$$

where a_1 is the slope of the line and a_0 is the x-intercept. Yet, even if we find this straight line that fits the data, we know for sure that it will **not** pass through all the data points; some will fall below the line, others above it. Therefore, the straight-line equation will have some errors.

We can also find multiple straight lines that will fit and approximate the data; we can draw any of the four coloured lines and say it approximates the data. So, which one to use?



Notice that for the data pairs $(x_1, y_1), (x_2, y_2), (x_3, y_3) \ldots (x_m, y_m)$, we are going to create a line whose equation is $y_n(x) = a_1 x + a_0$. So, for every data point x, we have its true value y and its corresponding value $y_n(x)$ on the straight line.



The difference between the true value y and its corresponding value on the straight line y_n is what we call the residual error which we express as $y - y_n = e$ (Notice the red lines in the above figure which illustrate this error). Ideally, we want to find a line whose values have the least residual error (difference) from ***all*** corresponding true values. That is, we need to minimize the sum of all absolute errors $\sum_{i=1}^{n} |e_i|$. At the same time, we don't need one or few outlier points to affect the line. For example, we don't want the slightly distant point at (30, 40) to severely shift the straight line downwards which we can arguably agree that it beautifully passes through the other points.

Least-Squares Fit of a Straight Line

One major algorithm used to fit a straight line to measurements is called the least squares fit or least squares errors. The algorithm provides one unique line and given its name, also the least error. The algorithm attempts to minimize the ***S***um of the ***S***quare of the ***R***esidual errors (**SSR**), also called ***S***um of the ***S***quared ***E***stimates of errors (**SSE**):

$$SSR = SSE = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - a_1 x_i - a_0)^2 \qquad (2)$$

Analysing Eq.2, the measurements collected provide us with x_i, y_i, yet we need to have the coefficients $a_0,\ a_1$ which define the line which constitute the two unknowns we need to solve for. We need to have two equations to solve for $a_0,\ a_1$. The derivation starts with differentiating the equation twice, once with respect to a_0 and another with respect to a_1:

$$\frac{\partial SSE}{\partial a_0} = -2 \sum (y_i - a_0 - a_1 x_i) \qquad (3)$$

$$\frac{\partial SSE}{\partial a_1} = -2 \sum (x_i(y_i - a_0 - a_1 x_i)) \qquad (4)$$

We already know that the minimum occurs when the derivative is zero, so we set both $\frac{\partial SSE}{\partial a_0}, \frac{\partial SSE}{\partial a_1}$ to zero, then we collect the terms. We end up with a system of two linear equations with two unknowns that we can easily solve.

$$a_0 = \frac{\sum y_i}{n} - a_1 \frac{\sum x_i}{n} = \bar{y} - a_1 \bar{x} \qquad (5)$$

$$a_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2} \qquad (6)$$

If we take a closer look at Eq. 6, we observe that a_1 depends solely on the measurement points; all terms in the equation are related to $x_i,\ y_i$, and the number of observations n. Once we have the value of a_1, then we can substitute it in Eq. 5 and solve for a_0, thus having the coefficients of the straight line that best fits the data.

Let us apply this equation to our first example where we had measurements of the car speed every 10 seconds. The points we have are (10, 60), (20, 65), (30, 55), (40, 65), (50, 63), (60, 70) where x_i denotes the sample time every 10 seconds, and y_i denotes its speed in km/h. The best approach to solve this by hand is to construct a table of these samples as shown below where we use it to compute all the terms in Eq. 6:

n	x_i	y_i	$x_i y_i$	x_i^2
1	10	60	600	100
2	20	65	1300	400
3	30	55	1650	900
4	40	65	2600	1600
5	50	63	3150	2500
6	60	70	4200	3600
Sum	**210**	**378**	**13500**	**9100**
Mean	**35**	**63**		

$$a_1 = \frac{6 \times 13500 - 210 \times 378}{6 \times 9100 - 210^2} = 0.1543$$

Solving for a_0:

$$a_0 = \frac{378}{6} - 0.1543 \times \frac{210}{6} = 57.5995$$

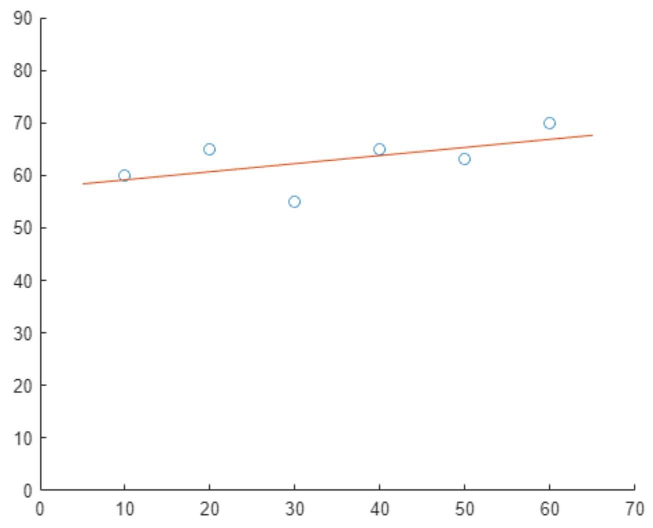
Therefore, the line which best fits the data is the one with the least SSR (SSE) compared to any other line and is represented by:

$$y = 0.1543x + 57.5995$$

We call the line resulting from this linear regression method **the regression line.**

We shall now plot this line with the scattered measurements on one plot:

```
figure
x = 10:10:60;
y = [60, 65, 55, 65, 63, 70];
scatter(x,y)
axis ([0 70 0 90])
hold on
xs = 5:0.1:65;
ys = 0.1543.*xs+57.5995;
plot(xs,ys)
```



Notice that the slightly far measurement at point (30, 55) did not shift the line towards it as much.

Quantifying the Goodness of Fit

What we learnt thus far is that the least squares method gives us the best fit it can given the measurements observed. Yet, how can we determine that best fit straight line the algorithm was able to provide is actually good enough? We need more criteria to tell us if this fit is good or not!

In Eq. 2, we defined the sum of the square of the residual errors $SSR = SSE = \sum_{i=1}^{n}(y_i - a_1x_i - a_0)^2$ as the square of the error between each measured point and each predicted point on the regression line. The problem with this measure is that with more points available, the more errors and SSR (SSE) keeps getting larger. However, if we divide this value by the number of points ***n***, then this is called *mean square error* or **MSE:**

$$MSE = \frac{SSR}{n} = \frac{SSE}{n} \tag{7}$$

There is also another goodness of fit metric called ***R***oot ***M***ean ***S***quare ***E***rror (**RMSE**) which is simply taking the square root of **MSE:**

$$RMSE = \sqrt{MSE} \tag{8}$$

For all three metrics, **SSR** (**SSE**), **MSE**, or **RMSE**, the closer the value to zero, means the less the errors, and therefore it is a better fit. A perfect fit will have all these value compute to zero.

We also have another way to quantify the goodness of the fit which is called the ***correlation coefficient****,* or simply ***r****.* It can be computed using the following formula:

$$r = \frac{n\sum(x_iy_i) - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}} \tag{9}$$

From which we can compute the R-Square metric r^2. For a perfect fit, $r^2 = 1$, and the closer the values we obtain to one means the better the fit.

To apply these metrics to the car example, we already found that the regression line equation is $y = 0.1543x + 57.5995$, if we substitute the values of x_i that we collected in the regression line formula, we will get:

```
p = [0.1543, 57.5995];
yn = polyval(p, 10:10:60)
```

```
yn = 1×6
   59.1425   60.6855   62.2285   63.7715   65.3145   66.8575
```

Compare these values to the actual measured car speed values:

```
disp(y)
```

```
    60    65    55    65    63    70
```

To calculate $SSR \ (SSE)$:

```
SSR = sum((yn - y).^2)
```

```
SSR = 88.3429
```

We calculate the *correlation coefficient **r***:

```
n = 6;
r = (n *sum(x.*y) - sum(x)*sum(y))/ (sqrt(n*sum(x.^2)-
sum(x)^2)*sqrt(n*sum(y.^2)-sum(y)^2))
```

```
r = 0.5661
```

```
disp(r^2)
```

```
    0.3204
```

It is worth to note that to find if the fit has high quality or not, we should not depend on the R-Square alone, or the MSE alone, because sometimes R-Square could be close to one, yet MSE is way high than zero. So, we should always consider both metrics.

MATLAB Built-in Functions for Regression

In this course, we only introduce linear regression. That is how to best fit linear lines. There are many numerical methods which attempt to fit non-linear lines (*e.g.,* quadratic, cubic, log, ln, *etc.*). MATLAB offers the command **`polyfit.`** This command can actually fit higher degree polynomials and not only linear regression. It is internally built on the concept of least square errors.

To fit and plot a linear regression line in our previous example, we can use:

```
x = 10:10:60;
y = [60, 65, 55, 65, 63, 70];
p = polyfit(x, y, 1)   % The 1 here denotes we need to fit a line (linear
                         regression)
```

```
p = 1×2
    0.1543   57.6000
```

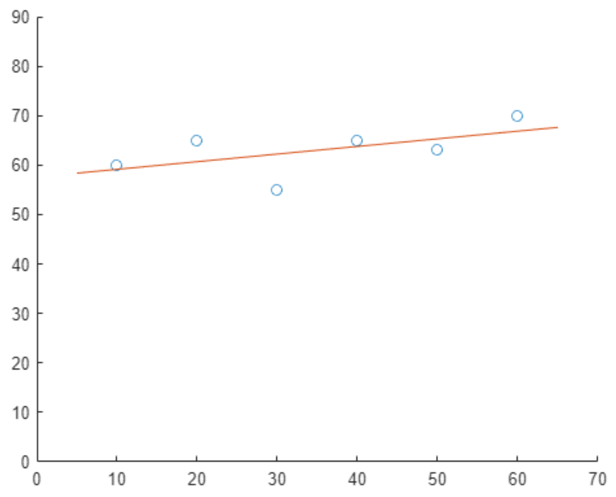
Notice how the result almost matches our previous results.

We can plot the measurements and linear regression lines using **polyfit** and **polyval** as follows:

```
figure
x = 10:10:60;
y = [60, 65, 55, 65, 63, 70];
scatter(x,y)
axis ([0 70 0 90])
hold on
p = polyfit(x, y, 1)
```

```
p = 1×2
    0.1543   57.6000
```

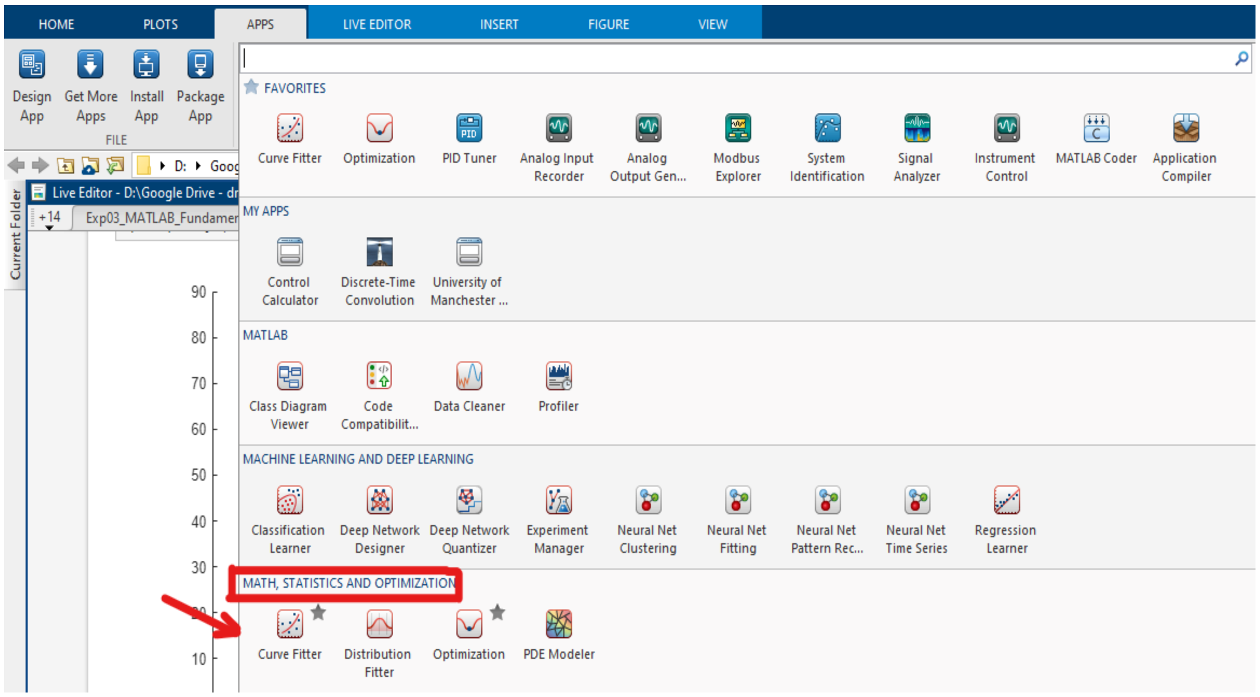
```
xs = 5:0.1:65;
ys = polyval(p, xs);
plot(xs, ys)
```



Curve Fitting

In the previous section, we learnt to use linear regression to find the equation of the linear line that fits the data with the least error possible. However, this technique only works when the data plot resembles a linear plot. What if the points are similar to a quadratic equation? an exponential equation? That is, they are non-linear. For sure linear regression will fail. There are many numerical techniques for non-linear regression but in this section, we will use MATLAB's curve fitting toolbox to find the equation that best fits the data points we have instead of doing numerical methods.

In MATLAB, go to the `Apps` tab, and under `Maths, Statistics, and Optimization` you will find a toolbox called `Curve Fitting`. Open this toolbox by clicking on it.

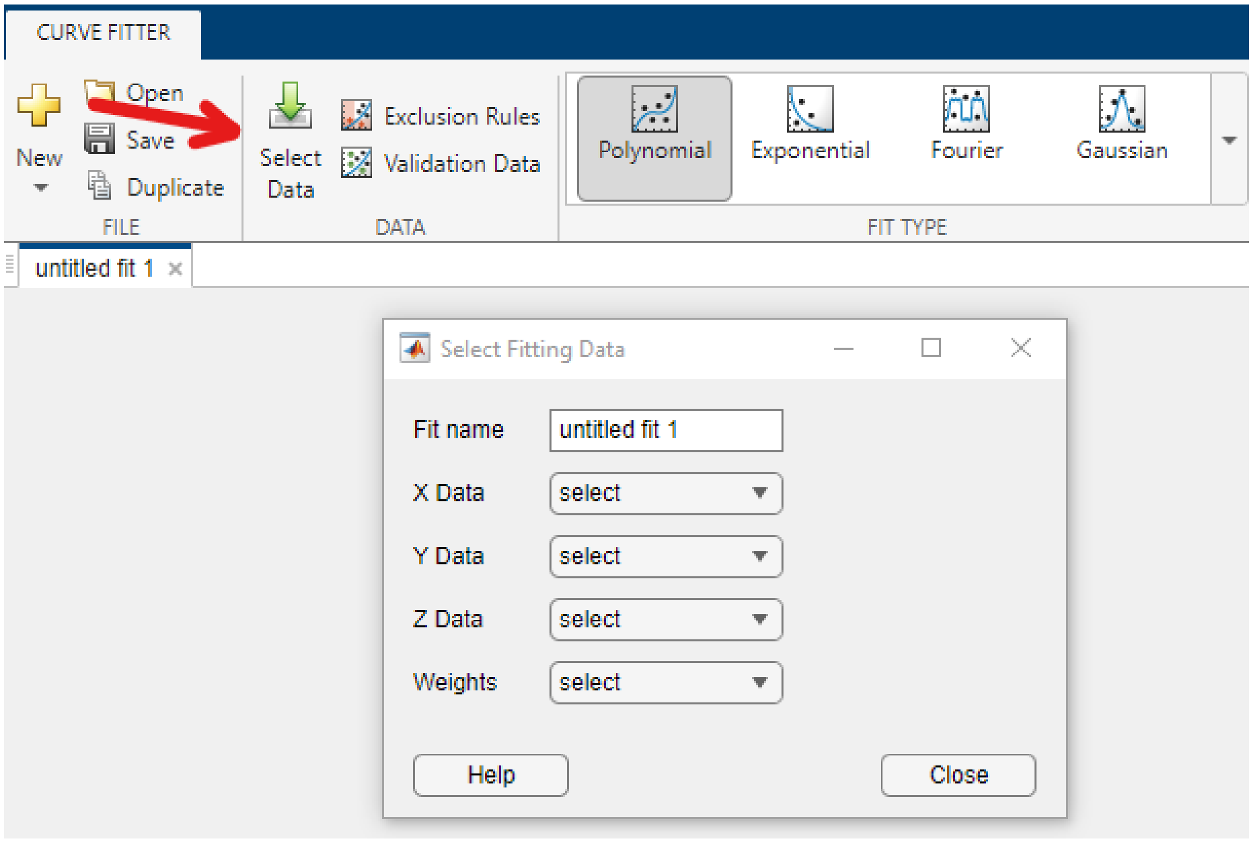


The tool is very simple to use. First, we need to select the data points for the x-axis, the y-axis (or the z-axis if the plot is 3D). Make sure that the length of the data points for all axes is equal.

Let us use the previous data set of the car example:

```
x = 10:10:60;                    % original measurments
y = [60, 65, 55, 65, 63, 70];
```

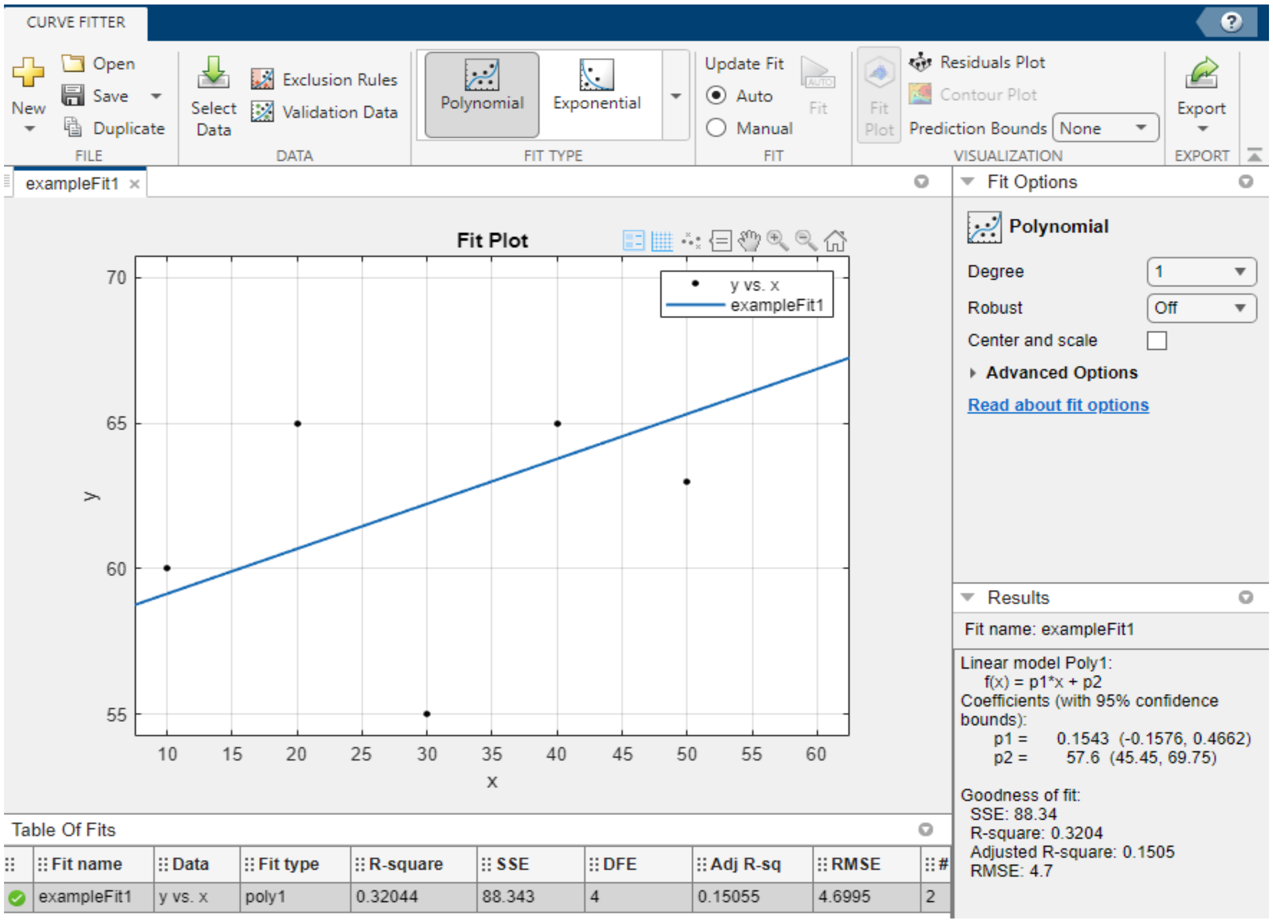
Then, from curve fitter window, click on **Select Data**, and from the new window, select the variable x for the **X Data**, and the variable Y for the **Y Data**, then close the window. You can also give your fit a name, say **exampleFit1**



Fit name: untitled fit 1
X Data: select
Y Data: select
Z Data: select
Weights: select

Notice that by default, the polynomial option was selected, and that the polynomial degree was set to 1, that is a linear line. So, the toolbox starts with linear regression.

