Islamic University of Technology (IUT)
Organization of Islamic Cooperation (OIC)
Department of Electrical and Electronic Engineering (EEE)

COURSE NO	:	EEE 4416
LAB NO	:	05 (Part A)
TOPIC	:	FUNCTIONS

FUNCTIONS

Functions are blocks of code that perform specific tasks. MATLAB offers a wide range of built-in functions. These functions make coding in MATLAB faster and easier. They eliminate the need to write lengthy or repetitive code for common operations such as finding the maximum, minimum, sum, or mean of an array. Instead of writing long loops to do such simple tasks, you can just use functions like max() or min(). These functions save time, reduce errors, and make your code cleaner and easier to understand.

These functions are already available in the MATLAB package. For instance, the 'sort' function allows you to sort an array without implementing any sorting algorithms like Quick sort, merge sort, or insertion sort (these are done internally inside the built-in function).

You have already been introduced to several useful built-in functions. To quickly get an idea of how to work with a built-in function, you should look into the MATLAB documentation. Let's take a look at that and understand more clearly how to apply these functions efficiently.

Configurations

Depending on the number of input and output arguments, functions can have four different configurations. They are as follows –

- SISO – single input, single output

```
y = sqrt(25)            % Output: y = 5
```

- SIMO - Single Input, Multiple Outputs

```
[rows, cols] = size(A)
```

- MISO - Multiple Inputs, Single Output

```
c = hypot(3,4)          %returns 25
```

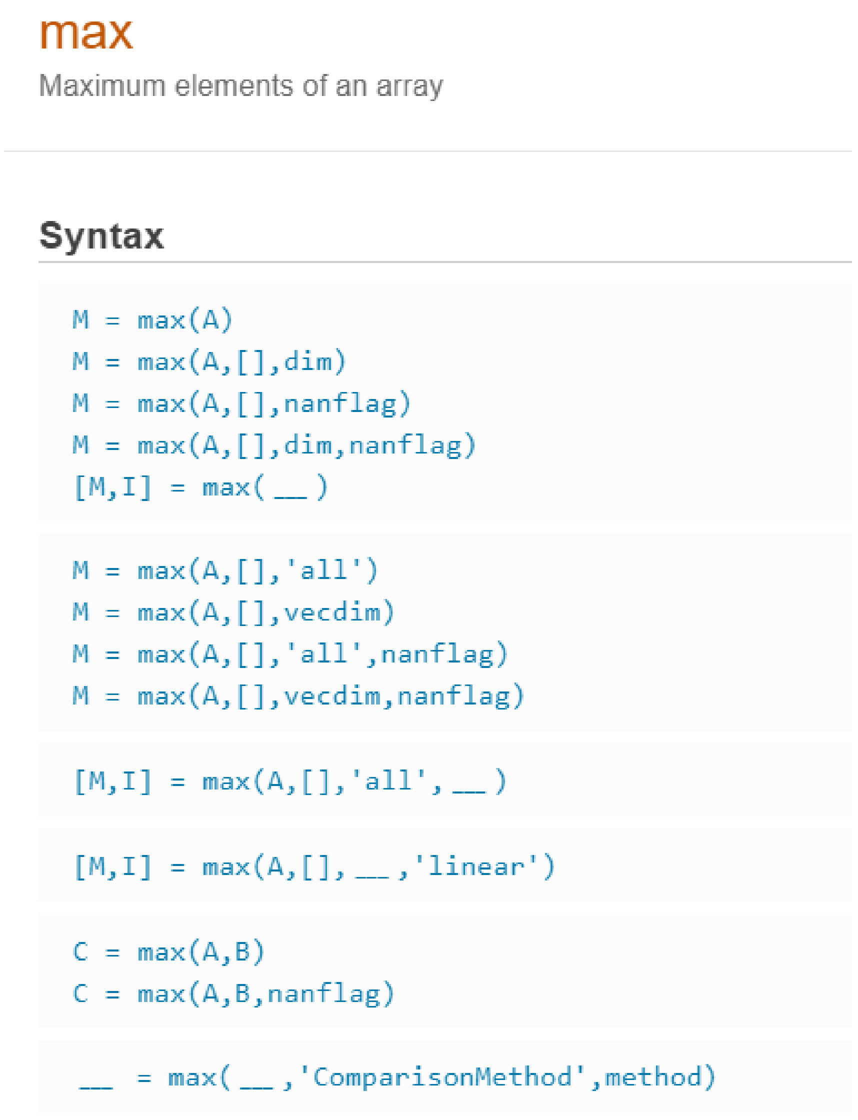
- MIMO - Multiple Inputs, Multiple Outputs

```
[s, idx] = sort(a, ‘descend’)
```