Notice the results and the goodness of fits measures:



Degree: 1
Robust: Off
Center and scale: ☐

Fit name: exampleFit1

```
Linear model Poly1:
    f(x) = p1*x + p2
Coefficients (with 95% confidence
bounds):
       p1 =      0.1543  (-0.1576, 0.4662)
       p2 =        57.6  (45.45, 69.75)

Goodness of fit:
  SSE: 88.34
  R-square: 0.3204
  Adjusted R-square: 0.1505
  RMSE: 4.7
```

Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓ exampleFit1	y vs. x	poly1	0.32044	88.343	4	0.15055	4.6995	2

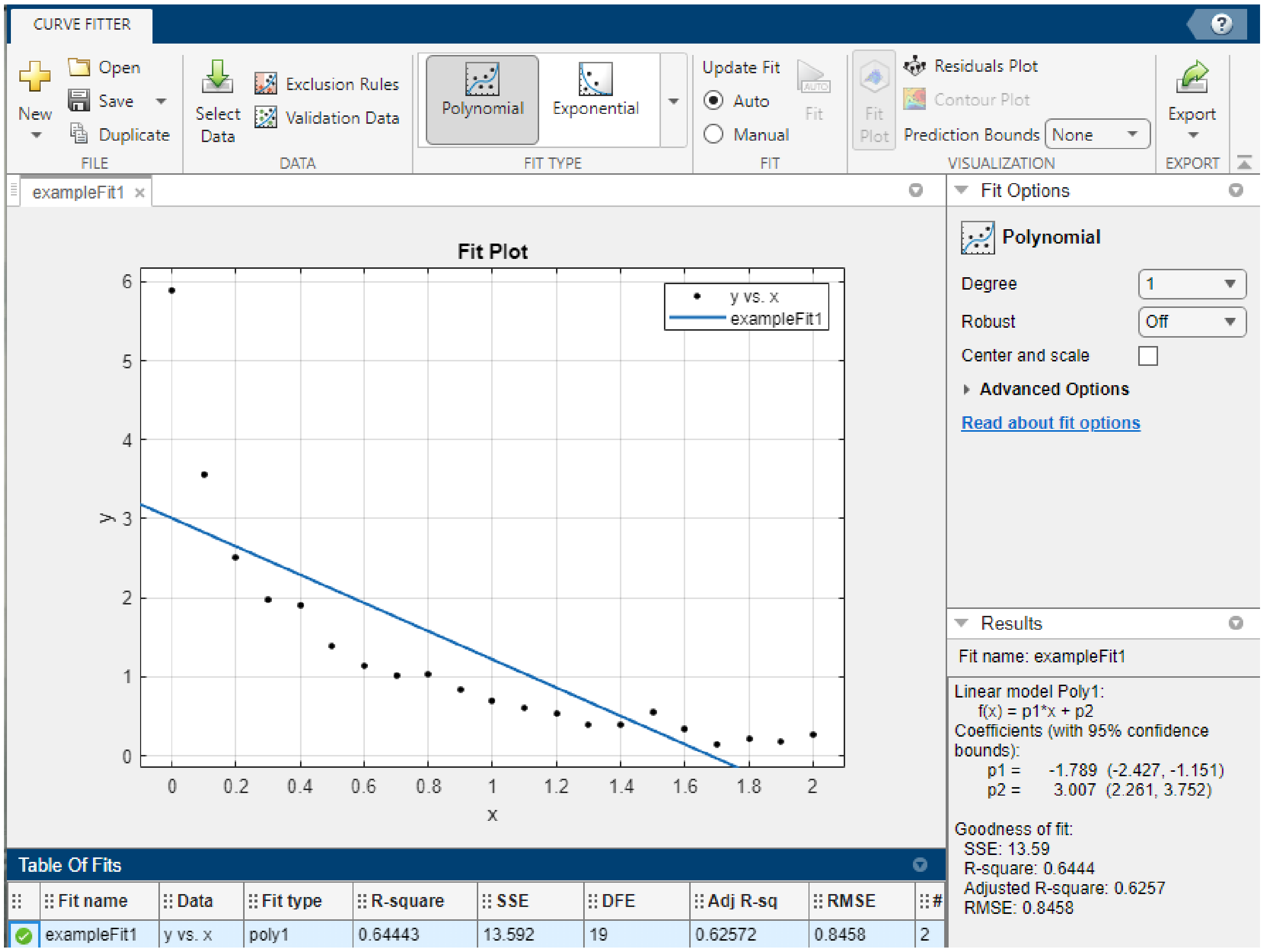
In this toolbox, **p1** and **p2** are the same as **a1** and **a0** that we calculated numerically. Also, notice the values for **SSE**, **RMSE**, and **R-square.**

Now, let us try a set of non-linear points, for example the **x** and **y** pairs in the matrix Data:

```
Data = ...
  [0.0000    5.8955
   0.1000    3.5639
   0.2000    2.5173
   0.3000    1.9790
   0.4000    1.8990
   0.5000    1.3938
   0.6000    1.1359
   0.7000    1.0096
   0.8000    1.0343
   0.9000    0.8435
   1.0000    0.6856
   1.1000    0.6100
   1.2000    0.5392
   1.3000    0.3946
   1.4000    0.3903
```

```
    1.5000    0.5474
    1.6000    0.3459
    1.7000    0.1370
    1.8000    0.2211
    1.9000    0.1704
    2.0000    0.2636];

 x = Data(:,1);
 y = Data(:,2);
```

Load the **x** and **y** values into the ***Curve Fitting Select Data*** Window. It is clear that Linear Regression does not fit the data well. This is obvious given that the **RMSE** is much higher than zero and R-Square is not close to 1.



Polynomial

Degree 1

Robust Off

Center and scale ☐

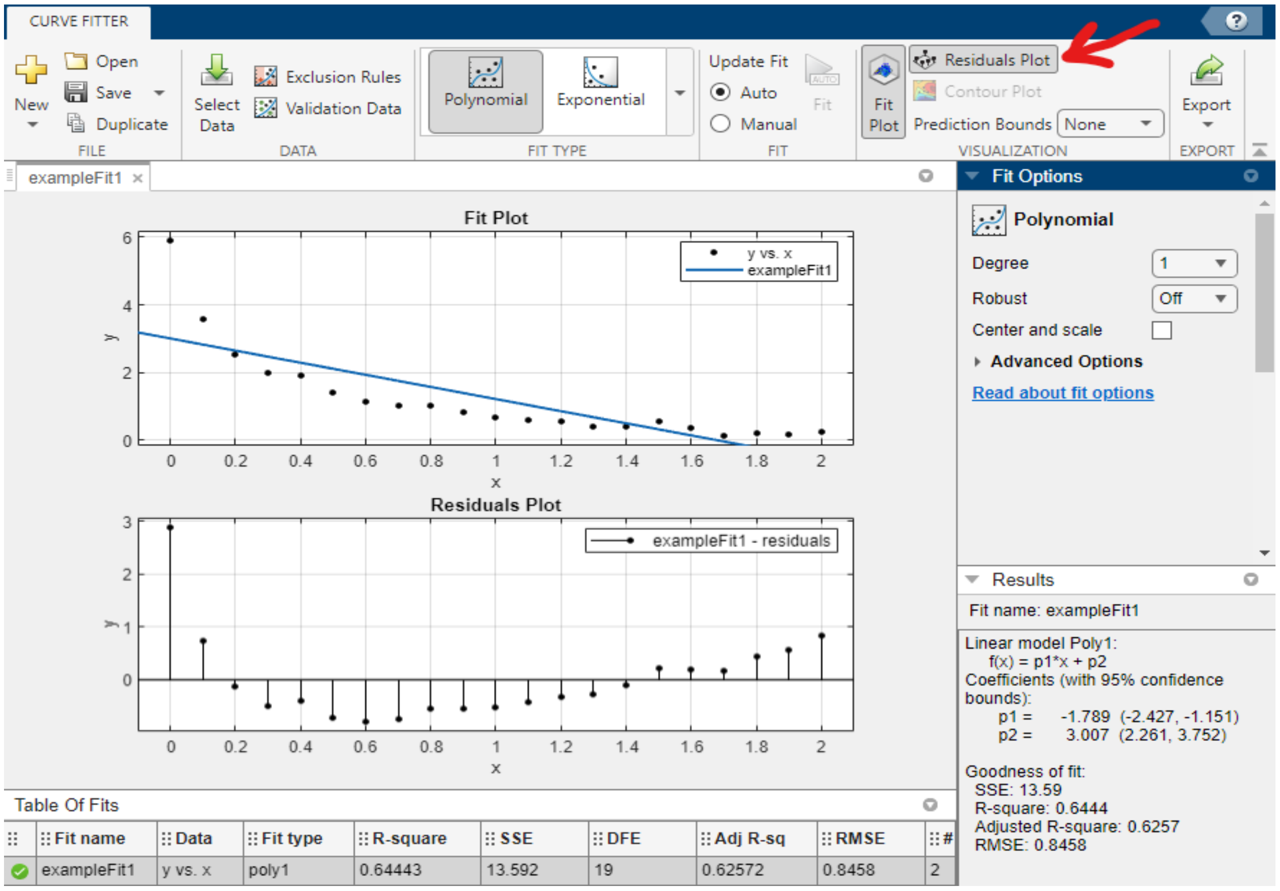
Fit name: exampleFit1

Linear model Poly1:
f(x) = p1*x + p2
Coefficients (with 95% confidence bounds):
p1 = -1.789 (-2.427, -1.151)
p2 = 3.007 (2.261, 3.752)

Goodness of fit:
SSE: 13.59
R-square: 0.6444
Adjusted R-square: 0.6257
RMSE: 0.8458

	Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓	exampleFit1	y vs. x	poly1	0.64443	13.592	19	0.62572	0.8458	2

If you click on the **`Residual Plot`** button, it will plot the difference between each actual point y and the corresponding point on the line y_n:



Degree: 1
Robust: Off
Center and scale: ☐

Fit name: exampleFit1

Linear model Poly1:
f(x) = p1*x + p2
Coefficients (with 95% confidence bounds):
p1 = -1.789 (-2.427, -1.151)
p2 = 3.007 (2.261, 3.752)

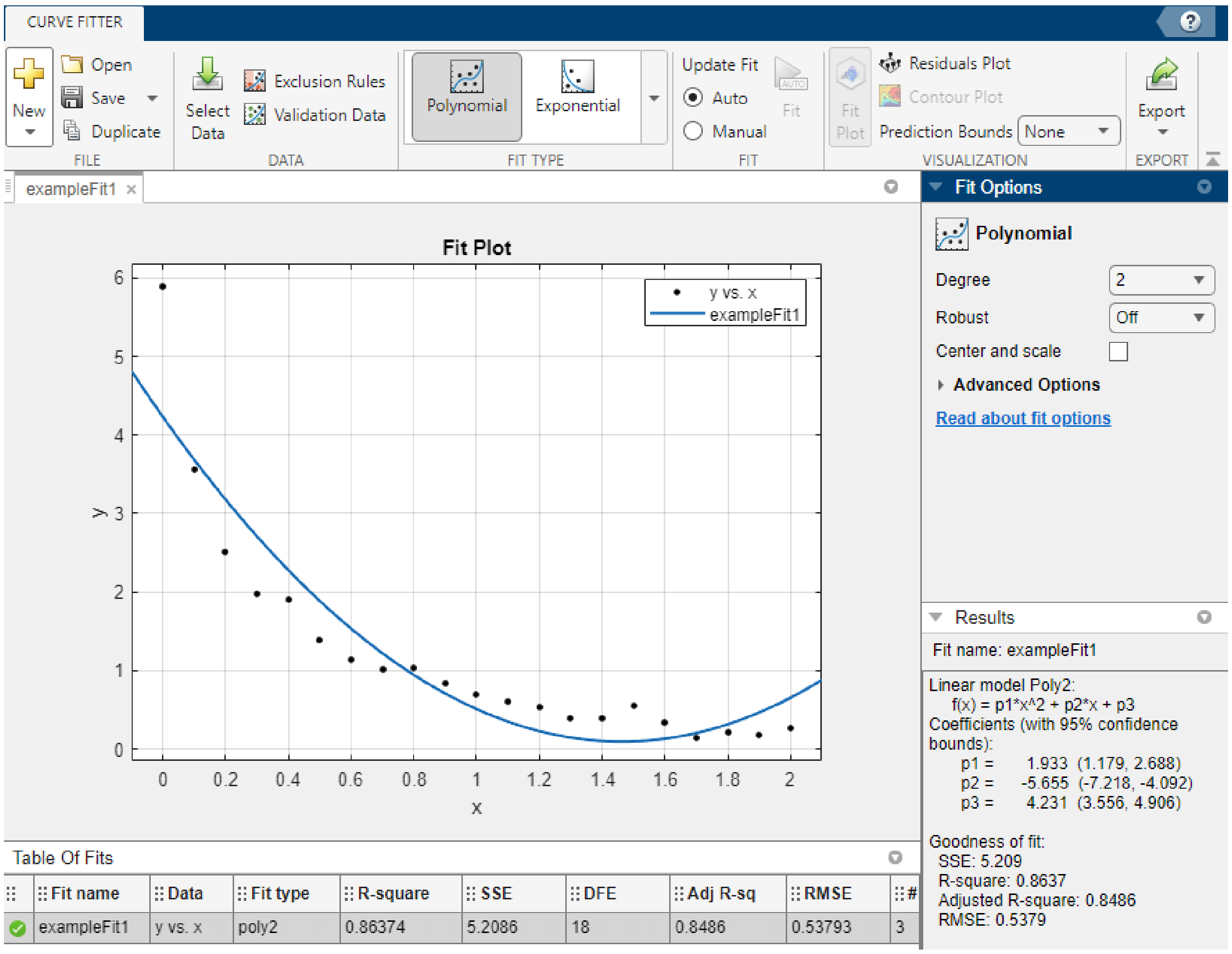
Goodness of fit:
SSE: 13.59
R-square: 0.6444
Adjusted R-square: 0.6257
RMSE: 0.8458

Table Of Fits

	Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓	exampleFit1	y vs. x	poly1	0.64443	13.592	19	0.62572	0.8458	2

Notice how the residual errors are quite large for a bad fit.

Let us try to use a quadratic fit using polynomial of degree 2. Even though R-Square has increased from 0.6443 to 0.8637, and the RMSE decreased from 0.8458 to 0.5379. It is clear we can do better.



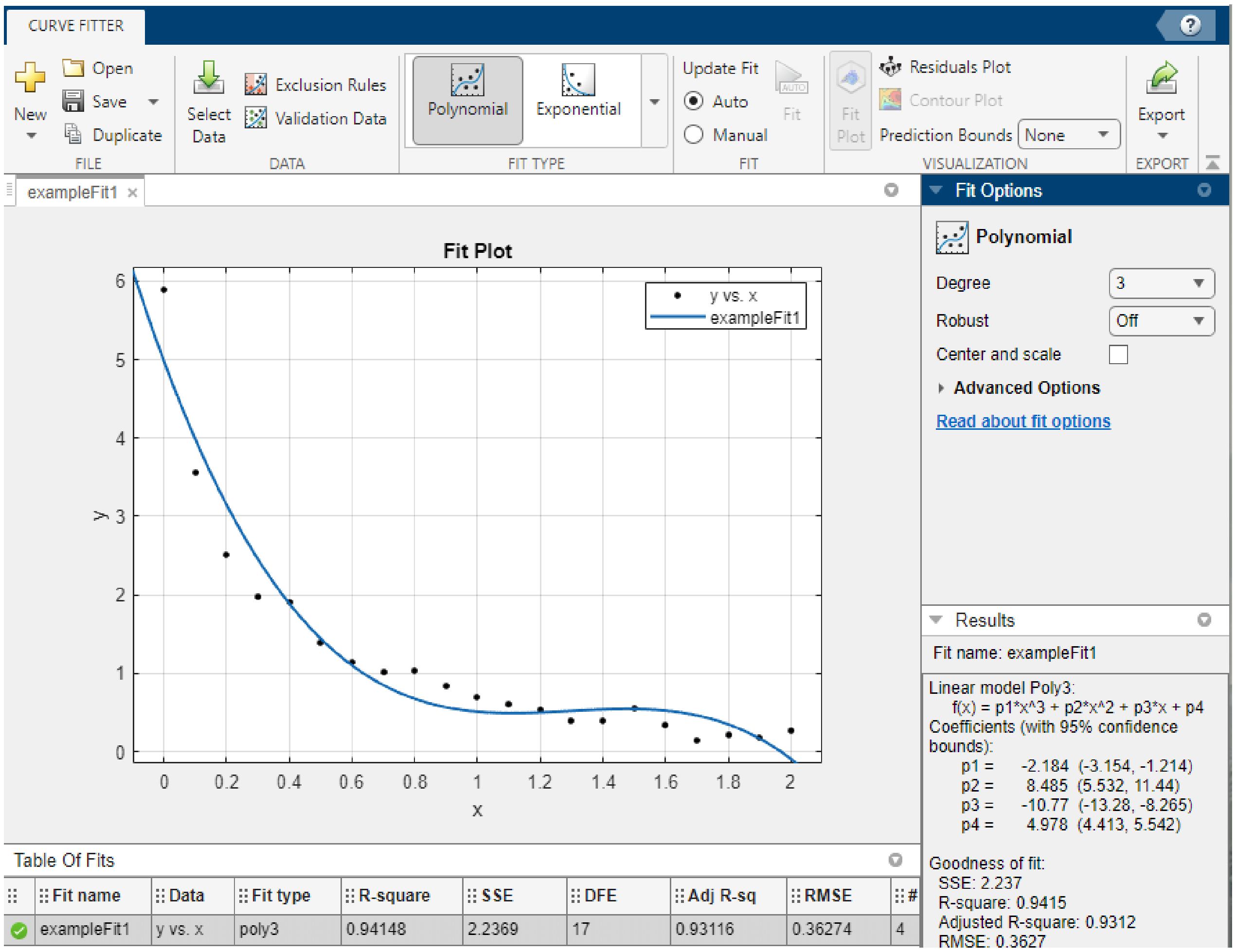
Degree: 2
Robust: Off
Center and scale: ☐

Table Of Fits

Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓ exampleFit1	y vs. x	poly2	0.86374	5.2086	18	0.8486	0.53793	3

```
Fit name: exampleFit1

Linear model Poly2:
    f(x) = p1*x^2 + p2*x + p3
Coefficients (with 95% confidence
bounds):
      p1 =      1.933  (1.179, 2.688)
      p2 =     -5.655  (-7.218, -4.092)
      p3 =      4.231  (3.556, 4.906)

Goodness of fit:
  SSE: 5.209
  R-square: 0.8637
  Adjusted R-square: 0.8486
  RMSE: 0.5379
```

What if we try to use a cubic equation by using a polynomial of degree 3?



Polynomial

Degree: 3
Robust: Off
Center and scale: ☐

Fit name: exampleFit1

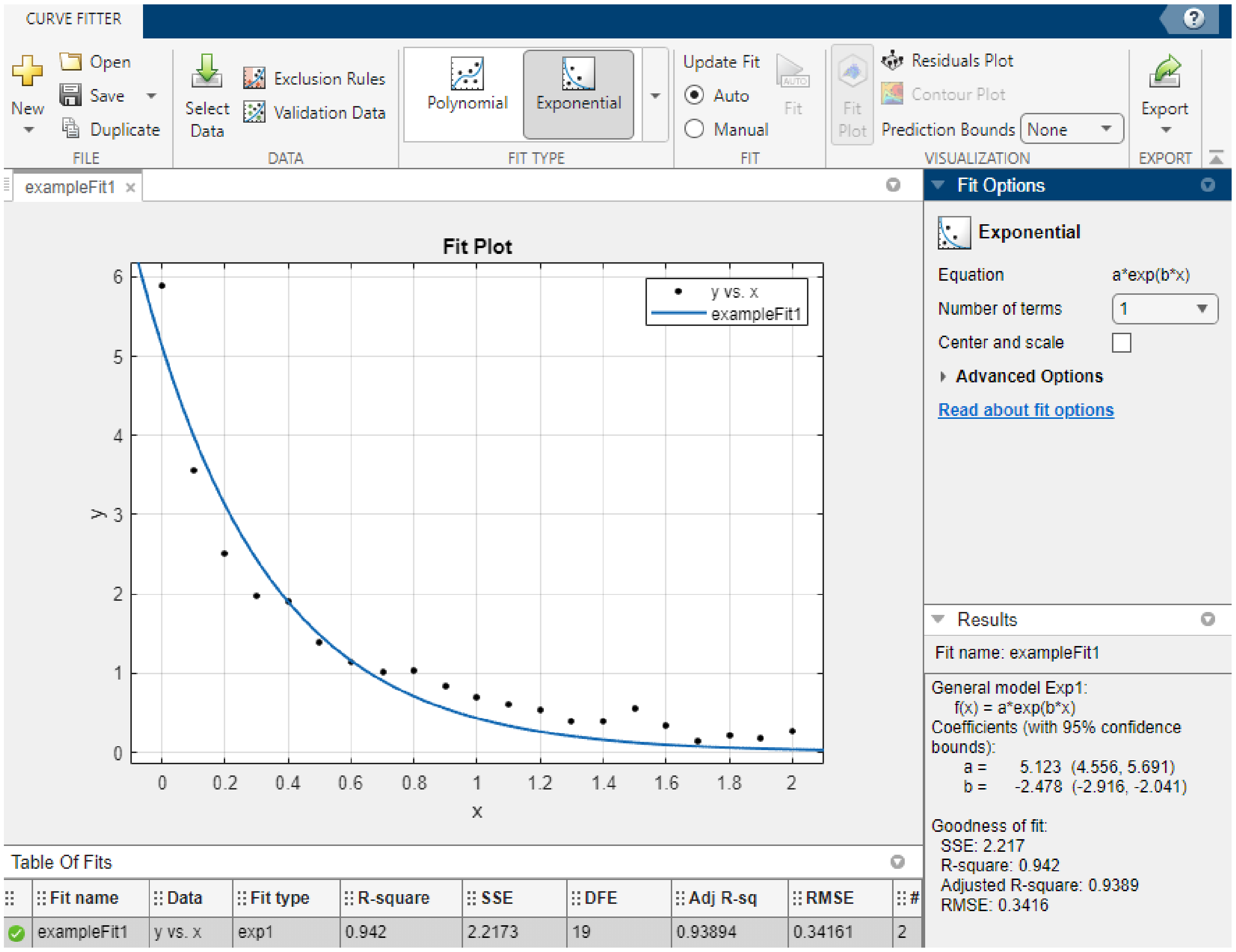
```
Linear model Poly3:
    f(x) = p1*x^3 + p2*x^2 + p3*x + p4
Coefficients (with 95% confidence
bounds):
      p1 =      -2.184  (-3.154, -1.214)
      p2 =       8.485  (5.532, 11.44)
      p3 =      -10.77  (-13.28, -8.265)
      p4 =       4.978  (4.413, 5.542)

Goodness of fit:
  SSE: 2.237
  R-square: 0.9415
  Adjusted R-square: 0.9312
  RMSE: 0.3627
```

Table Of Fits

	Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓	exampleFit1	y vs. x	poly3	0.94148	2.2369	17	0.93116	0.36274	4

Let us change our equations from polynomials to exponentials, and try to change the number of exponential terms from 1 to 2:

When we have one exponential term, we don't have much change in terms of goodness of fit compared to the cubic equation:



Exponential

Equation a*exp(b*x)

Number of terms 1

Center and scale

Advanced Options

Results

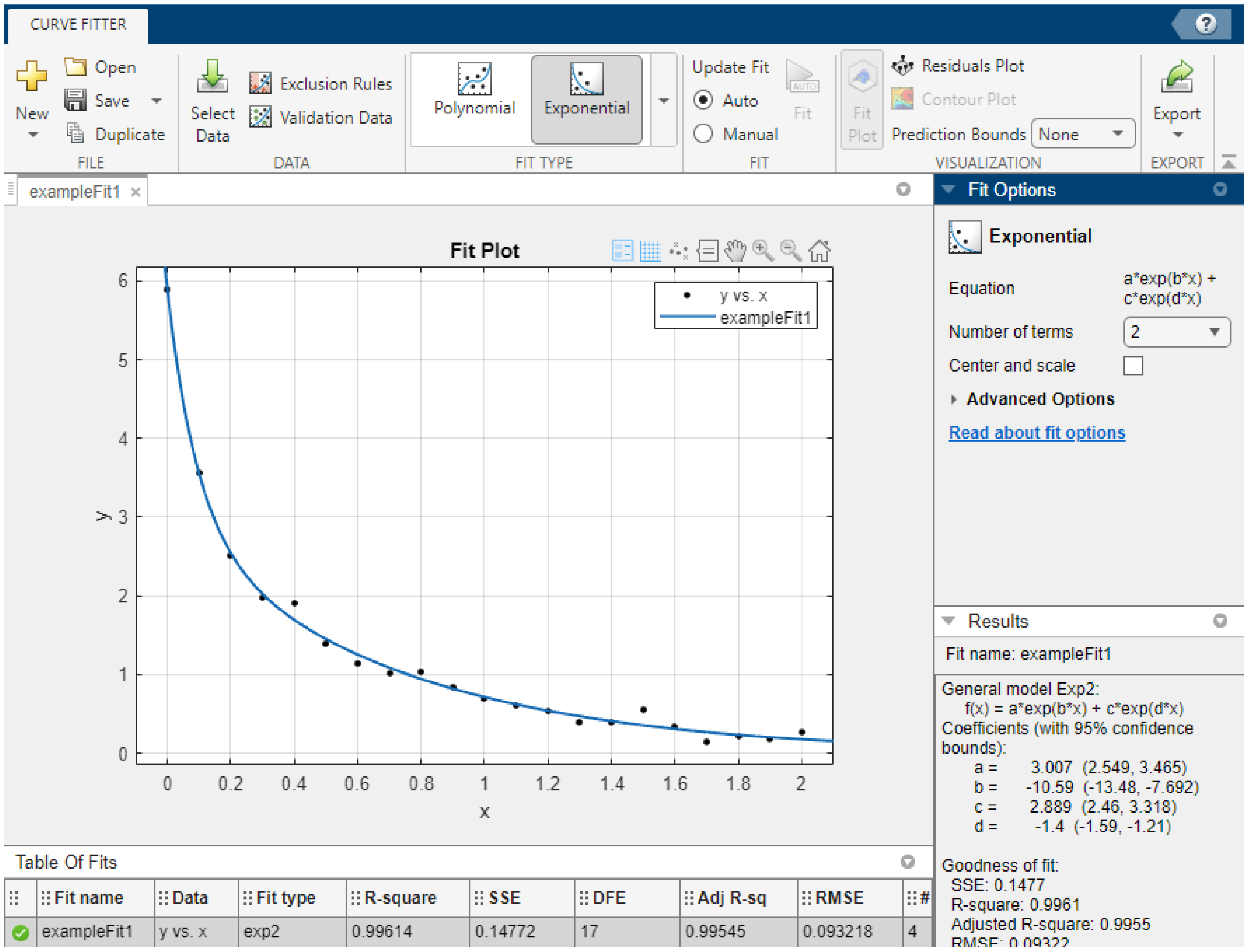
Fit name: exampleFit1

```
General model Exp1:
    f(x) = a*exp(b*x)
Coefficients (with 95% confidence
bounds):
       a =      5.123  (4.556, 5.691)
       b =     -2.478  (-2.916, -2.041)

Goodness of fit:
  SSE: 2.217
  R-square: 0.942
  Adjusted R-square: 0.9389
  RMSE: 0.3416
```

Table Of Fits

	Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓	exampleFit1	y vs. x	exp1	0.942	2.2173	19	0.93894	0.34161	2

However, if we use two exponential terms, notice how beautifully the curve fits the data. Also notice that R-Square is 0.9961 and very close to 1, while RMSE is 0.09322 and much closer to zero than before.



Update Fit: [✓ Auto] Manual

Prediction Bounds: None

Exponential

Equation: a*exp(b*x) + c*exp(d*x)

Number of terms: 2

Center and scale: ☐

Fit name: exampleFit1

```
General model Exp2:
    f(x) = a*exp(b*x) + c*exp(d*x)
Coefficients (with 95% confidence
bounds):
      a =      3.007  (2.549, 3.465)
      b =     -10.59  (-13.48, -7.692)
      c =      2.889  (2.46, 3.318)
      d =       -1.4  (-1.59, -1.21)

Goodness of fit:
  SSE: 0.1477
  R-square: 0.9961
  Adjusted R-square: 0.9955
  RMSE: 0.09322
```

	Fit name	Data	Fit type	R-square	SSE	DFE	Adj R-sq	RMSE	#
✓	exampleFit1	y vs. x	exp2	0.99614	0.14772	17	0.99545	0.093218	4

Of course, we can even tune the fit for better results by enabling the advanced options and selecting more parameters like certain algorithms among others. But this is out of scope of this lab course.

Now, in the previous fit, notice that the results returned four parameters $a, b, c,$ and d. Also note that the equation is given as a*exp(b*x) + c*exp(d*x), so we can write this in MATLAB:

```
f = @(x) 3.007*exp(-10.59*x)+ 2.889*exp(-1.4*x)
```

```
f = function_handle with value:
    @(x)3.007*exp(-10.59*x)+2.889*exp(-1.4*x)
```

and we can find any value on this curve by simply calling the function, for example, to find $f(1.75)$, write:

```
f(1.75)
```

```
ans = 0.2493
```

Interpolation

In engineering and scientific applications, we collect measurements from sensors or other experiments. These measurements are discrete in nature; that is, they are sampled at non-continuous points in time (*e.g.,* every 10 ms, second, day, *etc.*). Sometimes, we might have an erroneous measurement (possibly due to high noise) or a missing measurement (*e.g.*sensor failure). As such, we want to predict what the original value was and replace the erroneous or missing value. At other times, we might be interested in predicting the value for a point of time that we did not take a measurement for.

Suppose we are measuring the speed of a car every ten seconds for the duration of one minuet similar to the car example we have seen already. What if we wanted to predict the speed of the car at the 55th second? Or the 43rd second? These are values that we did not take a measurement for.

You could use the regression techniques we just learnt to come up with the regression line (polynomial or otherwise) to find a formula for the speed, then apply $f(43)$:

```
x = 10:10:60;                 % original measurments
y = [60, 65, 55, 65, 63, 70];

p = polyfit(x, y, 1); % finding the regression line 1st-degree polynomial
ys = polyval(p, 43)   % Evaluate the desired point using this polynomial
```

```
ys = 64.2343
```

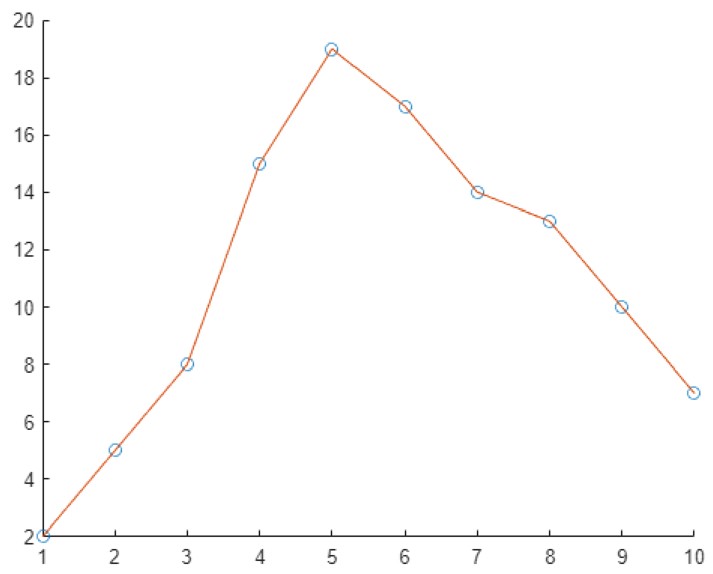
In the above example, we applied linear regression because we noticed through the scatter plot that a straight line better fits the data.

But what if you had measurements described as this:

```
figure
x2 = 1:1:10;
y2 = [2, 5, 8, 15, 19, 17, 14, 13, 10, 7];
scatter(x2,y2);
```

And you want to predict the value at 3.25? In this case, you might want to connect straight lines between the points $f(3)$ and $f(4)$, and compute the slope of this line segment, then write the equation of the line, then substitute $f(3.5)$ in the line equation.

```
hold on
plot(x2,y2)
```



Notice that we were not able to use linear regression on the entire points to predict the value because the scatter as a whole does not represent a linear function. Instead, we took two adjacent points (3, 8) and (4, 15) and connected them with a line and used the line equation to find the value at x = 3.5.

In a similar fashion, MATLAB offers the function **`interp1`** that interpolates data at certain data points. To predict the value of 3.5, we need to only pass the entire original measurements, and the data point we want to interpolate at:

```
interp1(x2, y2, 3.5)
```

```
ans = 11.5000
```

You can also interpolate at many points at once by passing a vector of points:

```
interp1(x2, y2, [3.5, 6.75, 8.25])
```

```
ans = 1×3
    11.5000    14.7500    12.2500
```

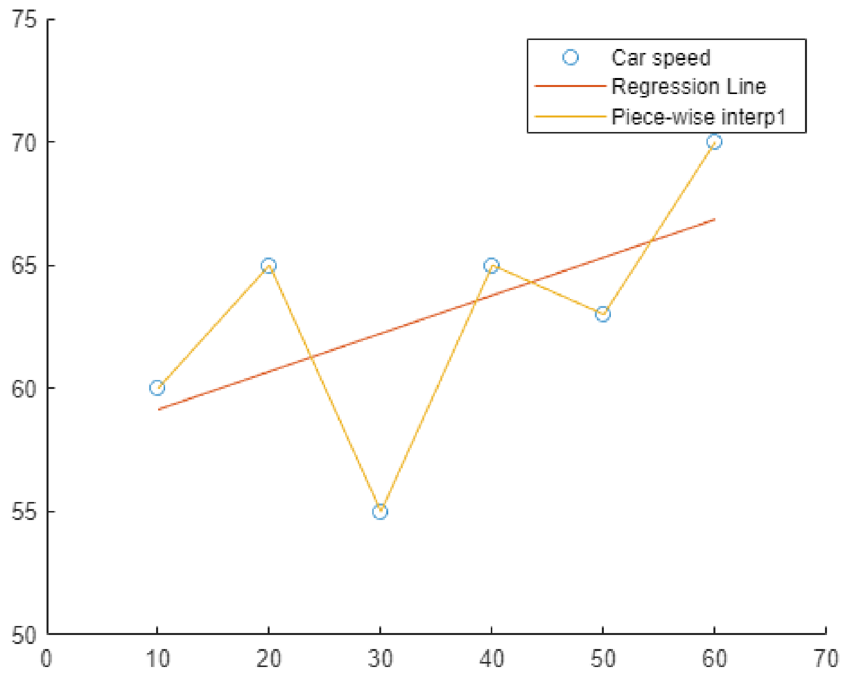
Let us compare the output of the **`interp1`** function for the car example with the output we got using **`polyfit`** and **`polyval`**:

```
interp1(x,y, 43)
```

```
ans = 64.4000
```

Why are the two values different? That is, the interpolated result using the regression line was 64.2343 and using **`interp1`** is 64.4. Let us examine the plot to illustrate how they differ. The interp1 command connects each successive two points with a line and uses the equation of that line piece to find the interpolation. The regression line is a line that approximates all points together and thus has a different equation.

```
figure
x = 10:10:60;                  % original measurements
y = [60, 65, 55, 65, 63, 70];
scatter (x,y)
hold on
p = polyfit(x, y, 1);          % finding the regression line 1st-degree polynomial
yn = polyval (p, 10:0.1:60);
plot(10:0.1:60, yn)
plot (x,y)
axis ([0 70 50 75])
legend('Car speed', 'Regression Line', 'Piece-wise interp1')
```



In either of the two previous cases, when we examine the previous figure, we can easily see that connecting the points using straight line or regression line does not best fit the function or might not offer the best interpolated value. We could have used a smoother fit which will capture the actual figure more accurately. This will then yield better predictions and interpolations.

MATLAB provides the command **`spline`** which performs cubic-spline interpolation instead of linear interpolation. It has the same syntax as **`interp1`** :

```
ys  = spline(x2, y2, 3.5)
```

```
ys = 11.3279
```

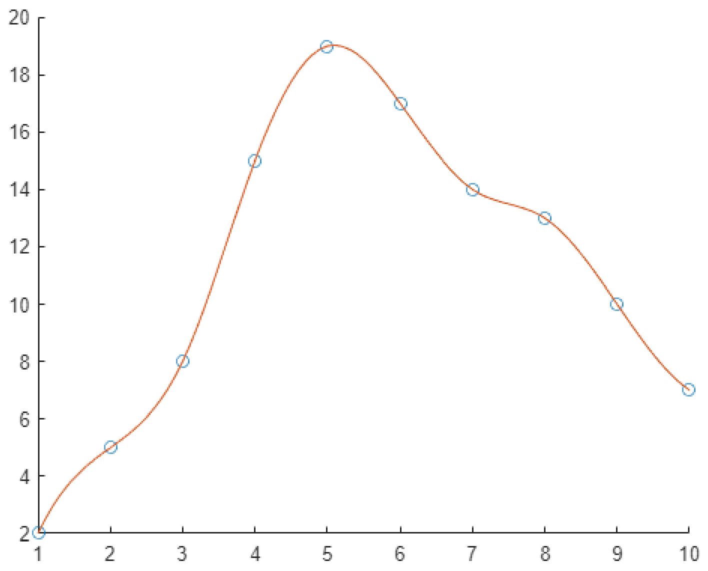
You can also interpolate at many points at once by passing a vector of points:

```
spline(x2, y2, [3.5, 6.75, 8.25])
```

```
ans = 1×3
    11.3279    14.5459    12.4709
```

To visualize how **`spline`** works, we can plot the smoothed curve:

```
figure
x2 = 1:1:10;
y2 = [2, 5, 8, 15, 19, 17, 14, 13, 10, 7];
scatter(x2,y2);
hold on
xnew = 1:0.1:10;
ynew = spline(x2, y2, xnew);
plot(xnew, ynew)
```



Experiment version 1.1
Original Experiment December 17th, 2020
Last Updated April 8th, 2022
Dr. Ashraf Suyyagh - All Rights Reserved

Revision History

Ver. 1.1

- Corrected the notation of the linear equation and the linear system and made it easier to understand.
- Replaced the plots and figures of the linear regression section with new ones and simplified the discussion
- Removed some of the previous metrics of the goodness of fit and introduced simpler ones. Removed the difficult interpretation of the some of these statistical metrics.
- Added a new section on the curve fitting toolbox to cover non-linear regression and other algorithms in simple manner.
- Added a clarification on why interpolation using the regression line and using **interp1** function can be different.
- Removed **interp2**, **interp3**, and **intern** commands.

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 7 - Error Analysis and Optimisation

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Error Analysis

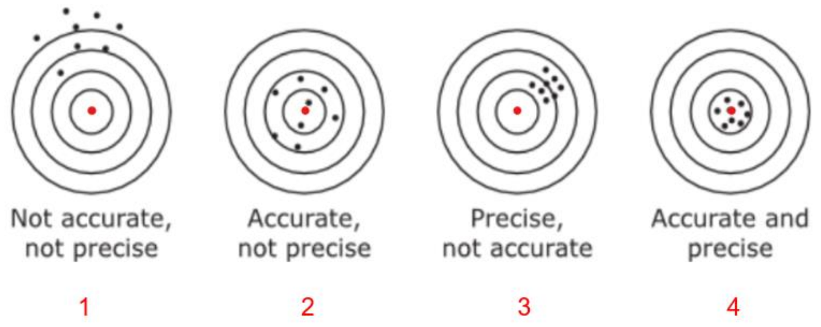
In numerical analysis, we learn the basics of how computers solve mathematical problems such as finding roots of functions, finding minima and maxima, how they integrate and differentiate functions. Numerical analysis is the basis upon which the math and specialized libraries in programming languages are implemented. In fact, the MATLAB commands we will introduce are written using numerical techniques.

In math courses, we learn about exact solutions for many problems that yield an exact answer. However, numerical analysis is based on approximating solutions to engineering and scientific problems. We get very close to the answer but not necessarily the exact answer. Consequently, with approximations there are inherent errors, and these errors must be well understood, and if possible reduced. In the first part of this lab, we will present the major error analysis concepts.

Difference between Accuracy and Precision

Engineers and scientists often deal with errors related to either calculations or measurements. These errors can be characterized with regard to their accuracy and precision. *Accuracy* refers to how closely a computed or measured value agrees with the true value, *i.e.,* the actual value. *Precision* refers to how closely individually computed or measured values agree with each other. The following figure illustrates the concepts of accuracy and precision by using a marksman target board. Suppose that the true value is the centre of this target board (in red):

1. In the first case, the collected values are neither precise (they are sparse) nor true (away from target).
2. In the second case, the values are closer to the true target (accurate), yet not in agreement with each other (not precise).
3. In the third case, we can see that the values are precise, as they agree collectively with each other, however, they are not necessarily on target.
4. In the fourth case, we can see that the values are precise, as they agree collectively with each other and also very close to the true value.



You can always plot the collected measurements and infer if they are precise and/or accurate. Given that most of our measurements are discrete in nature, it would make sense to use a discrete plot such as **`scatter`**.

Roundoff Errors

Numerical roundoff errors arise because digital computers cannot represent floating-point numbers accurately. This is due to the standards used to represent floating point numbers (IEEE 754 standard) and the hardware circuits that implement these standards. This is why in some languages we have single-precision type (32-bits: float) or double-precision type (64-bits: double) where doubles have a wider range and are more accurate than floats. MATLAB by default uses the double-precision format. However, even this format can result in roundoff errors, for example:

```
1000.43 - 1000
ans =
   0.429999999999950
```

or

```
0.7642 - 0.7641
ans =
     9.999999999998899e-05
```

The previous two error cases are called *subtractive cancellation* that results when we subtract two close numbers. Even though the roundoff error can be negligible, yet its cumulative error can result in erroneous results. For example, we know that summing the small number 0.0001 ten thousand times will result in the value one:

```
s = 0;
for i = 1:10000
    s = s + 0.0001;
end
disp(s)
  0.999999999999906
```

However, we notice that the result is clearly imprecise. This is because the number 0.0001 cannot be precisely expressed in binary format. There is no solution to roundoff errors except by a change in the design of a new standards and new computer circuits that have more bits to represent numbers more precisely.

Absolute Errors

In mathematics, **if we already know the true value**, then we can measure how any other value or an approximation of the true value is away from the exact value. Because we know the true value, we call this difference/discrepancy the ***true*** numerical error. It is also known as the *absolute error* E_t and is given by:

$$E_t = True\ \ Value\ \ -\ \ Approximation \qquad (1)$$

For example, we know that π has infinite digits: 3.1415926535 ... but we simply approximate it as 3.14. In this case:

$$E_t = 3.1415926535... - 3.14 = 0.0015926535...$$

Most of the time we are interested in the true percent error designated as:

$$\epsilon_t = \frac{True\ \ Value - Approximation}{True\ \ Value} \times 100\% \qquad (2)$$

which in our case of π will be:

$$\epsilon_t = \frac{3.1415926535... - 3.14}{3.1415926535...} \times 100\% \approx 0.05\%$$

In numerical analysis, we do not even know the true value to begin with! In fact, the whole purpose of numerical methods is to find this value. So how are we going to know if the value that the numerical method comes up with is precise and accurate relative to the true answer. We are not able to use the absolute error criterion for obvious reasons; thus, we shall introduce a new one called ***relative errors***.