Variable length input arguments

The only configuration possible for the ‘sqrt’ function is SISO. However, if you look at the documentation of the ‘max’ function, you can see it can work as all four types - SISO, SIMO, MISO, and MIMO.

Some functions can have one, two, or more input arguments. This is called variable-length input arguments. It allows for better flexibility in operation. Similarly, functions can have one, two, or more output arguments as well. Let’s solve exercise – 01 to better understand this concept.



max

Maximum elements of an array

Syntax

```
M = max(A)
M = max(A,[],dim)
M = max(A,[],nanflag)
M = max(A,[],dim,nanflag)
[M,I] = max( __ )

M = max(A,[],'all')
M = max(A,[],vecdim)
M = max(A,[],'all',nanflag)
M = max(A,[],vecdim,nanflag)

[M,I] = max(A,[],'all', __ )

[M,I] = max(A,[], __ ,'linear')

C = max(A,B)
C = max(A,B,nanflag)

 __ = max( __ ,'ComparisonMethod',method)
```

USER-DEFINED FUNCTION

While MATLAB offers a wide variety of built-in functions, there is always a need to perform some tasks for which MATLAB does not have any functions built in. In such cases, MATLAB provides the option to create your own function to perform a certain task and then reuse it again just like a built-in function in any other script. These are called user-defined functions.

You need to be a bit careful while creating such functions and maintain certain rules to use them properly.

- You have previously used the *primes(n)* function. It allows you to create an array containing prime numbers less than or equal to the input value n.

```
p = primes (25)
p = 1×9

     2     3     5     7    11    13    17    19    23
```

- Now, let's say, we want a list of triangular numbers. **Triangular numbers** are a type of **figurate number** that represents a triangle formed by arranging dots in an equilateral triangular pattern.

Formula: The n-th triangular number is:

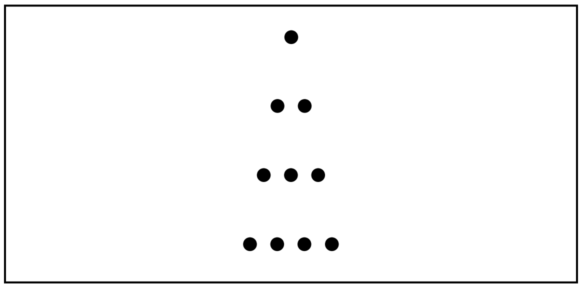
$$T_n = \frac{n\,(n+1)}{2}$$

Sequence:

1, 3, 6, 10, 15, 21, 28, 36, 45, 55, …

Explanation: The n-th triangular number is the total number of dots that can form a triangle with n rows. For example:

- T1=1: a single dot.
- T2=3: a triangle of 2 rows (1 + 2).
- T3=6: a triangle of 3 rows (1 + 2 + 3).
- And so on.



Now, MATLAB does not have any built-in function to extract triangular numbers. So, we want to create a function of our own that will return triangular numbers just like the primes() function.

Instructions

You need to maintain the following instructions carefully when creating a user-defined function.