Relative Errors and the Stopping Criterion

In numerical methods, we often go through multiple iterations using loops, and in each loop, we get a value that gets closer and closer to the true value (the answer). Basically, using equations that describe the problem, we get the first approximation in the first loop iteration; then we use the first approximation in the next iteration to get our second approximation, which in turn we use to get a third approximation and so on.

Now, given that we do not know the true value and we cannot calculate the absolute error, we use the relative error instead. This is the error between the answers that we get between each two successive approximations. It has a similar generic equation like the absolute error. The relative error ϵ_a is calculated as follows:

$$\epsilon_a = \frac{Present \;\; Approximation - Previous \;\; Approximation}{Present \;\; Approximation} \times 100\% \qquad (3)$$

Simply put, we measure ϵ_a between the second and first iterations, then between the third and second, then between the fourth and third. But how many iterations are we going to go over? We have two ways to stop the loop:

1. A fixed number of iterations, say we fix the loop to go for 100 or 200 iterations.
2. Use a *stopping criterion*

The stopping criterion uses an error threshold ϵ_s. In each iteration, we check if the computed relative error ϵ_a becomes less than this threshold ϵ_s. If so, we exit the loop and the last answer is our best approximated answer; otherwise, we keep going into the next iteration. We use the absolute value of ϵ_a in this check:

$$|\epsilon_a| < \epsilon_s \qquad (4)$$

But what is this error threshold ϵ_s? How do we get it?

- Either you specify that you need your approximation to be correct to for example within 0.00001%, or
- You want the approximation to be correct to at least ***n*** significant digits, and we calculate ϵ_s accordingly as:

$$\epsilon_s = (0.5 \times 10^{2-n})\% \qquad (5)$$

So, if we want our final approximation to be correct for the first five significant digits, then $e_s = 0.5 \times 10^{(2-5)}\% = 0.5 \times 10^{-3} = 0.0005\%$

Example: *Maclaurin series expansion* of e^x and Relative Errors

To illustrate this concept, we shall use the *Maclaurin series expansion* of e^x. The value of e^x can be approximated as:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + ... + \frac{x^n}{n!} \qquad (6)$$

Suppose we want to find the value of $e^{0.5}$, we can approximate the answer using the *Maclaurin series* by substituting x by 0.5 and adding the terms together. But how many terms do we need? Two terms? Three terms? Nine terms, or a hundred terms? Suppose we do not know that true value of e^x and that we want to stop when we find an answer whose error is less than $\epsilon_s = 2\%$.

Initially we approximate the value of $e^{0.5}$ by using the first term only, then we approximate it using the second term. We calculate the relative error between these two approximations, and we get $\epsilon_a = 33\%$, which is way higher than $\epsilon_s = 2\%$.

Next, we approximate the value of $e^{0.5}$ using three terms, then we calculate the relative error between the third and second approximations and we get $\epsilon_a = 7.69\%$, which is higher than $\epsilon_s = 2\%$. We continue in this order until we reach an approximation using six terms. When we calculate the relative error between approximations using the six and five terms, we get $\epsilon_a = 0.0158\%$, which is less than $\epsilon_s = 2\%$, and therefore we can stop the approximation.

No. Terms	Substitution	Result	ϵ_t
1	1	1	-
2	1+0.5	1.5	33.3%
3	$1 + 0.5 + \frac{0.5^2}{2}$	1.625	7.69%
4	$1 + 0.5 + \frac{0.5^2}{2} + \frac{0.5^3}{3!}$	1.645833333	1.27%
5	$1 + 0.5 + \frac{0.5^2}{2} + \frac{0.5^3}{3!} + \frac{0.5^4}{4!}$	1.648437500	0.158%
6	$1 + 0.5 + \frac{0.5^2}{2} + \frac{0.5^3}{3!} + \frac{0.5^4}{4!} + \frac{0.5^5}{5!}$	1.648697917	0.0158%

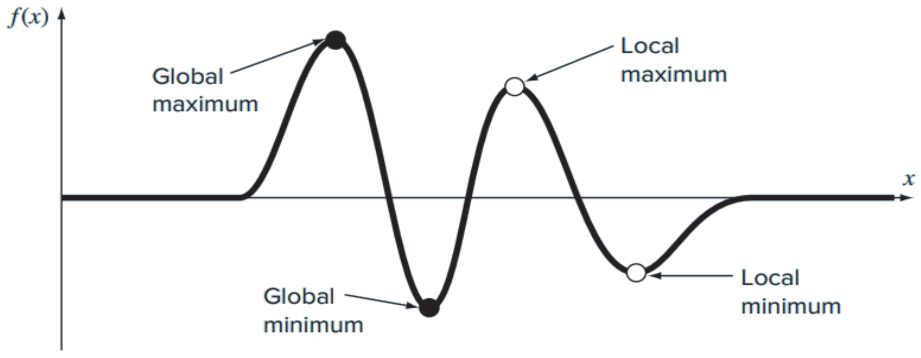
It is worth noting that in order to have a precise and accurate result, we need to use infinite terms which is not practical. We always stop the computation short at some term when we reach an acceptable error margin. Because we stopped at few terms, we say that we truncated the number of terms, and in this case, this type of approximation error is called the **truncation error**.

In the implementation of any numerical method, the total error results from both roundoff errors and/or truncation errors. For most engineering and scientific applications, we must settle for a close value of our approximation within an acceptable error threshold. To reduce (but not necessarily eliminate) total errors

Introduction - What is Optimization?

In engineering, scientific, and economic applications, we normally use terms like highest performance, least overhead, most gains, minimum cost, maximum efficiency, highest speed, highest growth, *etc*. Therefore, we are interested in the points at which the measured or collected data is higher or lower than neighbouring data. When we plot these points (or functions) we can observe from the plot when the minimum or maximum point is located.

Sometimes, the function has one minimum or one maximum point, and we call these functions ***unimodal***. Other functions have shapes like hills and valleys going up and down and can have multiple high points or multiple low points. We call these functions ***multimodal,*** and these points ***local maxima*** and ***local minima*** points. If we are referring to either point, we call them ***local optima***. In such cases, we are mostly interested in the best case which we call the ***global optimum*** which is the point the yields the best solution for the problem understudy (*i.e.,* lowest cost, highest bandwidth, *etc*.). The following figure illustrates an example of local and global optima points for a one-dimensional problem ($f(x)$). Optimization is merely finding the points at which our function has a global maximum or minimum value.

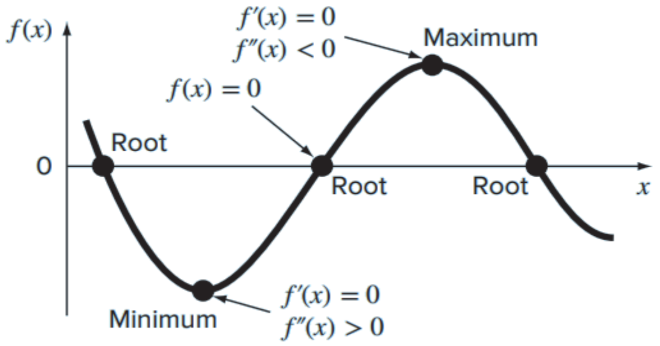


Single Variable Optimization

Of course, we cannot rely on plots to find the local or global optima and we need a systematic mathematical way to do so. From calculus courses, we know that at either the local or global optimum points that the slope of the tangent line touching the point is 0. That is, $f'(x_{opt}) = 0$. Intuitively, we need to differentiate the function $f(x)$ and find the roots of $f'(x)$, and these roots will be the optimum points. However, this still does not tell us if the said point is a maximum or a minimum, we only know that it is an optimum. Again, calculus comes to our aid. By taking the second derivative of the function $f''(x)$, then substituting the values x_{opt} which are the roots we found in the earlier step, then we can determine if the point is a minimum or a maximum by:

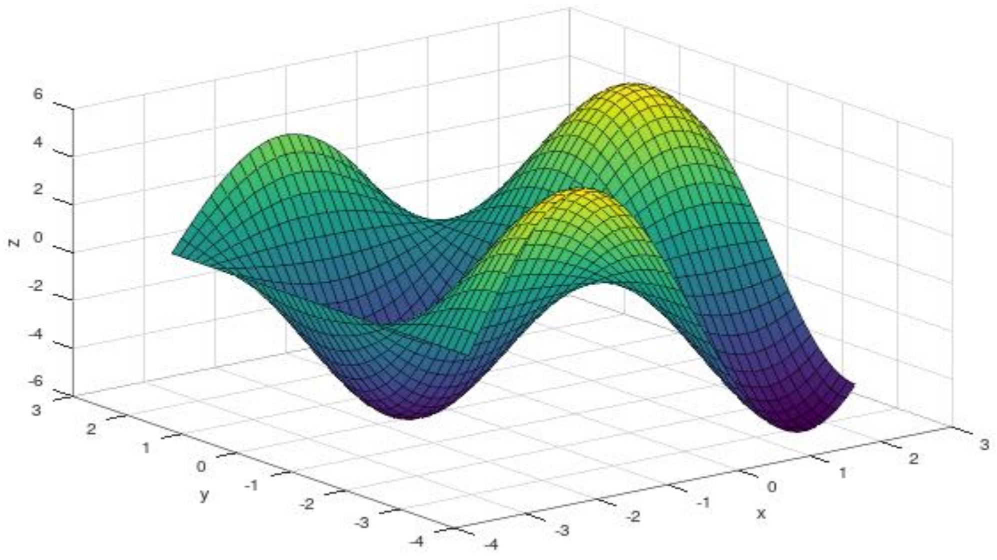
$f''(x_{opt}) > 0,$ then the point is a minimum.

$f''(x_{opt}) < 0,$ then the point is a maximum.



Multi-variable Optimization

Optimization can also be done in two or more dimensions. For example, the function $f(x, y)$ could possibly have the shape shown in the following figure. Notice that the function has multiple *hills* and *valleys.* The peaks and lows of these hills and valleys represent local maxima and minimas, and the highest peaks and lowest valleys represent the global maximums and minimums. We have already learnt how to use different types of 3D plots to draw such graphical representations.



In this course, we are mainly interested in one-dimensional (single variable) optimization. Yet, we will present MATLAB built-in functions for both one- and multi-dimensional optimization.

The Golden Number

Mathematicians have been fascinated by many numbers across history; for example: prime numbers, π, and ϕ (The Golden number). The golden number has been known since the ancient Greeks times and the mathematician Euclid of Alexandria (إقليدس) (the father of geometry) devised a geometric way to find it. It has aesthetical quality and is associated with natural beauty when observed in nature. The Golden number ϕ has the value $\frac{1}{2}(1+\sqrt{5}) = 1.6180889...$. This number has been found to govern many naturally occurring shapes or patterns in nature as the following figure illustrates:

Overview of the Golden-Section Numerical Search (A Bracketing Method)

The Golden-Section search method is a bracketing method. This means it works over an interval (bracket) and finds the minimum value within this interval. The Golden-Section search divides the function into intervals that contain the minimum point, then starts looking for the minimum point by making an assumption (approximation). Based on this approximation, it updates the ends of the interval and makes a new assumption. It compares the current approximation with the previous approximation and computes the relative error, and we compare it to the stoppage criterion. If the relative error is larger than the stoppage criterion, it keeps iterating, once it is smaller, it stops, and the last approximation is the final answer for the optimal point.

Details of the Golden-Section Search Method through an Example

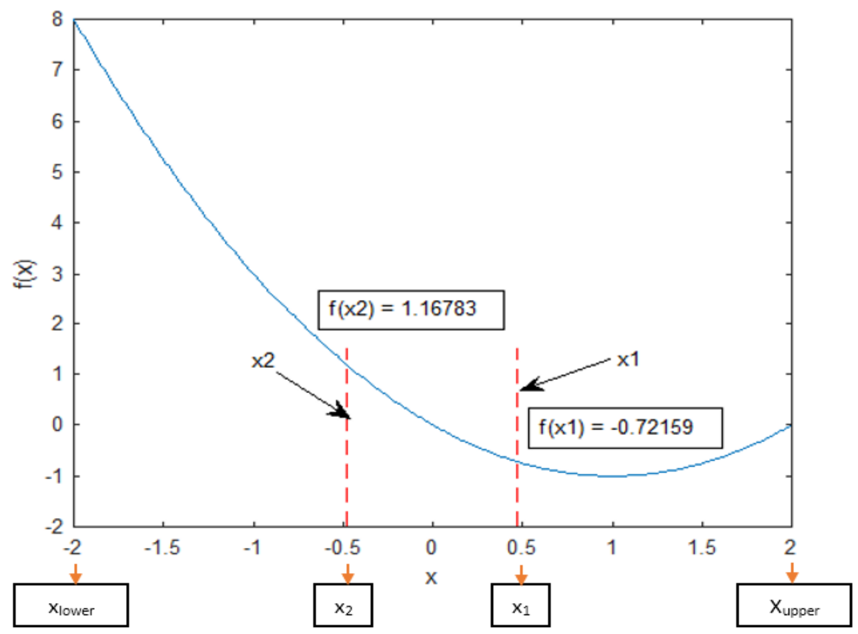
The golden section method can only determine the **minimum** points of a function. We start each iteration with an interval $[x_{lower},\ x_{upper}]$ inside which we know that we have a minimum. As we said, an interval known to have a single minimum (or maximum) is called ***unimodal.*** The algorithm starts with finding two points inside the interval x_1 and x_2 based on the Golden number ϕ that we have already seen. The values of x_1 and x_2 are computed as follows:

$$d = (\phi - 1)(x_{upper} - x_{lower}) \tag{7}$$

$$x_1 = x_{lower} + d \tag{8}$$

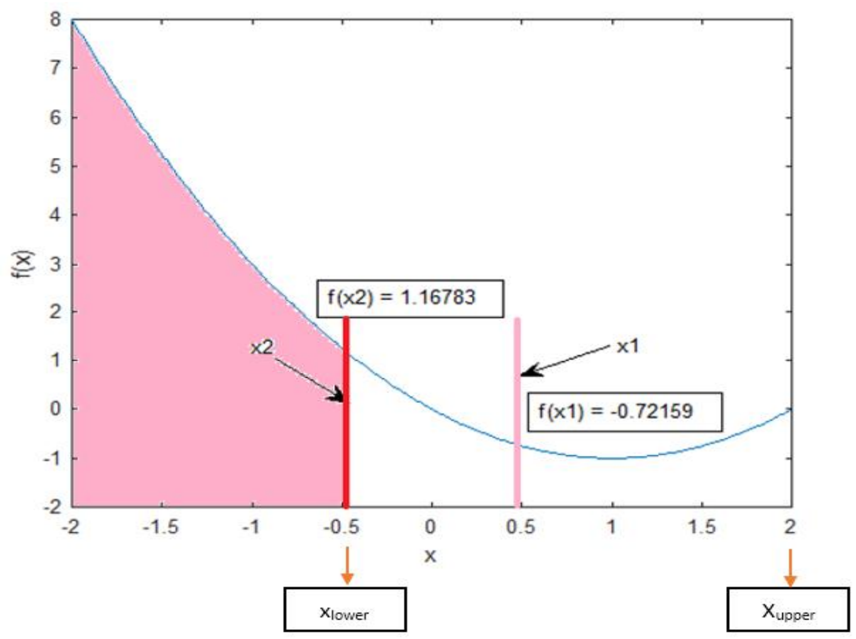
$$x_2 = x_{upper} - d \tag{9}$$

Suppose we have the function $f(x) = x^2 - 2x$ that we graphically can tell it has a minimum at $x = 1$ in the range $[-2,\ 2]$, so for this function $d = 2.4723556$, and therefore $x_1 = 0.4723556$ and $x_2 = -0.4723556$.



The numerical method proceeds by computing $f(x_1) = f(0.4723556) = -0.72159$ and $f(x_2) = f(-0.4723556) = 1.16783$, and remember that we are looking for the minimum value, so we do the following comparisons:

- If, as in the figure, $f(x_1) < f(x_2)$, then we assume that $f(x_1)$ is the approximate minimum in this iteration, and all values to the left of x_2, from x_{lower} to x_2 will be ignored because they do not contain the minimum. We update the interval such that x_2 becomes the new x_{lower} for the next iteration.
- If $f(x_2) < f(x_1)$, then we assume $f(x_2)$ is the approximate minimum in this iteration, and all value of x to the right of x_1, from x_1 to x_{upper} will be ignored. We update the interval such that x_1 becomes the new x_{upper} for the next iteration.



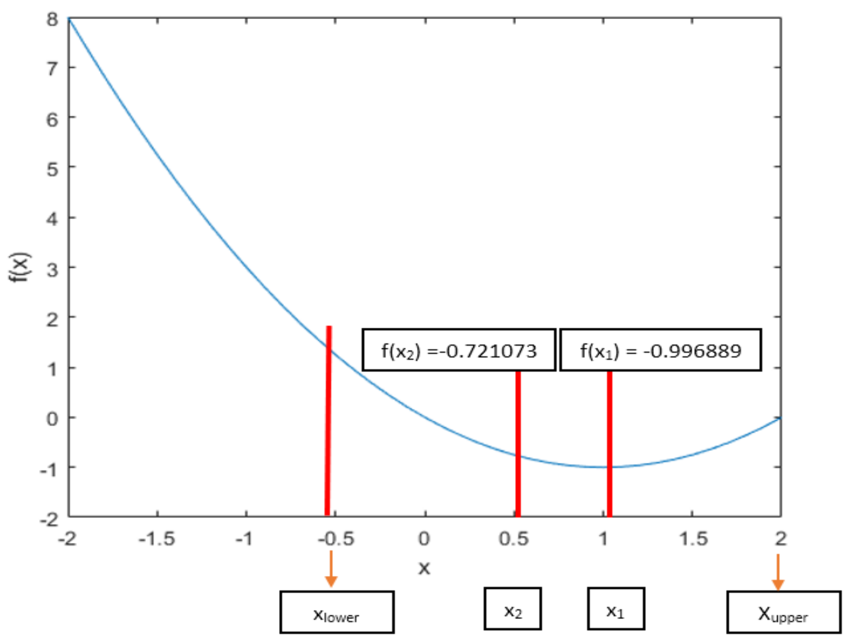
The new interval is now $[x_{lower}, \ x_{upper}]$ = [-0.472356, 2].

In the next iteration, the distance d is recomputed based on the new interval $[x_{lower}, \ x_{upper}]$ = [-0.472356, 2], and we get $d = 1.528136$. Therefore, the values for:

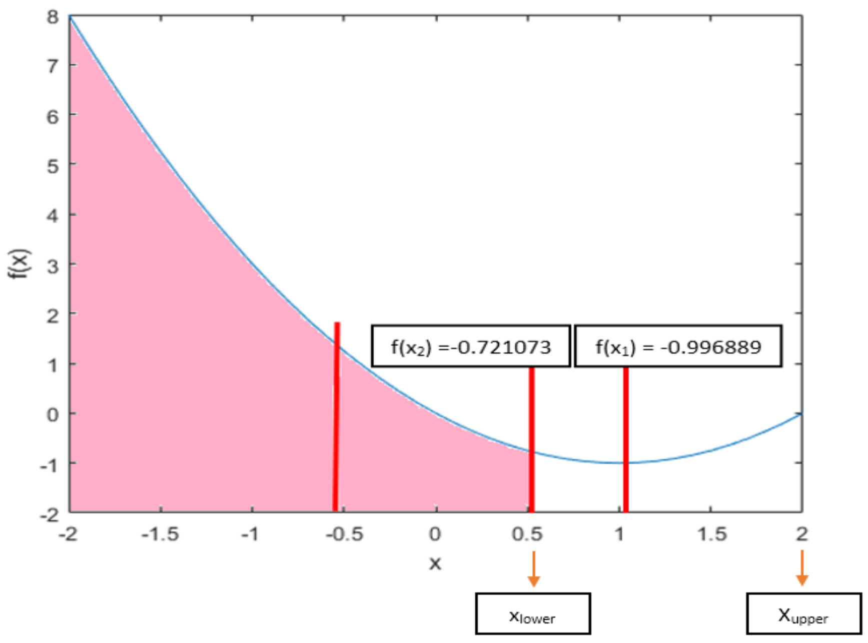
$x_1 = -0.472356 + 1.528136 = 1.05578$

$x_2 = 2 - 1.528136 = 0.471864$

The numerical method proceeds by computing $f(x_1) = f(1.05578) = -0.996889$ and $f(x_2) = f(0.471864) = -0.721073$



If, as in the figure, $f(x_1) < f(x_2)$, then we assume that $f(x_1)$ is the NEW approximate minimum in this iteration, and all values to the left of x_2, from x_{lower} to x_2 will be ignored because they do not contain the minimum. We update the interval such that x_2 becomes the new x_{lower} for the next iteration.



Also, now that we have two iterations, and old approximate and a new approximate, we can calculate the relative error. Because we need to compare it to a stoppage criterion in order to know when to exit the loop. We already know the equation for the relative error from a previous section. However, mathematicians came up with new relative error formulae for the golden search algorithm:

$$\epsilon_a = (2\phi - 3)\left|\frac{(x_{upper} - x_{lower})}{x_{opt}}\right| \times 100\% \tag{10}$$

or

$$\epsilon_a = (2 - \phi)\left|\frac{(x_{upper} - x_{lower})}{x_{opt}}\right| \times 100\% \tag{11}$$

where x_{lower} and x_{upper} are the ones used in each iteration.

We compute ϵ_a, based on the second equation and we find that $\epsilon_a = 323.6\%$, a huge error, so we need to continue with more iterations.

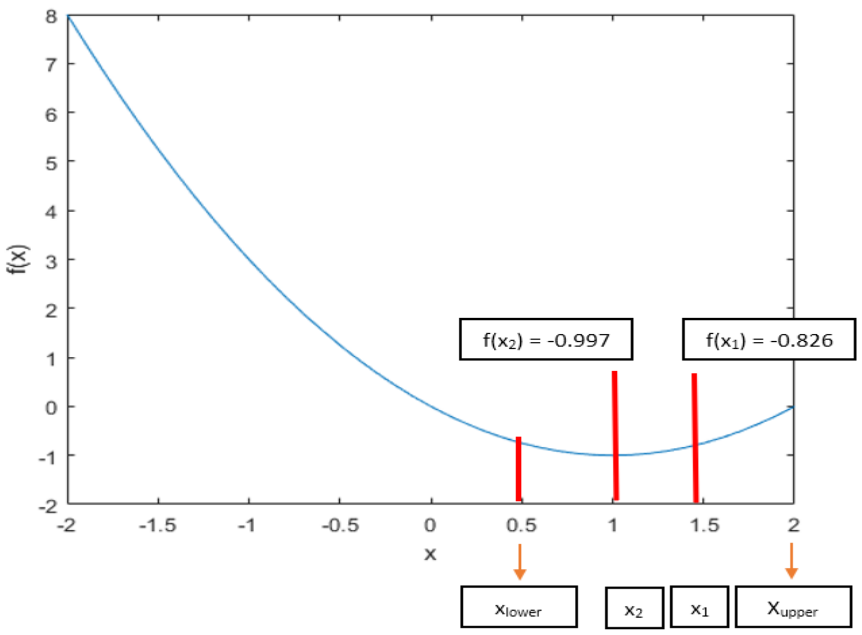
The new interval is now $[x_{lower},\ x_{upper}]$ = [+0.472356, 2].

In the next iteration, the distance d is recomputed based on the new interval $[x_{lower},\ x_{upper}]$ = [+0.472356, 2], and we get $d = 0.944524$. Therefore, the values for:

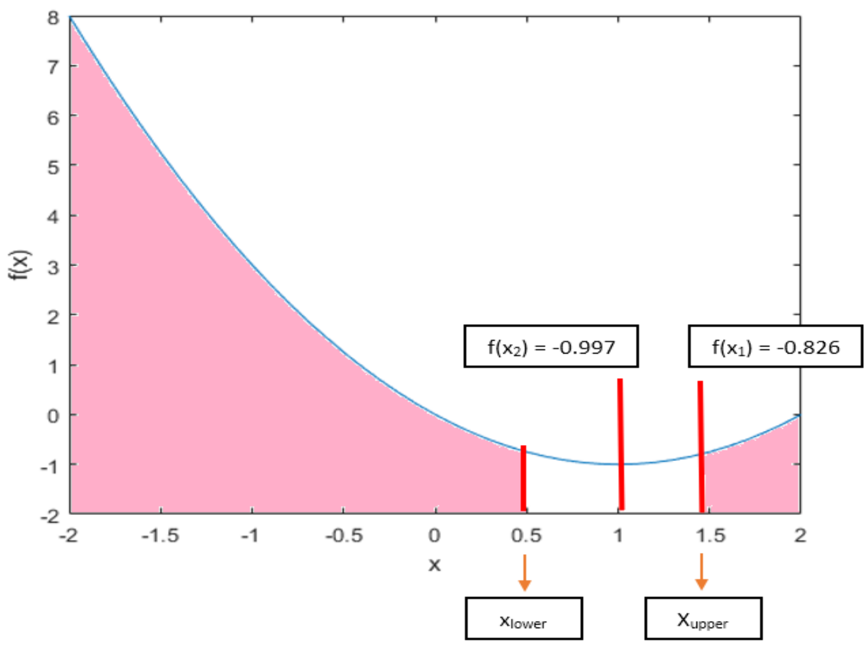
$x_1 = +0.472356 + 0.944524 = 1.416388$

$x_2 = 2 - 0.944524 = 1.055476$

The numerical method proceeds by computing $f(x_1) = f(1.416388) = -0.826621$ and $f(x_2) = f(1.055476) = -0.996922$



Notice now that $f(x_2) < f(x_1)$, so we assume $f(x_2)$ is the approximate minimum in this iteration, and all value of x to the right of x_1, from x_1 to x_{upper} will be ignored. We update the interval such that x_1 becomes the new x_{upper} for the next iteration.



We compute ϵ_a based on the second equation and we find that $\epsilon_a = 200.1\%$, a huge error, so we need to continue with more iterations.

The new interval is now $[x_{lower},\ x_{upper}]$ = [+0.472356, 1.416388].

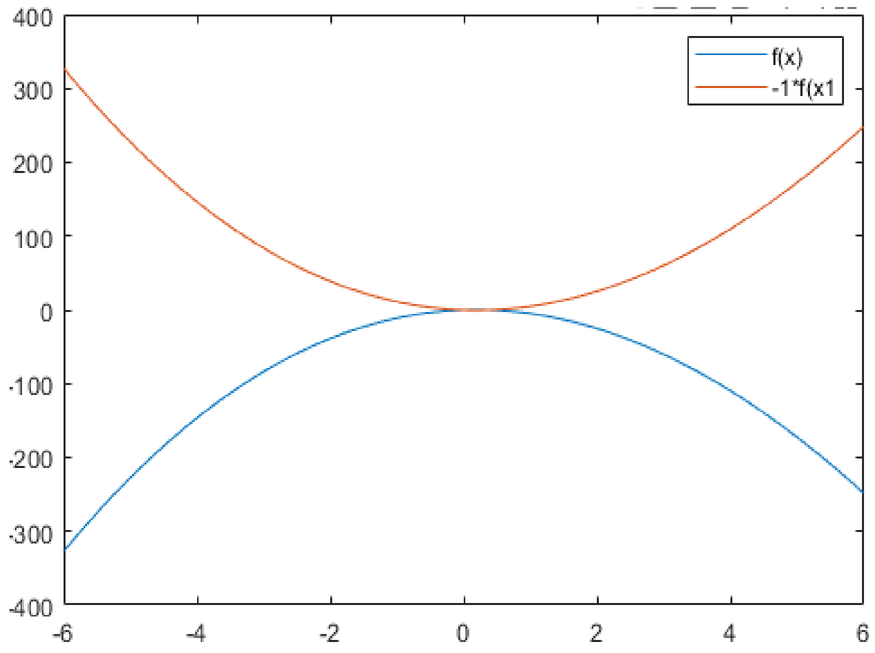
The following table summarizes the first 10 iterations of golden search algorithm. The value in bold represents the pair $(x, f(x))$ at which the minimum occurs in the iteration. We copy the value of x as x_{opt} to denote that this is the optimal estimate thus far in this iteration.

	x_{lower}	$xupper$	d	x_1	x_2	$f(x_1)$	$f(x2)$	x_{opt}	ϵ_a
1	-2	2	2.472356	**0.472356**	-0.472356	**-0.721591**	1.167831	**0.472356**	1170.2%
2	-0.472356	2	1.528136	**1.055780**	0.471864	**-0.996889**	-0.721073	**1.055780**	323.6%
3	0.471864	2	0.944524	1.416388	**1.055476**	-0.826621	**-0.996922**	**1.055476**	200.1%
4	0.471864	1.416388	0.583800	**1.055664**	0.832589	**-0.996902**	-0.971973	**1.055664**	123.6%
5	0.832589	1.416388	0.360840	1.193429	**1.055548**	-0.962585	**-0.996914**	**1.055548**	76.4%
6	0.832589	1.193429	0.223031	1.055620	**0.970397**	-0.996906	**-0.999124**	**0.970397**	51.4%
7	0.832589	1.055620	0.137853	**0.970442**	0.917767	**-0.999126**	-0.993238	**0.970442**	31.8%
8	0.917767	1.055620	0.085205	**1.002972**	0.970414	**-0.999991**	-0.999125	**1.002972**	19.0%
9	0.970414	1.055620	0.052665	1.023079	**1.002955**	-0.999467	**-0.999991**	**1.002955**	11.7%
10	0.969896	1.023079	0.032872	**1.002768**	0.990207	**-0.999992**	-0.999904	**1.002768**	7.3%

Notice that the result is slowly approaching the minimum value which is 1.

What about the maximum points?

When we introduced the golden-search bracketing method, we said that this numerical method is used to get the minimum value of a function in a certain interval. We can readily use the golden search to get the maximum by a simple twist; when we multiply the function by -1, we simply flip the function upsides down, so when we search for the minimum of $-f(x)$ (orange line), then we are looking for the maximum of $f(x)$ (blue line) as the next figure illustrates:



MATLAB Optimization Built-In Functions

MATLAB's **fminbnd** function takes as an input a function for which you want to find the minimum, and an interval for that function to search in between. To apply **fminbnd** on our previous example:

```
y = @(x)  x.^2 - 2.*x;
[x_opt, y_x] = fminbnd (y, -2, 2)
x_opt =
     1
y_x =
    -1
```

The function **fminbnd** uses hybrid techniques to find the minimum within an interval. It is mainly based on the golden search we described above, and another technique called the parabolic interpolation. We can see the iterations MATLAB uses and the numerical method by enabling some options:

```
y = @(x)  x.^2 - 2.*x;
options = optimset('display','iter');
[x_opt, y_x]  = fminbnd(y,-2, 2,options)

 Func-count     x          f(x)         Procedure
    1      -0.472136      1.16718        initial
    2       0.472136     -0.72136        golden
    3        1.05573    -0.996894        golden
    4              1           -1        parabolic
    5        1.00003           -1        parabolic
    6       0.999967           -1        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04
x_opt =
     1
y_x =
    -1
```

If you have a multi-dimensional function such as $f(x, y)$ and you need to find its minimum value using MATLAB, then you can use the **fminsearch** command. You can specify if you need the minimum near a specific point, or within any interval. If you attempt defining your function as follows, it will fail. This is because fminsearch passes one variable to the function, whereas it takes two variables:

```
f=@(x,y) 100*(y - x^2)^2 + (1 - x)^2
[x,fval]=fminsearch(f,[-0.5,0.5])
```

To circumvent this issue, we design the function such that it accepts one value, but we pass a vector denoting, x, y, z, *etc.*

```
fun = @(m)100*(m(2) - m(1)^2)^2 + (1 - m(1))^2;   %m(1) is x, m(2) is y
x0 = [-1.2,1];
x = fminsearch(fun,x0)
x = 1×2
   1.000022021783570   1.000042219751772
```

Dr. Ashraf Suyyagh - All Rights Reserved
Experiment version 2.0
Original Experiment December 17th, 2020
Last Updated April 22nd, 2022

Revision History

Ver. 2.0

- Split the optimization part from the old lab and made it into its own experiment.
- Split the error Analysis from the root finding experiment and merged with the optimization experiment.
- Removed the derivation of the Golden number.
- Rewrote the entire sections of absolute and relative errors, added more examples.
- Clarified the stoppage criterion section and gave a more detailed example
- Rewrote the Golden search algorithm section by clarifying the steps through a step by step numerical example and added figures at each step

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 8 - Numerical Methods for Finding Roots

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

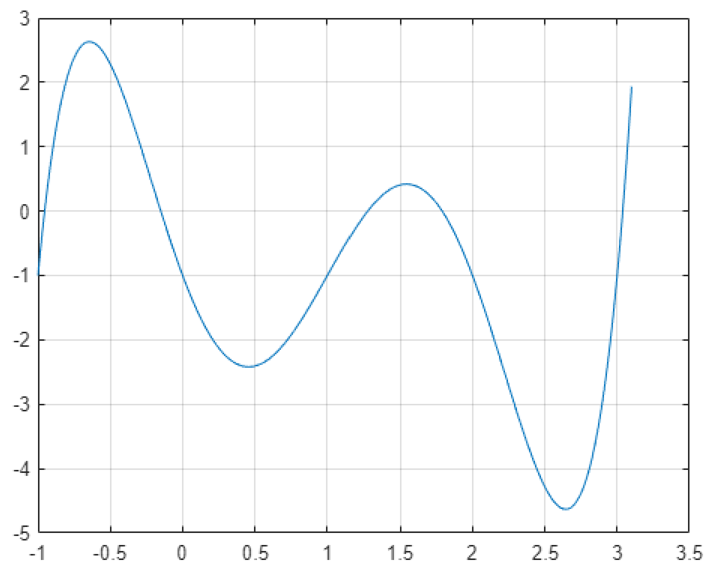
Numerical Methods for Finding Roots

We are all familiar with the quadratic formula $f(x) = ax^2 + bx + c = 0$ whose roots can be directly computed using the equation:

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

Remember that the roots are the values of x that make $f(x)$ evaluate to 0. But what if $f(x)$ is not a quadratic formula? How then can we find the roots of $f(x)$? Crude methods include plotting the function and observing where the function intersects with the x-axis. For example, you can estimate the roots of this polynomial equation of degree five $f(x) = x^5 - 5x^4 + 5x^3 + 5x^2 - 6x - 1$ by plotting the function:

```
x = -1:0.01:3.1;
y = x.^5 -5*x.^4 + 5*x.^3 + 5*x.^2- 6*x - 1;
plot(x,y)
grid on
```



And we can observe that this function has five roots.

Another method involves *trial and error* by guessing the value of x. Both techniques are obviously inefficient and inadequate for the requirements of engineering and science practice. The former is imprecise, while the latter is time-consuming. Numerical methods represent approximate alternatives but employ systematic strategies to close in on the true root.

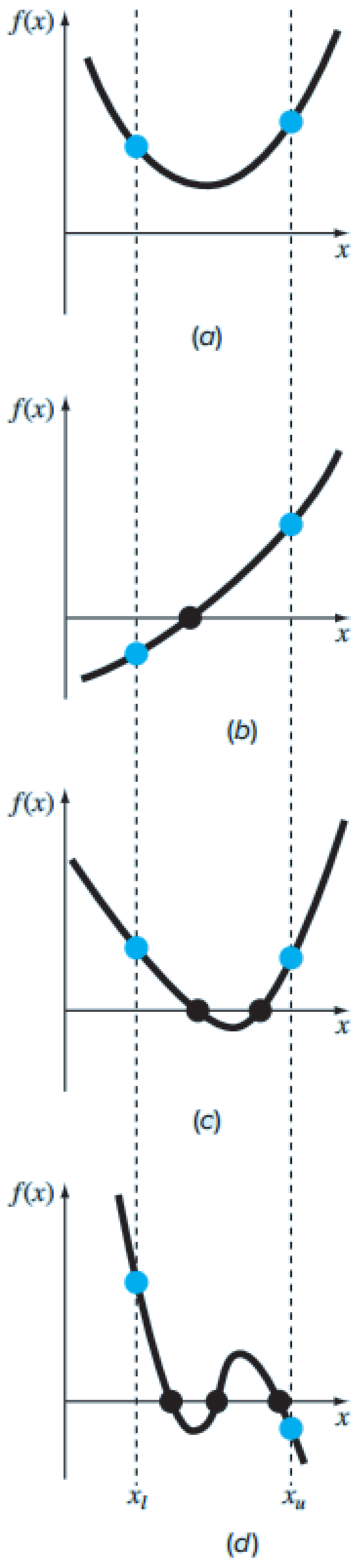
The Bracketing Methods

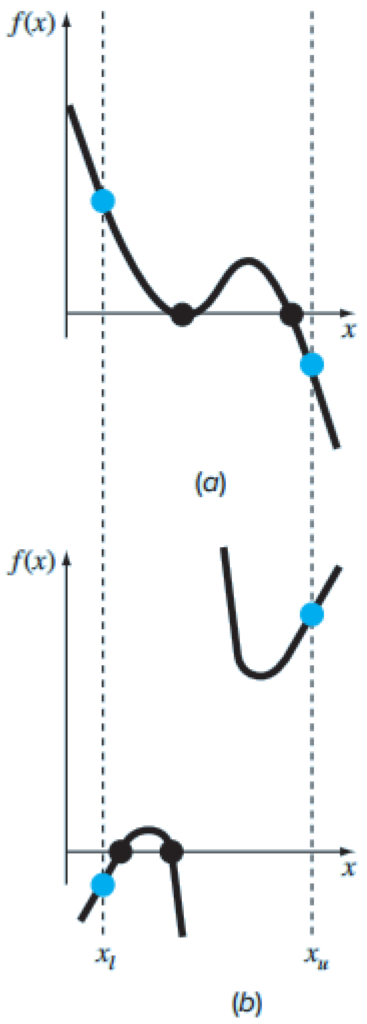
We have already seen in the previous experiment how we found the minimum value of a unimodal function by "bracketing" it inside an interval with a lower and upper bound, and each iteration, we modified this interval. In each interval, the interval (bracket) gets shorter and shorter towards the true solution. We will do a similar thing to find roots.

Since we know that the root is the value that makes the function $f(x)$ = 0, then we must find two values such that one is positive; and one is negative. This way, we know that the function must cross the x-axis in order to change sign, this crossing is a root since crossing the x-axis at a value x means that $y = f(x) = 0$.

The problem we face is how to choose the two values that bracket the root. The figure to the left represents the generic cases we might encounter. If we have two values of the same sign, this means that we either have no roots in between (Left Figure (a)), or an even number of roots (Left Figure -c)). If we have two numbers with opposite signs, this means they might bracket an odd number of roots (Left Figure (b) and (d)).

There are exceptions to the generic rule which we present in the figure to the right. We can have a bracket of two values of opposite signs that encompass an even number of roots if some of these roots are tangential to the x-axis; that is, the root just touches the x-axis without crossing it (Right Figure (a)). We can also have an even number of roots if the function is discontinuous (Right Figure (b))


$f(x)$
x
(a)
$f(x)$
x
(b)
$f(x)$
x
(c)
$f(x)$
x
x_l
x_u
(d)


$f(x)$
x
(a)
$f(x)$
x
x_l
x_u
(b)

To find the roots, there are two steps:

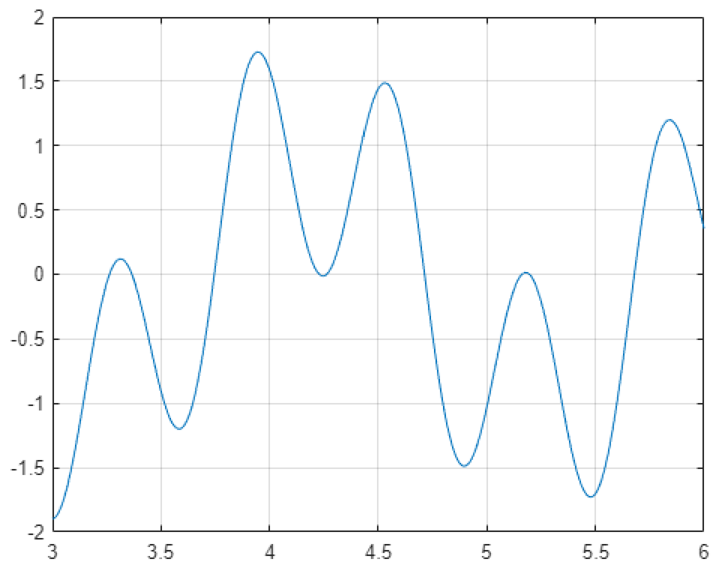
- Find the intervals where the function changes sign because we know that inside this interval we will have a root. To search for these intervals, we will use the *Incremental Search* method.
- Find the approximate value of the root inside each interval that we obtained in the previous step. For this step, we can use the *Bisection* method

Bracketing Method 1: Incremental Search

The incremental search works on real continuous functions and tries to find an interval where the function changes signs. It provides us with the number of roots but not the exact value of the root. It starts with dividing the function into n intervals of a certain width (spacing). If it finds that $f(x_1)$ and $f(x_2)$ have opposite signs, then there must be a root in between. If the distance (spacing) between the numbers is too small, the search can be very time consuming. On the other hand, if the distance is too great, there is a possibility that closely spaced roots might be missed. The problem is compounded by the possible existence of multiple roots.

Example: When plotting the function $f(x) = sin(10x) + cos(x)$, we will observe it has nine roots in the interval [3, 6]. Note that there are few roots that are too close to each other that you might mistake them for one root; you need to zoom into the function to distinguish between them!

```
x = 3:0.01:6;
y = sin(10*x) + cos(3*x);
plot(x,y)
grid on
```



If we want to write a code that divides the function into 50 brackets and tries to count the number of brackets that have roots in between, we can write:

```
x = linspace(3,6,50);
y = sin(10*x) + cos(3*x);
numberBrackets = 0;
xb = []; %xb is null unless sign change detected
for k = 1:length(x)-1
    if sign(y(k)) ~= sign(y(k+1)) %check for sign change
        numberBrackets = numberBrackets + 1;
        xb(numberBrackets,1) = x(k);
        xb(numberBrackets,2) = x(k+1);
    end
end

if isempty(xb) %display that no brackets were found
    disp('no brackets found')
else
    disp('number of brackets:') %display number of brackets
    disp(numberBrackets)
    disp(xb)
end
```

```
number of brackets:
     5
    3.2449    3.3061
    3.3061    3.3673
    3.7347    3.7959
    4.6531    4.7143
    5.6327    5.6939
```

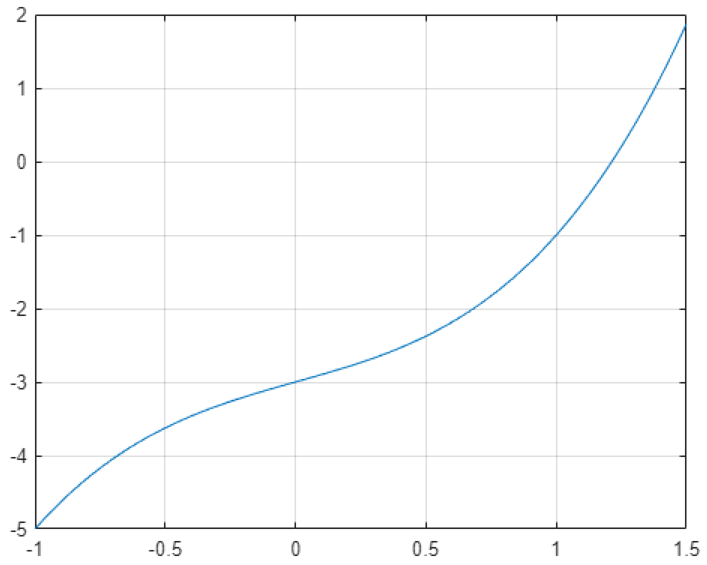
The code returned five brackets that changed sign, implying that we have five roots whereas we know that the function has nine roots in the same interval. This error is due to the fact that we have used too wide steps (small number of intervals). If we increase the number of intervals to 100 and run the code again, we would get nine brackets implying nine roots which is the correct answer.