- First, you need to choose a name for your user-defined function. This name cannot be the same as any built-in functions that are provided by MATLAB.
- Try to make the name representative of the task that you are trying to perform.
- Don't create any variable with a name the same as that of your user-defined function.
- You should create a user-defined function in a **separate script**. The script must not contain any other code except for your function.
- A single script should contain only one user-defined function.
- Save the script. The **filename** for the script must be **exactly the same** as your function name.
- Keep track of your directory. Your script (function) can only be reused just as a built-in function as long the function is in the **same directory**.
- After you have created and saved your user-defined function, you can test it in the command window. Don't test it in the same script.

Structure

```
function output_variable_name = function_name (input_variable_name)

        … …

        … …

        … …

end
```

- The reserved keyword to create a user-defined function is 'function'. It should be completed with an 'end', just like a loop or conditionals.
- You need to give your function a name which should be unique. It must start with a letter. Can contain letters, digits, and underscores (_). **No spaces, dashes (-), or special characters.**
- You can have one or more input variables. For more than one input variable, separate them using a comma.
- You can also have one or more output variables. For more than one output variable, separate them using a comma and place them inside a square bracket [ ].
- Inside the function, you should write the relation between output and input variables.
- You can create separate variables inside, as many as needed.
- All these variables are **local variables**. They are only available inside your function. Cannot be used outside.
- Similarly, you **cannot use global variables** inside your user-defined function.

Example – 01

Let's start by creating a simple user-defined function that will return the product of two inputs.

When building your user-defined, you need to first fix the following things –

i. What are the input arguments
ii. What are the output arguments
iii. Function name

For this problem,

i. There are two input arguments which are two numbers. Let's name them as 'a' and 'b'.
ii. There is only one output argument. Let's name it as 'out'.
iii. Let's fix the function name as 'product'. You cannot use 'prod' since it's a built-in function.

```
function out = product(a,b)

 out = a*b;

end
```

The statement in the middle line provides the relation between the input and output parameters. Bear in mind, here, 'a', 'b', and 'out' - all are local variables. They don't have any exact value. They are only present to provide the relation.

To use this function, save it first by setting the filename as 'product.m'. Then in your command window, write –

```
⇨ product (10, 5)          % this will return 50
⇨ product (2, 0)           % this will return 0
⇨ product (10)             % this will return an error
```

Here, 10 and 5 are your input variables. They are internally assigned as a =10 and b=5.

Since your defined function has two input arguments, you have to provide exactly two inputs while calling the function.

Class Task: Create a function called 'upper_first_last' that takes a string as input. It returns the same string as output with all the characters converted to lowercase, except the first and the last one.

- 'Bruce' => 'BrucE'
- 'wayne' => 'WaynE'
- 'ALBUS' => 'AlbuS'

Example – 02

Let's create a user-defined function similar to the 'primes' function of MATLAB. It will return the triangular numbers up to 'r'.