Bracketing Method 2: Bisection

The *bisection method* is used to find an approximation of a root within an interval. In each iteration, the search interval is divided in half. If a function changes sign over an interval, the function value at the midpoint is evaluated. The location of the root is then determined as lying within the subinterval where the sign change occurs. The subinterval then becomes the interval for the next iteration. The process is repeated until the root is known to the required precision.

Suppose we have the cubic function $f(x) = x^3 + x - 3$. If we plot this function over the interval [-1, 1.5], we will know that it has only one root.

```
x = -1:0.01:1.5;
y = x.^3  + x - 3;
plot (x, y)
grid on
```



We will use the bisection method to *approximate* this root:

1. The first step is to find x_1 and x_2 such that $f(x_1)$ and $f(x_2)$ have opposite signs. Say 0 and 1.5, for $f(0) = -3$ and $f(1.5) = 1.875$.
2. In the second step, we take the midway point between 0 and 1.5 as our first approximate root, which is 0.75 and compute $f(0.75) = -1.828125$
3. Given that $f(0.75)$ is negative, we know that our second approximation of the root is between 0.75 and 1.5, so we take the midway point between them (= 1.125) as our second root approximation. We compute $f(1.125) = -0.451171875$
4. Given that $f(1.125)$ is negative, we know that our third approximation of the root is between 1.125 and 1.5, so we take the midway point between them (= 1.3125) as our third root approximation. We compute $f(1.3125) = 0.573486328125$
5. Given that $f(1.3125)$ is positive, we know that our fourth approximation of the root is between 1.125 and 1.3125, so we take the midway point between them (= 1.21875) as our fourth root approximation. We compute $f(1.21875) = 0.029022216796875$

So, when do we stop? Which value do we think is a good enough approximation of the root? Our approximations were 0.75, 1.125, 1.3125 and 1.21875. We must compute ϵ_a between each two successive approximations. This will yield 33.33%, 14.29%, 7.69%

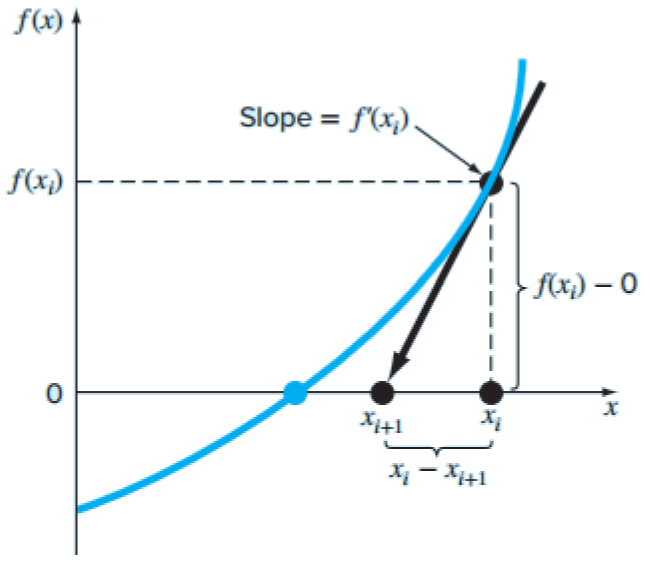
As you can see, the error in approximating the root is decreasing, but it is still a large error. In this case, you might need to choose a value for ϵ_s such that when the approximation error falls below it, you will stop and consider the value you stopped at as a good enough approximation.

Open Methods

For well-posed problems, the bracketing methods always work but converge slowly (*i.e.,* they typically take more iterations to home in on the answer). In contrast, the open methods do not always work (*i.e.*, they can diverge), but when they do they usually converge quicker. In this section we present one of the most widely used open methods for finding roots.

Newton-Raphson Method

The Newton-Raphson method starts with an initial random guess x_i. At the point $f(x_i)$, we draw a tangent line that we extend until it intersects with the x-axis at a new point x_{i+1}. This new point is regarded as a better approximation of the root. As we repeat this operation, the crossing of the tangent lines gets closer and closer to the true root.



However, to convert this into a mathematical formula, we can use the concept of the derivative. We know that the first derivative at x is going to be the slope of the tangent line, and we know that the slope is computed as $\Delta y/\Delta x$ which in our case is:

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

when we arrange the terms, we get:

$x_{i+1} = x_i - \dfrac{f(x_i)}{f'(x_i)}$ which is called the Newton-Raphson formula.

Example: Suppose we want to find the root of $f(x) = e^{-x} - x$, the first step is to find the derivative $f'(x) = -e^{-x} - 1$, and start from a random initial guess, say $x_i = 0$. We substitute the functions in the Newton-Raphson formula:

$$x_{i+1} = x_i - \frac{e^{-x_i} - x_i}{-e^{-x_i} - 1}$$

We substitute $x_i = 0$, which will give us $x_{i+1} = 0.5$, then we substitute this value again to give us a new $x_{i+1} = 0.566311003$ and so on. We can then compute ϵ_a after each successive approximation and we can as well have a stoppage criterion ϵ_s of our choice. The following table shows the successive approximations for this example. Notice how quickly we converged to the root and how extremely small the approximation error is.

i	x_i	$\|\varepsilon_t\|$, %
0	0	100
1	0.500000000	11.8
2	0.566311003	0.147
3	0.567143165	0.0000220
4	0.567143290	$<10^{-8}$

To solve the above example in code, one can write:

```
y = @(x) exp(-1*x) - x;
yd = @(x) -1*exp(-1*x) - 1;
x = 0;
maxit = 50;
iter = 0;
es = 0.0005;
while (1)
    xold = x;
    x = x - y(x)/yd(x);
    iter = iter + 1;
    if x ~= 0
        ea = abs((x - xold)/x) * 100;
    end
    if ea <= es || iter >= maxit
        break
    end
end
root = x
```

```
root = 0.5671
```

MATLAB Built-in Functions for Finding Roots

MATLAB is a numerical tool whose built-in functions employ these techniques that you are learning in this course. The **fzero** function is designed to find the real root of a single equation. A simple representation of its syntax is

```
fzero(function,x0)
```

where function is the name of the function being evaluated, and x_0 is the initial guess. The function must be written as an anonymous function. Suppose we want to find the roots for $f(x) = x^2 - 9$, we know that it has two roots -3 and 3, we can use the function **`fzero`** to find the closest root to our initial guess:

```
fzero(@(x) x.^2 - 9, -2)
```

```
ans = -3
```

or

```
fzero(@(x) x.^2 - 9, 4)
```

```
ans = 3
```

Note that this function only returns one root at a time!

MATLAB Polynomials and the `roots/poly` Commands

A polynomial can mathematically be described as $f_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_2x^2 + a_1x + a_0$ where n is the order of the polynomial, and the *a*'s are constant coefficients . For such cases, the roots can be real and/or complex. In general, an *n*th order polynomial will have *n* roots.

Suppose we have the function $f_1(x) = x^5 - 3x^4 + 2x^3 - x^2 + x + 2$, this polynomial can be represented in MATLAB by having a vector of its ordered coefficients from highest order to lowest:

```
c1 = [1 -3 2 -1 1 2]
```

```
c1 = 1×6
     1    -3     2    -1     1     2
```

The function $f_2(x) = x^2 - 9$ can be represented as:

```
c2 = [1 0 -9]
```

```
c2 = 1×3
     1     0    -9
```

Notice that missing terms have a coefficient of 0, also do not forget the signs of the terms.

You can use this representation of MATLAB polynomials with the **`roots`** command to find the polynomial roots; for example:

```
roots (c1)
```

```
ans = 5×1 complex
   2.0000 + 0.0000i
   1.6180 + 0.0000i
   0.0000 + 1.0000i
   0.0000 - 1.0000i
  -0.6180 + 0.0000i
```

returns three real roots and two imaginary roots for $f_1(x)$, and the command:

```
roots (c2)
```

```
ans = 2×1
      3
     -3
```

returns the two roots for the function $f_2(x)$.

The **poly** command is the inverse of the **roots** command; that is, it takes the roots and returns the polynomial. For example:

```
poly([-3 3])
```

```
ans = 1×3
      1      0     -9
```

```
                                    Experiment version 1.1
                    Original Experiment December 10th, 2020
                              Last Updated May 10th, 2022
                     Dr. Ashraf Suyyagh - All Rights Reserved
```

```
Revision History

Ver. 1.1
Moved the Error Analysis to a previous experiment.
Clarified that there are two steps involved in finding the root.
Corrected the polynomial equation notation to be consistent with previous
  experiments.
```

University of Jordan

School of Engineering and Technology

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Experiment 9 - Differentiation and Integration

Material prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Differentiation

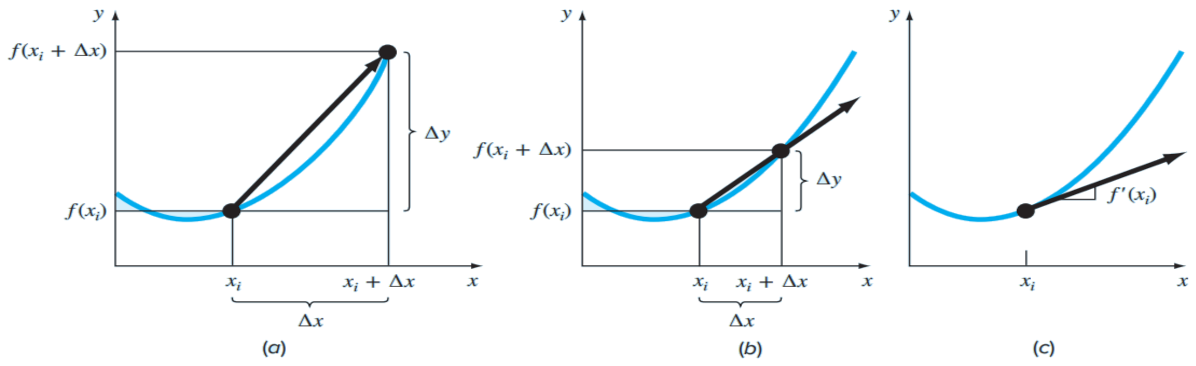
Differentiation is one of the most important operations in Calculus. In this section, we will introduce three variations of one numerical method that we use to compute the differential of a one-dimensional function $f(x)$ at a point x. These techniques are derived directly from the definition of the differential. They are called the forward, backward, and central derivative methods. They are considered less accurate than more elaborate techniques such as the Richardson technique or the high accuracy differentiation formulas.

Slope of a function

The slope of a straight line (linear function) is calculated by using any two points on that line (x_1, y_1) and (x_2, y_2), then calculating the *y*-difference between these two points over the distance between these two points (*x*-difference):

$$Slope = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \quad (1)$$

But what if the function is not linear and we need to calculate the slope of the line tangent to a point $(x_i, f(x_i))$ on the line? We start by choosing any other random point $(x_2, f(x_2))$, then measure the slope between these two points. Notice that x_2 is some distance Δx from x_i; that is x_2 is $x_i + \Delta x$. But if Δx is large, then we do not get an accurate result, because the resulting line is not the tangent (see leftmost figure). If we choose another point closer to x_i, that is Δx is small, then we get closer to a line that is the tangent line (see middle and right most figures).



The Derivative

We know from calculus that the derivative of a function $f(x)$ at a point x_i is derived from the definition of the slope. The derivative of $f(x_i)$ at point x_i is defined as the slope of the tangent line barely touching $f(x_i)$. As we have just seen, to theoretically compute the derivative of $f(x)$ at x_i, we must have Δx to be as close as possible to zero but not be zero:

$$f'(x) = \frac{\delta y}{\delta x} = \lim_{\Delta x \to 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \quad (2)$$

Numerically, however, it is hard, if not impossible, to have the limit of Δx approach zero, therefore, we contend with the following approximation:

$$f'(x) \approx \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} \quad (3)$$

To represent this equation with a true assignment operator, we must acknowledge the fact that there is an error due to the step size Δx:

$$f'(x_i) = \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x} + E(\Delta x) \quad (4)$$

The above definition is called the ***forward difference approximation of the first derivative***. This is because we are using a forward point ($x_i + \Delta x$ is to the right of x_i) to compute the derivative. Let us see how the forward difference approximation of the first derivative works through a numerical example:

Suppose we have the function $f(x) = -0.1x^2 + 0.35x + 1$, and that we want to compute the derivative at $x = 1$. We know from Calculus that $f'(1) = 0.15$, but let us try to approximate it numerically:

Let's start by having Δx = 0.5, then:

$f'(1) \approx \dfrac{f(1.5) - f(1)}{0.5} = 0.1$ and $E(\Delta x)$= 0.15 - 0.1 = 0.05, therefore $\epsilon_t = 33.3\%$

If we keep choosing smaller Δx, say $\Delta x = 0.1$, then:

$f'(1) \approx \dfrac{f(1.1) - f(1)}{0.1} = 0.14$ and $E(\Delta x)$= 0.15 - 0.14 = 0.01, therefore $\epsilon_t = 6.67\%$

and if we try another smaller step where $\Delta x = 0.01$, then:

$f'(1) \approx \dfrac{f(1.01) - f(1)}{0.01} = 0.148999..$ and $E(\Delta x)$= 0.15 - 0.148999... = 0.01, therefore $\epsilon_t = 0.667\%$

Notice that in the above example, we were able to compute ϵ_t because we know the true value of the derivative mathematically.

In reality, we don't know the actual mathematical derivative, and in fact it is what we want the computer to do; to find the derivative for us. In this case, we will do what we have already done before, we will initially approximate the derivative using a large random Δx, and in each successive iteration we make Δx smaller (*i.e.* divide it by 1.5, or 2, or 3, *etc*.). Then compute ϵ_α between each two successive iterations, and if ϵ_α is less than ϵ_s, we stop.

Backward and Centered Difference Approximation of the First Derivative

What if we choose a point to the left of x_i instead of a one to the right of x_i? We can redefine the derivative approximation as:

$$f'(x_i) = \frac{f(x_i) - f(x_i - \Delta x)}{\Delta x} + E(\Delta x) \qquad (5)$$

This way, we apply the formula using x_i and a **previous** value, for example:

$f'(1) \approx \dfrac{f(1) - f(0.99)}{0.01} = 0.151$ and $E(\Delta x)$ = 0.151 - 0.15 = 0.01, therefore $\epsilon_t = 0.667\%$

Both the forward and backward difference approximation methods have a minor drawback; our intention is to approximate the derivative at x_i, but since we use a second point either to the left or to the right of x_i, then in our approximation of the derivative, it will be for a middle point between x_i and the second point, that is at a point $x_i \pm \dfrac{\Delta x}{2}$ and not the derivative at x_i.

To have an approximation of the derivative exactly at x_i, we use a third variation called the centered difference approximation. This numerical method uses a value before x_i and a value after x_i such that x_i is centered in between. So Δx must be the same in either direction. Therefore, we redefine the derivative as:

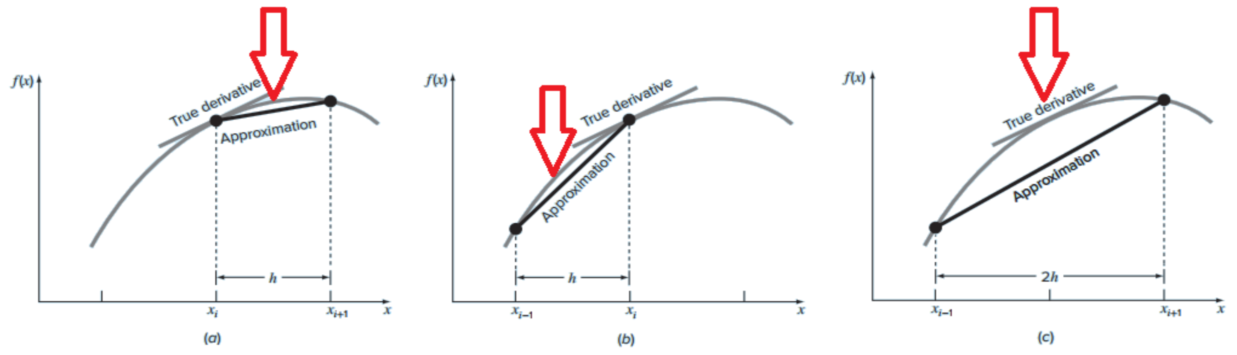
$$f'(x_i) = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2\Delta x} + E(\Delta x) \qquad (6)$$

$$\text{such that } x_i = \frac{(x_i + \Delta x) + (x_i - \Delta x)}{2} \qquad (7)$$

For example:

$f'(1) \approx \frac{f(1.05) - f(0.95)}{0.1} = 0.15$, $E(\Delta x)$ = 0.15 - 0.15 = 0.00, therefore $\epsilon_t = 0\%$

Notice that we used a larger step size Δx of 0.1 rather than 0.01, and we managed to get an accurate result better than the forward or backward difference approximation with Δx = 0.01 because we are having the derivative centered at x_i from the start.



Higher Order Approximations of the First Derivative

The set of equations that approximate the first derivative that we have seen so far are actually based on the Taylor series expansion of the derivative. In fact, they are based on only the first two terms, and truncating the rest, so the error $E(\Delta x)$ can also be thought of as a truncation error. If we include more terms, then the error $E(\Delta x)$ will become smaller and the results will be more accurate. The equations that approximate the differential using more terms are called higher order approximations of the first derivative. In this section, we will only list the three variants of the forward, backward and centered approximations when use three terms of the Taylor series. You can look up the derivations in any numerical methods or Calculus books but they are out of scope of this introductory course. Notice that in what we already explained, we only used two points in our previous approximations. Here, we are using three to four points in our equations:

The forward higher order approximation version:

$$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{x_{i+2} - x_i} \qquad (8)$$

The backward higher order approximation version:

$$f'(x_i) = \frac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{x_i - x_{i-2}} \tag{9}$$

The centered higher order approximation version:

$$f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12\Delta x} \tag{10}$$

Derivative of an Entire Function

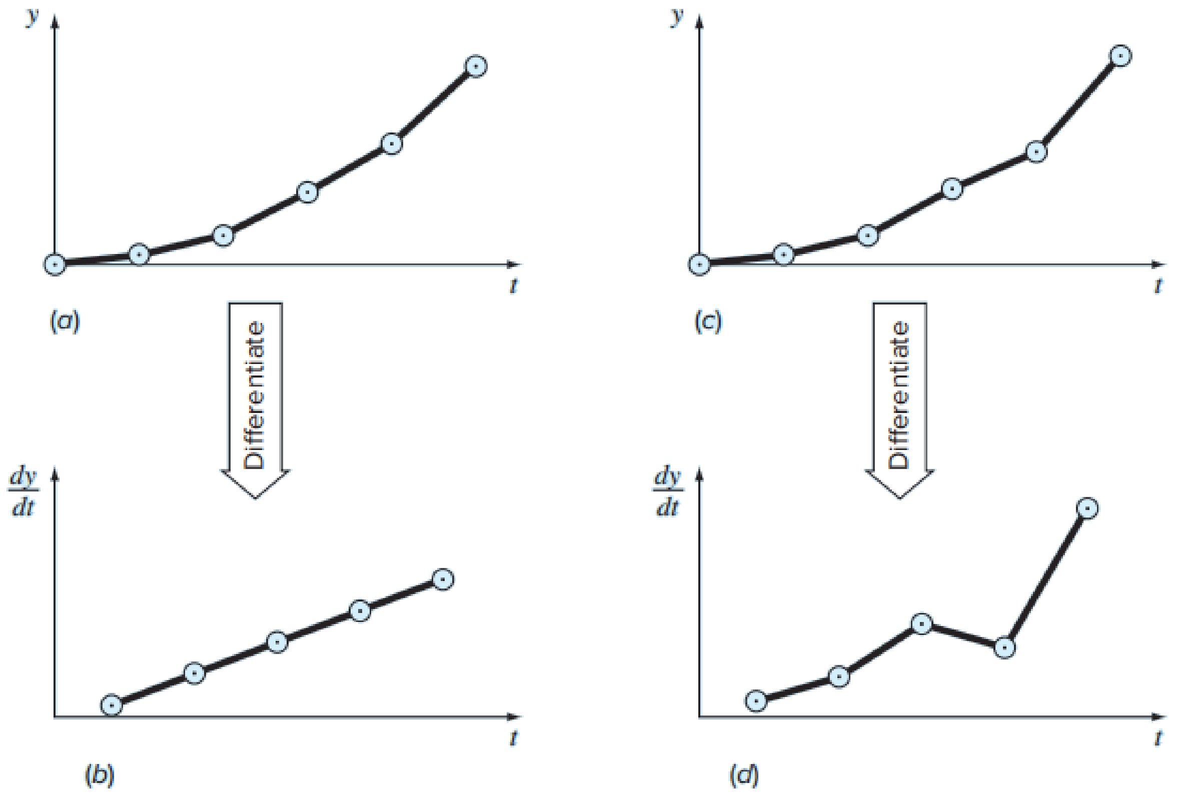
So we learnt how to approximate and find the derivative of function $f(x)$ at a certain point x_i. But, what if we want to find the derivative of the function $f(x)$ in general. In this case, we generate a vector x, and we approximate the derivative at every point x. Simply put, we call our function in a loop over each value x. Yet, it is imperative that the values in vector x must be evenly spaced in order to have an accurate result.

Derivatives of Unequally Spaced Empirical Data

The assumption we have used thus far in finite difference approximations of the first derivative is that the sample data points we measure, collect or generate in vector x are evenly spaced. This might not hold true under empirical circumstances. Suppose you are reading values from a sensor, and due to the jittering problem, the samples you get are not 100% sampled at exactly the same sample rate and that some points are delayed or others are sampled early. Also, suppose that in some experiments, you have missing data points. In these cases, you might not have the necessary data points to approximate the differential. In these applications, we need to guess the missing data through what we call interpolation before using it. We have already covered the interpolation in this course.

Derivatives for Data with Errors

Not only do we have to contend with uneven data samples that we collect from our experiments, sometimes the measurements we collect contain error. Suppose you are having a sensor that measures the speed of a moving vehicle (*e.g.*, car, train, airplane, *etc.*) and then you need to differentiate the collected speeds to measure the vehicle acceleration, if the data has errors, these will be amplified in the derivative. For example, in the figure below, if the collected measurements of the car speeds indeed reflect the speed accurately, then the acceleration will be smooth. However, if the sensor is not accurate enough, errors will be introduced in the measurement, and therefore, they will be highly visible in the incorrect acceleration derivative. In such situations, we often need to account for these errors by attempting to fit a smoother polynomial before calculating the acceleration.

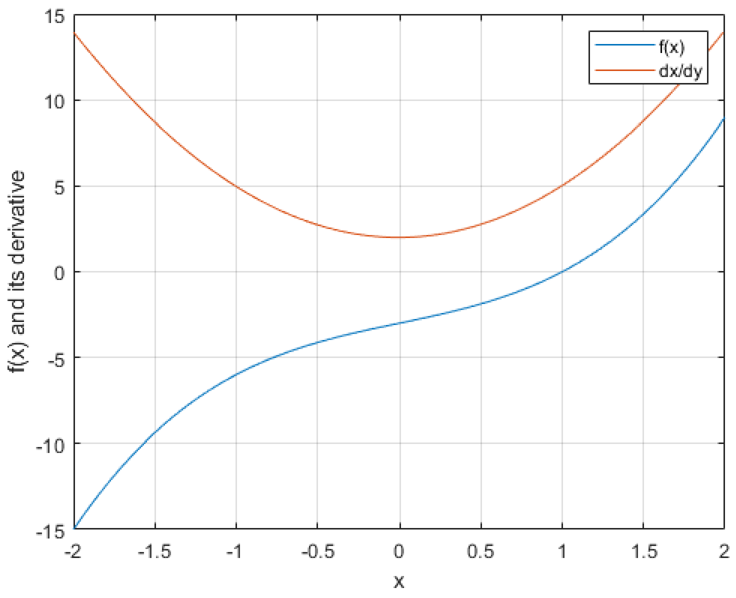


MATLAB Functions for Differentiation

MATLAB offers two built-in functions **`diff`** and **`gradient`** to determine the derivatives of data. When we pass a one-dimensional vector of length *n*, the **`diff`** function returns a vector of length *n* - 1 containing the differences between adjacent elements. Practically, this function computes the nominator of the fraction. The last step to do is to divide the values by the step size Δx. Kindly note that with **`diff`** we can differentiate an entire function at all points of x.

Suppose we want to differentiate and plot the function $f(x) = x^3 + 2x - 3$ alongside its derivative over the range [-2, 2].

```
 y = @(x) x.^3 + 2.*x - 3;  % Define the function as an anonymous function
 x = [-2:0.01:2];             % Create evenly spaced x points with even spacing
delta_x = 0.01;
 fx = y(x);                 % calculate the function f(x)
 dx = diff(fx);
 dx = dx / 0.01;
 plot (x, fx, x(1: length(x)-1), dx)
 grid on
 xlabel('x')
 ylabel('f(x) and its derivative')
 legend('f(x)', 'dx/dy')
```

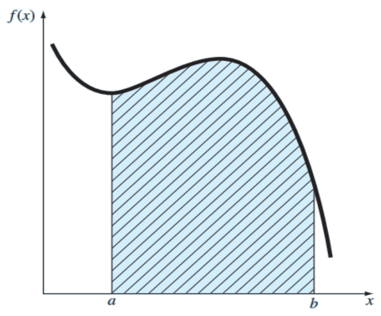


In comparison to the **diff** command, the **gradient** command uses central derivatives, thus giving the derivative at the point itself. You must pass Δx to the function directly.

```
 y = @(x) x.^3 + 2.*x - 3;  % Define the function as an anonymous function
 x = [-2:0.01:2];             % Create evenly spaced x points with even spacing
delta_x = 0.01;
 fx = y(x);                 % calculate the function f(x)
 dx = gradient(fx, 0.01);
```

Integration (Closed Form)

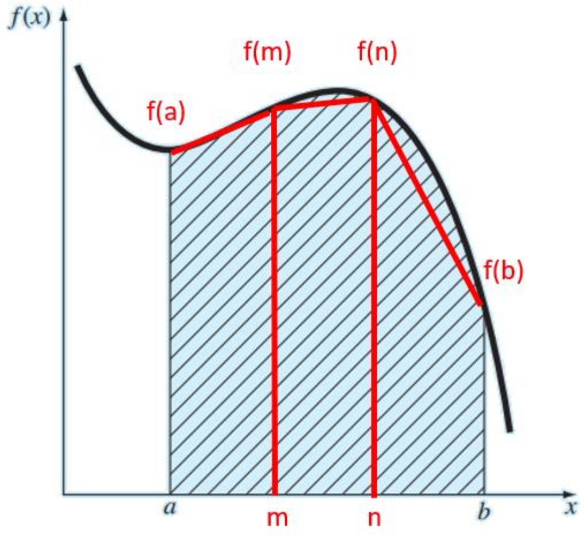
The closed form integration is defined as the area under the curve $f(x)$ from the points a to b as the figure below illustrates. Numerical integration is sometimes referred to as *quadrature*. This is an old name; most modern books use *numerical definite integration.* The most famous formulas for numerical definite integration are Newton-Cotes formulas. We will explain two of the formulas in this section: the Trapezoidal technique and one of Simpson's rules:



The Trapezoidal Rule

The main idea of the trapezoidal rule is to divide the function into multiple trapezoids. Since we know the equation that computes the area of a trapezoid, we can apply it, and sum all the areas of the trapezoids.

The first step is to divide the distance between a and b into **evenly spaced** multiple segments. In this simple example, we divided the range into three segments between the four points a, m,n, and b.The points a, $f(a)$, m, and $f(m)$ form a trapezoid. In general, remember that the function **`linspace`** is useful in this case, or the colon notation (*i.e.* 2 : 0.1 : 8) in order to generate the points x_i. Once we get this vector of x-values, we need to substitute it in the given function $f(x_i)$.

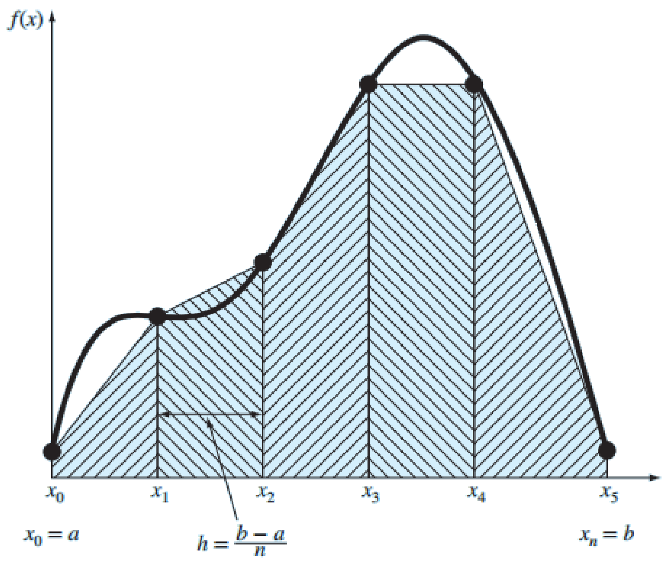


Then, we can compute the area of any trapezoid using the following equation:

$$area_s = (x_2 - x_1)\frac{f(x_2) + f(x_1)}{2} \tag{11}$$

So initially, we compute the area of the first trapezoid in the first iteration, then the second trapezoid in the second iteration, and so forth. The total summation approximates the area under the curve $f(x)$.

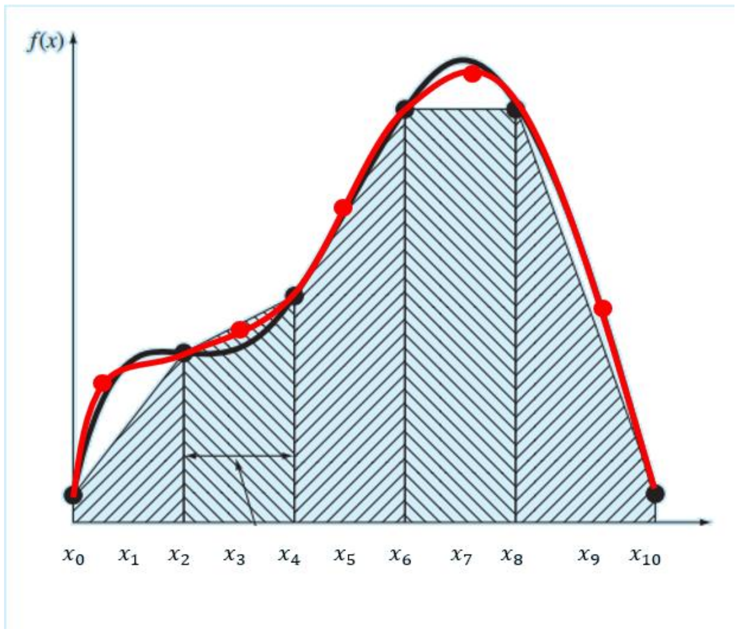
Note that the more segments you have (smaller Δx), the better your approximation gets.



Simpson's $\frac{1}{3}$ Rule

We have noticed that the trapezoidal rule successively splits the area under the curve $f(x)$ into multiple trapezoids. Notice that the trapezoid between points x_0 and x_1 in the figure above severely under approximates the actual area, while the trapezoid formed between x_1 and x_2 overly estimates the area. This is because the trapezoid is formed using linear segments, whereas the functions are not necessarily linear. One possible approach to mitigate this issue is to use smaller width trapezoids (decrease Δx, increase their numbers). However, more trapezoids entail more computations and time!

Simpson's $\frac{1}{3}$ rule is based on the idea of instead of connecting the upper segments using a line, it uses three points to draw a parabola shape instead. It assumes that parabolas better approximate non-linear functions. It uses three values in every interval to find a parabola shape that connects between them and therefore better approximate the original function $f(x)$. Note in the figure below how the new segments outlined by the red segment are now indeed a better approximation.


$f(x)$
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}

Now, the area for each segment can be computed as:

$$area_s = \frac{1}{3}\frac{\Delta x}{2}\left(f(x_0) + 4f(x_1) + f(x_2)\right) \tag{12}$$

Now suppose we have five segments, it is clear that in each segment we need three points to calculate the above equation. therefore, the number of x values we need to generate must be larger than the number of segments. Examining the above equation, we can see that we need $2n+1 \;\; x-$ points where n is the number of segments.

MATLAB Built-In Integration Functions

MATLAB has a built-in function that evaluates integrals for data based on the trapezoidal rule called **`trapz`**, the values of x must be sorted in ascending order:

```
x = [0 .1 .2 .3 .4 .5 .6 .7 .8];
y = 0.2+25*x-200*x.^2+675*x.^3-900*x.^4+400*x.^5;
trapz(x,y)
```

```
ans =
   1.594800960000011
```

MATLAB uses more accurate numerical methods (we did not cover it in this course) to do the integration using the **`integral`** command. For example, to integrate $f(x) = e^{(-x^2)}(ln(x))^2$, one can write:

```
y = @(x) exp(-x.^2).*log(x).^2;
q = integral(y,0,1)
```

```
q =
   1.933057085069212
```

The function can also take ∞ as a parameter:

```
q = integral(y,0,Inf)
```

```
q =
   1.947522220295560
```

For multiple integrals, MATLAB offers the **integral2**, and **integral3** commands (Check them - self study)

```
                                          Experiment version 1.1
                             Original Experiment December 17th, 2020
                                        Last Updated May 17th, 2022
                            Dr. Ashraf Suyyagh - All Rights Reserved
```

```
Version History

1.1
 - Removed the Optimization Section and made it into its own standalone
   experiment
 - Simplified the text language in many paragraphs.
 - Fixed the mathematical notation to match the figures
 - Removed the long formulae for the entire integration sum
```

MATLAB Commands List

You are free to check the MATLAB documentation for any of them during the exam time. Use the `doc` command followed by the command name

Chapter 5	Chapter 6	Chapter 7	Chapter 8/9
`min`	`poly`	`fzero`	`fminbnd`
`max`	`polyval`	`roots`	`fminsearch`
`mean`	`inv`	`poly`	`diff`
`median`	`polyfit`		`gradient`
`mode`	`interp1`		`trapz`
`std`	`interp2`		`integral`
`var`	`interp3`		`Integral2`
`kurtosis`	`interpn`		`Integral3`
`skewness`	`spline`		
`prctile`			
`iqr`			
`boxplot`			
`movsum`			
`hist/ histogram`			
`bar`			
`rand`			
`randi`			
`randn`			
`rng('shuffle')`			
`rng(0)`			
`randperm`			
`cumsum`			

Equations and Formulae

$E_t = True\ \ Value\ \ -\ \ Approximation$	
$\epsilon_t = \dfrac{True\ \ Value - Approximation}{True\ \ Value} \times 100\%$	
$\epsilon_a = \dfrac{Present\ \ Approximation - Previous\ \ Approximation}{Present\ \ Approximation} \times 100\%$	
$\epsilon_s = (0.5 \times 10^{2-n})\%$	
$x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$	

$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$	Newton-Raphson formula

$x_1 = x_{lower} + d$	Golden Section Optimization
$x_2 = x_{upper} - d$	
$d = (\phi - 1)(x_{upper} - x_{lower})$	
$\epsilon_a = (2\phi - 3)\left\lvert \frac{(x_{upper} - x_{lower})}{x_{opt}} \right\rvert \times 100\%$	
$\epsilon_a = (2 - \phi)\left\lvert \frac{(x_{upper} - x_{lower})}{x_{opt}} \right\rvert \times 100\%$	
$Slope \ = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x} = \frac{f(x_2) - f(x_1)}{x_2 - x_1}$	Slope
$f'(x) = \frac{\delta y}{\delta x} = \lim_{\Delta x \to 0} \frac{f(x_i + \Delta x) - f(x_i)}{\Delta x}$	Derivative
$f'(x) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta x}$	Forward difference order approximation
$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta x}$	Backward difference order approximation
$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x}$	Centred difference order approximation
$f'(x_i) = \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{x_{i+2} - x_i}$	Forward difference higher order approximation
$f'(x_i) = \frac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{x_i - x_{i-2}}$	Backward difference higher order approximation
$f'(x_i) = \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12\Delta x}$	Centred difference higher order approximation
$area_s = (x_2 - x_1)\frac{f(x_2) + f(x_1)}{2}$	The Trapezoidal Rule Segment Area
$area_s = \frac{1}{3}\frac{\Delta x}{2}(f(x_0) + 4f(x_1) + f(x_2))$	Simpson's 1/3 Rule Segment Area

Regression - Equations and Formulae

$y = a_1 x + a_0$	
$a_0 = \frac{\sum y_i}{n} - a_1 \frac{\sum x_i}{n} = \bar{y} - a_1 \bar{x}$	
$a_1 = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$	
$e = y - a_1 x - a_0$	
$S_r = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - a_1 x_i - a_0)^2$	
$S_t = \sum_{i=1}^{n} (y_i - \bar{y})^2$	
$r^2 = \frac{S_t - S_r}{S_t}$	Coefficient of Determination – Method 1
$r = \frac{n \sum (x_i y_i) - \sum x_i \sum y_i}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$	Coefficient of Determination – Method 2

The University of Jordan

School of Engineering

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Additional Exercises 1 - MATLAB Fundamentals I

`Exercises prepared by Dr. Ashraf E. Suyyagh`

Table of Contents

Exercise 1 - Equivalent MATLAB Expressions (10 Marks + 14 Marks)

Part 1 - For the given mathematical expressions, write down the equivalent MATLAB expression and the result.

Expression	MATLAB Result
$e^{-2.5} + 3.47ln(112)e^{-5^{2.4}}$	
MATLAB Expression:	
$sin^{-1}(cos^4(37°) + sin(25°))$ Write final answer in degrees	
MATLAB Expression:	
$4tan^{-1}\left(\frac{1}{5}\right) - tan^{-1}\left(\frac{1}{239}\right)$	
MATLAB Expression:	
Show if the left-hand side of Euler's Identity* equals the right-hand side: $e^{i\pi} + 1 = 0$	

MATLAB Expression:	
$\frac{log_4(8)log_9(282)}{log(70)ln(18)}$	
MATLAB Expression:	

* Euler identity is one of the most beautiful mathematical equations. It has the number 1, the basis of all numbers; the 0, the concept of nothingness; π the number that defines circles; e the number that defines exponential growth and decline; and finally, *i*, the imaginary square root.

Part 2 - Suppose that u = 5, v = -4, and w = 7.25. Use MATLAB to evaluate the following expressions.

Expression	**MATLAB Expression**	**Answer**
$\left(1-\frac{w}{(u-v)^{\frac{1}{3}}}\right)^{u-v}$		
$\left\|(5\pi\sqrt{v}-u^{-2})^{ln(w^2)}\right\|$		
$(8log_{10}^2(u^{1/w^v}))$		
$\angle(uv^{-\frac{1}{4}}-\frac{1.4}{u^5+6})$ The first symbol means angle		

$\frac{\sqrt{uv}\,\lvert u+wi\rvert}{log_{10}(v)-sin^2(u^2)}$		
$\frac{e^{5uv^2w}}{e^{10v^4}}$		
$\left\lvert\frac{1000}{e^{\sqrt{u}-\sqrt{v}}}\right\rvert \; and \; \angle\frac{1000}{e^{\sqrt{u}-\sqrt{v}}}$		

Exercise 2 - Infinite Square Roots Problem (6 Marks)

Suppose we have the following infinite square roots equation:

$$\sqrt{x+\sqrt{x+\sqrt{x+\sqrt{x+\cdots}}}}=4$$

Let us solve for *x* by hand.

Step 1: Take the square (x^2) for each side, so we get:

$$x+\sqrt{x+\sqrt{x+\sqrt{x+\sqrt{x+\cdots}}}}=16$$

Step 2: The new equation has the very same infinite square root term which we know that it equals 4, so:

$$x+4=16$$

Step 3: Solve for x, thus x = 12.

Now use MATLAB to check the answer. Given that this is an infinite square roots problem, you must use a limited number of square roots as illustrated in the table, but by doing so, the result will be close to 4 but not 4. Use MATLAB to fill the table.

How many square roots must we use such that the difference between the approximate result and the actual value is less than 0.5%? *Hint: The error percentage equation is given by:*

$$\frac{ApproximateResult-Result}{Result}\times 100$$

Approximation	Approximate Result	Difference Error Percentage (%)
$\sqrt{12+\sqrt{12}}$		
$\sqrt{12+\sqrt{12+\sqrt{12}}}$		
$\sqrt{12+\sqrt{12+\sqrt{12+\sqrt{12}}}}$		
$\sqrt{12+\sqrt{12+\sqrt{12+\sqrt{12+\sqrt{12}}}}}$		

Exercise 3 - Approximations of Pi (9 Marks)

Pi is one of the few numbers that has fascinated humans for thousands of years. Numerous mathematicians have come up with approximations for π.

1. Use MATLAB to find out the result of some of these approximations listed in the following table.
2. Calculate the percentage error between these approximations and the actual value of π using the percentage error difference equation:

$$\frac{Approximation - \pi}{\pi} \times 100$$

3. The 3rd and 4th approximations are accurate to the 30th and 18th decimal digits, respectively. That is, they are still not exact values of Pi. Why is your percentage error difference different than expected?

Hint: Change the output format to see more significant digits.

No.	***Approximation Formula***	***Approximation Result (1 Mark Each)***	***Percentage Error Difference (0.5 Marks Each)***
1	$3 + \frac{8}{60} + \frac{29}{60^2} + \frac{44}{60^3}$		
2	$\frac{9}{5} + \sqrt{\frac{9}{5}}$		
3	$\frac{ln(640320^3 + 744)}{\sqrt{163}}$		
4	$\frac{80\sqrt{15}(5^4 + 53\sqrt{89})^{\frac{3}{2}}}{3308(5^4 + 53\sqrt{89}) - 3\sqrt{89}}$		
5	$\sqrt[193]{\frac{10^{100}}{11222.11122}}$		
6	$\frac{63}{25} \times \frac{17 + 15\sqrt{5}}{7 + 15\sqrt{5}}$		

Exercise 4 – MATLAB Display Formats (1.5 Marks)

Suppose that *x* = *10* and *y* = *75.75*. Show the result of performing the operations when MATLAB display format is set to the options given in the table.

$$\frac{9x^{-0.2}}{y - y^{-2}}$$

Format	Answer
long	
short e	
long e	
bank	
rat	
+	

Exercise 6 – Finding MATLAB functions (1 Mark)

Use MATLAB function look up capability to find the functions which do the following:

Functionality	Function
Find the wavelet Fourier transform	
Decode JSON-formatted text	

The University of Jordan

School of Engineering

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Additional Exercises 2 - MATLAB Fundamentals II

Exercises prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Before starting to solve this lab sheet, please load these matrices into MATLAB by clicking on the provided *labsheet_02.mat* file or by inputting them manually (to practice). If you change these variables by accident, simply load the file again to restore the original values.

$$v_1 = \begin{bmatrix} 8 & 9 & -6 & 0 \end{bmatrix} \qquad v_2 = \begin{bmatrix} 7 & 4 & -3 & -2 \end{bmatrix}$$

$$v_3 = \begin{bmatrix} 12 \\ 55 \\ -53 \\ -11 \end{bmatrix} \qquad v_4 = \begin{bmatrix} 19 \\ -51 \\ 72 \\ 14 \end{bmatrix} \qquad v_5 = \begin{bmatrix} -97 \\ -13 \\ -27 \\ 49 \end{bmatrix}$$

$$v_6 = \begin{bmatrix} 12 & -19 & 32 \\ 14 & 33 & 56 \\ -7 & 41 & 4 \end{bmatrix} \qquad v_7 = \begin{bmatrix} -39 & 19 & 22 \\ -15 & 2 & 52 \\ 10 & 4 & 16 \end{bmatrix}$$

$$v_8 = \begin{bmatrix} 2 & 4 \\ 0 & 7 \\ 0 & 1 \end{bmatrix} \qquad v_9 = \begin{bmatrix} 7 & 4 & 1 \\ 5 & 3 & 1 \end{bmatrix}$$

$$v_{10} = \begin{bmatrix} 1 & -12 & 13 & 71 & 12 & 23 \\ 12 & 0 & 6 & 73 & 4 & -12 \\ 78 & 32 & 5 & 82 & 7 & 54 \\ 79 & 44 & -23 & -19 & 16 & 7 \\ 12 & 48 & -32 & -12 & 18 & -7 \\ 0 & 89 & 91 & 3 & -18 & 9 \end{bmatrix}$$

Exercise 1 – Basic Vector Operations (11 Marks)

Use MATLAB commands to find the output of the following operations. Provide a numeric answer, unless we ask for the command instead.

	Operation	Answer
1	Multiply v3 by v4 and then the result divide by v5 (elementwise)	*Numeric Answer*
2	What is the MATLAB expression (command) to append all vectors v1 through v5 vertically	*Command*
3	Multiply vectors v1 by v5 (normal array multiplication)	*Numeric Answer*
4	Multiply vectors v4 by v2 (normal array multiplication)	*Numeric Answer*
5	Append v1 and v2 together, save them in x. Then append v3 and v4 together, save them in y. How many elements are there in the vector y * x?	*Command (Numeric Answer for no of elements)*
6	By only using the colon operator, create a vector z that has the values from 1000 to -1000 with a spacing of 5	*Command*
7	Using MATLAB commands only, create a vector with 150 elements between 25 and 625? What is the spacing between elements?	*Command Numeric Answer for spacing*
8	Create a logarithmically spaced vector between 1,000,000,000 and 10 such that the elements of this vector are multiples of 10 in descending order	*Command*
9	Write the command that repeats each element in v3, 9 times and shows the result as a row vector (horizontally)	*Command*
10	Write the command that converts v5 to a 2x2 array	*Command*
11	Write the command that shifts v2 twice to the left	*Command*

Exercise 2 – Basic Matrix Operations (19 Marks)

Use MATLAB commands to find the output of the following operations. Provide a numeric answer, unless we ask for the command instead.

	Operation	Answer
1	Multiply (element wise) the natural logarithm of the absolute values of v6 by the log10 of the square of each element in v7?	*Numeric Answer*
2	What is v8*v9 – v6 + transposed v7?	*Numeric Answer*
3	What is the command to multiply the ***LEFT*** diagonal of v6 by the ***LEFT*** diagonal of v7?	
4	What is the command to retrieve all the negative elements in v10?	
5	What is the command to retrieve the indices of all occurrences of -12 in v10?	
6	Write the command(s) that count all numbers between 70 and 100 in v10?	
7	Write the command to sort the rows of v10 in descending order and save them into a new matrix?	
8	Write the command(s) to extract the ***RIGHT*** diagonal of v10?	
9	Write the command to change all negative numbers in v6 to 0	
10	Write the command to concatenate v6 and v7 vertically, then change the shape of the resulting matrix to be 2x9	
11	Write the command to extract the middle row in v6?	
12	Write the command to extract the middle column in v7?	
13	Write the command(s) to sum all the elements in v10	
14	Write the command(s) to replace the elements $\begin{matrix} 44 & -23 & -19 \\ 48 & -32 & -12 \end{matrix}$ in v10 by v9	
15	Write the command(s) that multiples all elements in a magic cube of size 3	
16	Write the command(s) that show how a magic cube of size 4 actually has all its columns and rows sum to the same number	
17	Write the command that rotates v8 by 180	
18	Write the command that flips v8 vertically	
19	Write the command that creates and initializes an 8x8 matrix to zero?	

Exercise 3 – Fun with Numbers I (7 Marks)

Starting with only the command ***ones(2)***, write the sequence of commands to create the following matrix:

$$\begin{bmatrix}0&0&0&0&0&0&0&0&1&1&1&1\\0&0&0&0&0&0&0&0&1&1&1&1\\0&0&0&0&0&0&0&0&1&1&1&1\\0&0&0&0&0&0&0&0&1&1&1&1\\0&0&0&0&2&2&2&2&0&0&0&0\\0&0&0&0&2&2&2&2&0&0&0&0\\0&0&0&0&2&2&2&2&0&0&0&0\\0&0&0&0&2&2&2&2&0&0&0&0\\4&4&4&4&0&0&0&0&0&0&0&0\\4&4&4&4&0&0&0&0&0&0&0&0\\4&4&4&4&0&0&0&0&0&0&0&0\\4&4&4&4&0&0&0&0&0&0&0&0\end{bmatrix}$$

Exercise 4 – Fun with Numbers II (4 Marks)

Starting with only the command ***eye(4)***, write the sequence of commands to create the following matrix:

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

Exercise 5 – Multidimensional Matrices (4 Marks)

Inside the file **labsheet_02.mat**, the variable v11 has a 3D array of 3 slices, and each slice is 4x4.

	Operation	Answer
1	Extract the elements in the third slice of v11	
2	Extract the first column in the first slice of v11	
3	Using linear indexing, retrieve the last element in v11	
4	How many zero-valued elements are there in v11?	

The University of Jordan

School of Engineering

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Additional Exercises 3 – Scripts, Functions and Control Flow

Exercises prepared by Dr. Ashraf E. Suyyagh

Table of Contents

Exercise 1 – MATLAB Functions

You are required to write a MATLAB primary function that accepts as an input two variables. The function arguments are:

- *myVec*: a 4-element positive numeric and integer vector whose values are larger than 10
- *myArray*: a numeric array with exactly 4 columns, should only contain real integer numbers

You must use **function argument validation** to check if *myVec* and *myArray* satisfy the requirements.

If all requirements are met, then the function asks the user for a third variable *myTask* to input using the command window. The variable *myTask* will have string values only.

- If *myTask* has the value '**1'**, the code will multiply *myVec* by *myArray* rows on an element-by-element basis.
- If *myTask* has the value '**2'**, the code will count how many instances each element in *myVec* appears in *myArr*
- If *myTask* has the value '**3**, the code will perform array multiplication between *myVec* and *myArr*. You might need to transpose *myVec* in some cases to make sure the multiplication is legal.

Each one of the above three tasks must be implemented as a **subfunction** that your primary function calls.

Finally, the primary function must return the result to the user in the variable ***result*** and display to the user the message: "This function has been called x times". It is your responsibility to keep track of x.

Exercise 2 – Spring is Here: Love and Flowers

Write a MATLAB script which asks the user to input either one of these two values: '*1* or '2.

1. If the user inputs '1, you must plot the following equation on the x-y coordinate system:

$$x = t \times sin(0.872\pi\frac{sin(t)}{t})$$
$$y = -\,|t| \times cos(\pi\frac{sin(t)}{t})$$

   where *t* has at least 350 values between $-\pi$ and π

2. If the user inputs '2, then the program must plot the following polar equation:

$$r = 3 + 8cos(8\theta)$$

   and θ has 1000 values between 0 and 10.

   Hint: Use plot(x, y) to plot on the Cartesian coordinate system, and polarplot(theta, r) to plot on the polar coordinate system.

Exercise 3 – Geometric Series

A geometric series is defined as the sequence $1, x, x^2, x^3$ ... x^m ,in which the powers of x range from 0 to ∞.

If $|x| < 1$, then the series will converge to $\frac{1}{1-x}$. Otherwise, the series will diverge towards ∞

a) For $x = 0.329$, compute the value the series will converge to.

b) Define an anonymous function that implements the above geometric sequence.
Hint: The function will have two arguments, x and n, where n is a vector of the powers used.

c) Generate and sum the values for the first 5 elements of the geometric series by calling the anonymous function for x = 0.329. Repeat for the first 50 elements, then 100 elements.

The University of Jordan

School of Engineering

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Exercises Set 4 – MATLAB Plots

Exercises prepared by Dr. Ashraf E. Suyyagh

Table of Contents

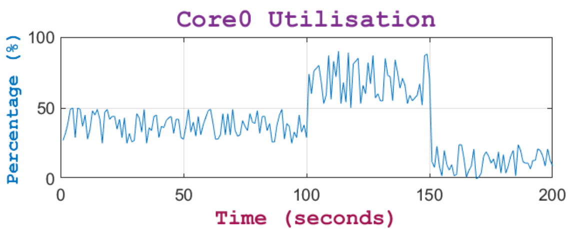
Exercise 1 – Computer Performance Profiling

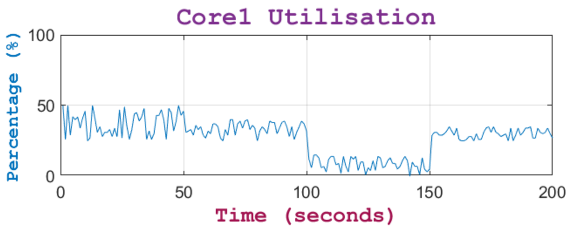
The file *'profiling.mat'* consists of an array of eight columns. Eight column is a performance log for a computer system component recorded over the period of 200 seconds at a rate of 1 second (200 samples). The file contains data for these system components in order:

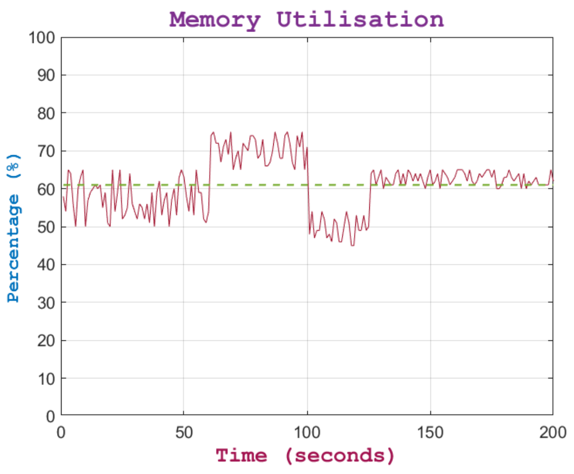
Core0, Core1, Core2, Core3, Memory, HDD0, HDD1, SSD

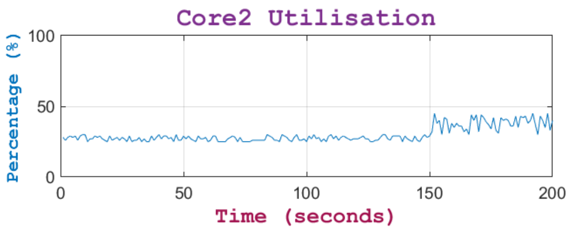
You are required to write a MATLAB script that loads the file *'profiling.mat'* and visualizes its data in **one figure EXACTLY** as shown in Figure 1. The specifications to recreate the Figure are as follows:

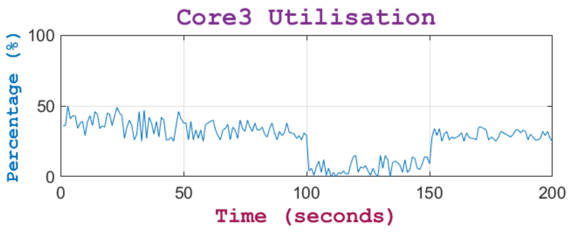
1. Use tiled formatting with compact spacing in between tiles.
2. The figure size must be 1024 x 768 pixels and starts 100 pixels from the left bottom corner of the screen.
3. The main figure and subfigures titles are bold, and in fixed width font.
4. All figure titles must appear exactly as shown.
5. The main figure title size is 18, the subfigures title size is 14, the x-axis labels are size 12, and y-axis labels are size 10.
6. The grid must be visible for all graphs.
7. The x-axis range must show the whole range from 0 to 200, and the y-axis must have the complete range from 0 to 100. The 0's must appear on the graph.
8. For the memory plot, you must also compute the average memory utilization and plot it in dashed line format.
9. For the disk access plot, you must show a legend for the three disks and let MATLAB choose the default coloring for the three plots.
10. You must save the figure as a '.bmp image at the end.
11. The y-axis label color is (#0072bd), the x-axis label color is (#a2124f), the titles color is (#7E2F8E), and the average memory plot color is (#77ac30)

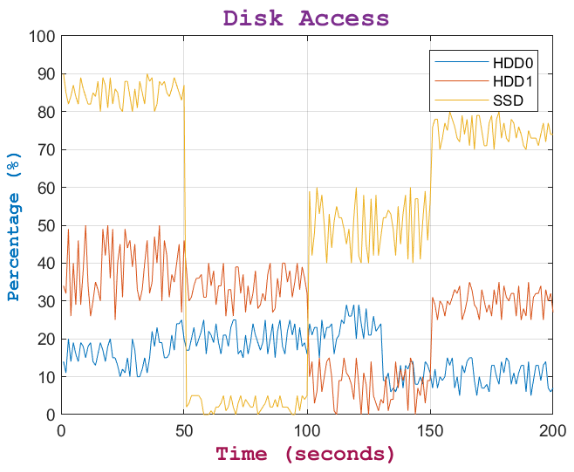

Core0 Utilisation
Percentage (%)
Time (seconds)


Core1 Utilisation
Percentage (%)
Time (seconds)


Memory Utilisation
Percentage (%)
Time (seconds)


Core2 Utilisation
Percentage (%)
Time (seconds)

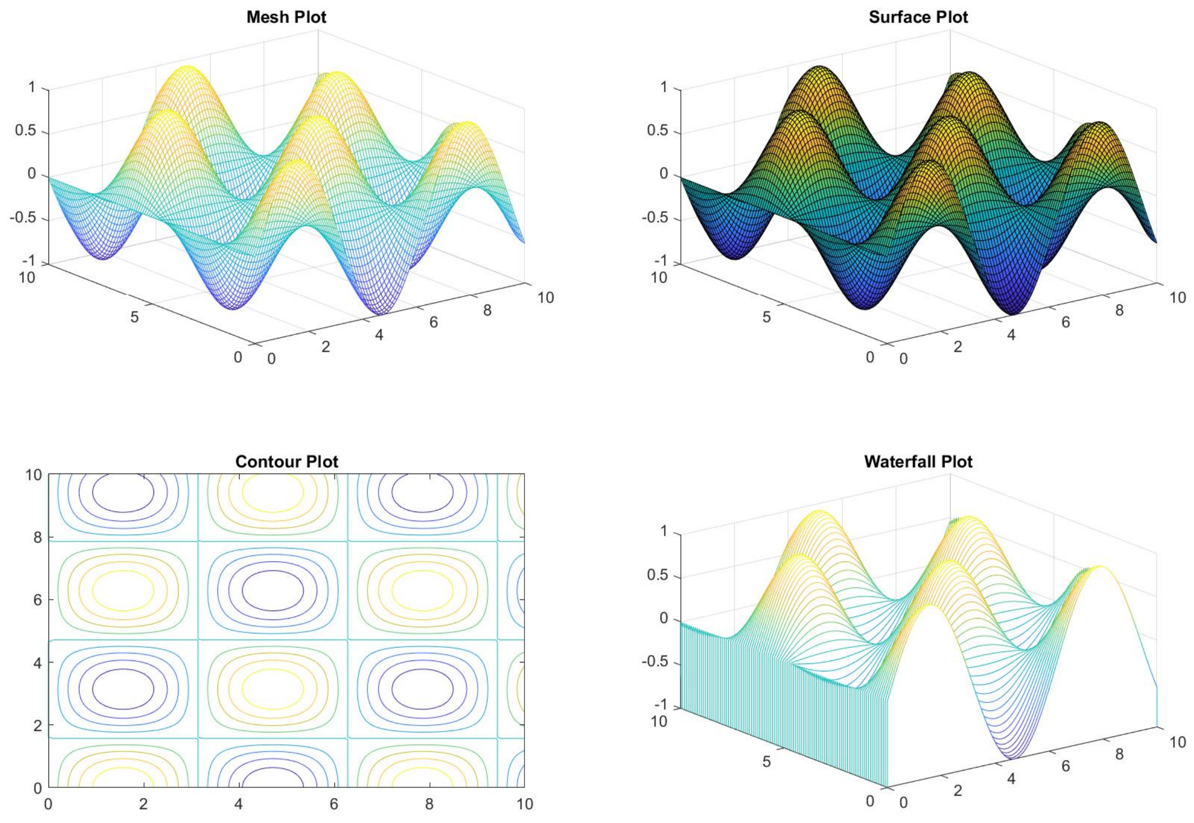

Core3 Utilisation
Percentage (%)
Time (seconds)


Disk Access
HDD0
HDD1
SSD
Percentage (%)
Time (seconds)

Exercise 2 – Basic 3D Plots

You are required to write a MATLAB script that visualizes its data in **<u>one figure</u>** **<u>EXACTLY</u>** as shown below.

1. Use tiled formatting with normal spacing in between tiles.
2. The underlying function is: $f(x, y) = \sin(x)\cos(y)$.
3. You need to plot the figure extending from the range 0 to 10, each axis must have 100 points.
4. The figure size must be 1200 x 800 pixels
5. Create a subtitle for each plot
6. Save the plot as a pdf image



.

The University of Jordan

School of Engineering

Department of Computer Engineering

Practical Numerical Analysis (CPE313)

Exercise Set 5 – Statistics and Probability

`Exercises prepared by Dr. Ashraf E. Suyyagh`

Table of Contents

Exercise 1 – Task Execution Time Analysis

In computer systems in general, and especially in real-time, control, and embedded systems, we take care of measuring the execution time of system tasks and understanding their behavior. For example, we do not want the execution time of a system task to exceed its deadline, so we need to improve and shorten its time, if possible. On one system, the same task can have different execution times. There are many reasons for this; if its data is already allocated in the cache, it will execute faster, if not, it will be slower trying to fetch the data from the main memory into the cache memory. Yet, the most important reason is that tasks have loops and if-statements; depending on the input variables, some loops will execute longer or shorter, and some if-statements will execute the if-part, others the else-part. Therefore, we can expect varying execution paths and therefore execution times for each task.

Measuring the execution time and other metrics is called task profiling, and the output logs are called traces. The file *'taskTraces.mat'* consists of an array of three columns. Each column contains the times (in *ms*) it took to execute tasks τ_1 , τ_2, and τ_3 **100,000** times. Load the file *'taskTraces.mat'* into MATLAB and answer the following questions:

No.	Question	Answer			
1	What is the minimum and maximum execution times for each task?		τ_1	τ_2	τ_3
		minimum			
		maximum			
2	a) What is the skewness of the execution time distribution of each task? b) Which task execution times distribution is normally distributed? c) What about the other tasks? Are they lightly, moderately, or heavily skewed? d) Where do you expect the majority of the execution times to reside; at the left, center, or right of the distribution graph?		τ_1	τ_2	τ_3
		skewness			
			Description (a, b, c)		
		τ_1			
		τ_2			
		τ_3			

3	Which command would you use to find the central tendency of each execution time distribution?		Command	Result
		τ_1		
		τ_2		
		τ_3		
4	a) Which of the three distributions do you think you can compute the standard deviation and variance for and have valuable information? b) What is the result of the standard deviation or variance for this/these distribution?			
5	Classify the three execution times distributions into mesokurtic, leptokurtic, platykurtic.		Type	
		τ_1		
		τ_2		
		τ_3		
6	Based on the excess kurtosis value, which one of these three distributions have task execution times that are more extreme and away from the other time samples?			

7	Compute the three quartile points and the interquartile range for each of these three task execution time distributions?		Q_1	Q_2	Q_3	IQR
		τ_1				
		τ_2				
		τ_3				

8	Draw the boxplots for each of the execution times distributions on individual plots and copy/paste them in their specified location to the right. Briefly explain what you understand from each plot **<u>regarding the execution times themselves.</u>**	Plot 1
		Explanation:
		Plot 2

		Explanation:
		Plot 3
		Explanation:
9	Draw a proper histogram for each of the three task distributions. The bin width for τ_1 must be 10 *ms*, for τ_2 must be 5 *ms*, and for $\tau_3$it is 1 *ms*. Copy/paste your graphs in the designated space to the right	Plot 1
		Plot 2

		Plot 3
10	What is the probability that the execution time for task τ_1 is less than or equal 500 ms?	
	What is the probability that the execution time for task τ_3 is more than 8.5 ms?	