For this problem,

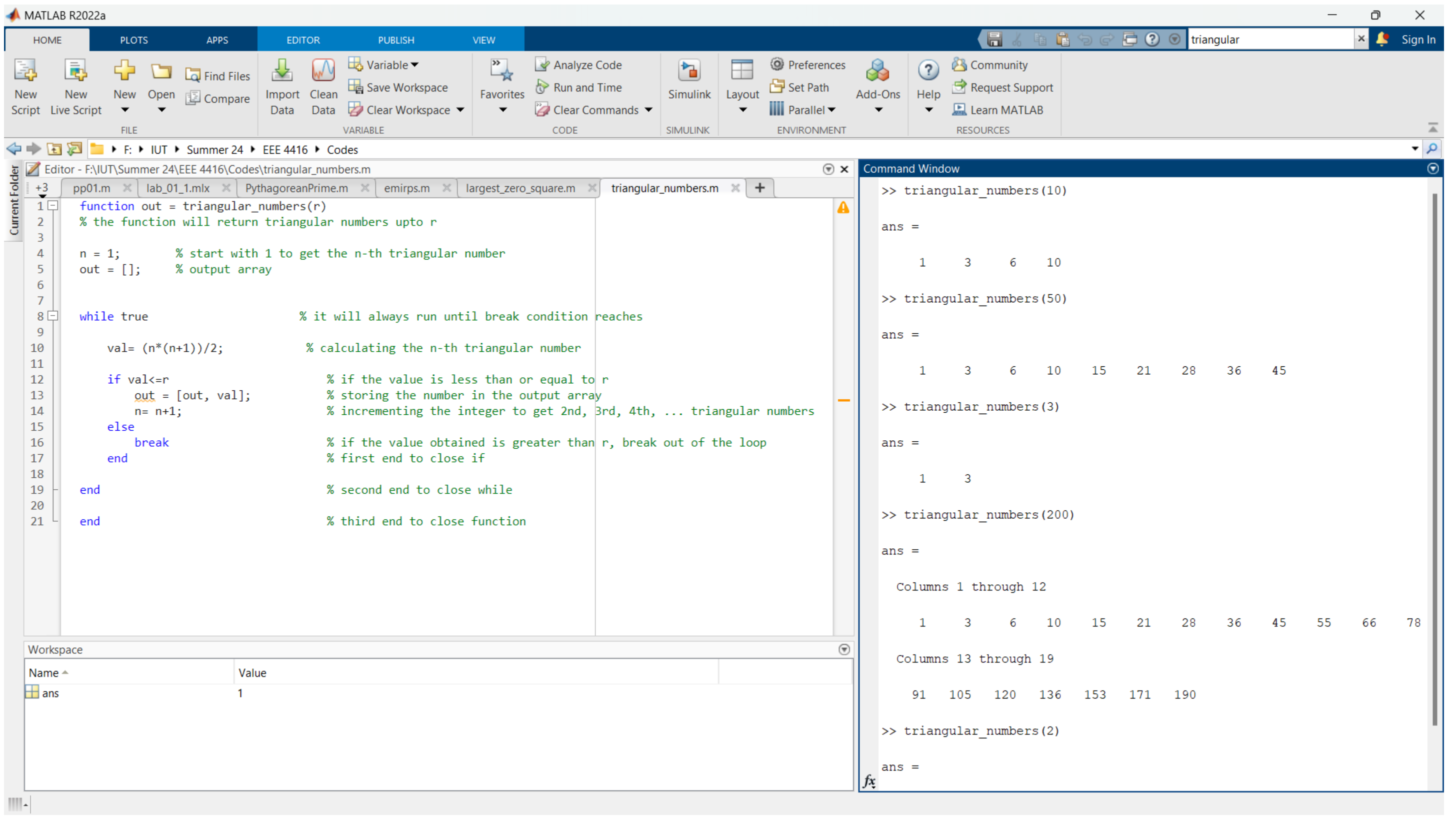
i. The input argument is an integer (variable name – 'r') up to which the function will return triangular numbers.
ii. The output argument is a list (variable name – 'out') that will contain triangular numbers up to 'r'.
iii. The function name is taken as – 'triangular_numbers'

Method –

- Keep generating triangular numbers and store them in an array until the limit 'r' is reached.

```
function out = triangular_numbers(r)
% the function will return triangular numbers upto r

n = 1;        % start with 1 to get the n-th triangular number
out = [];     % output array


while true                      % it will always run until break condition reaches

    val= (n*(n+1))/2;           % calculating the n-th triangular number

    if val<=r                   % if the value is less than or equal to r
        out = [out, val];       % storing the number in the output array
        n= n+1;                 % incrementing the integer to get 2nd, ... triangular numbers
    else
        break                  % if the value obtained is greater than r, break out of the loop
    end                        % first end to close if

end                            % second end to close while

end                            % third end to close function
```

- Save your file with the same name as your function and in the same directory.
- Then you can test the function in your command window or in a separate live script.



```matlab
function out = triangular_numbers(r)
% the function will return triangular numbers upto r

n = 1;        % start with 1 to get the n-th triangular number
out = [];     % output array


while true                       % it will always run until break condition reaches

    val= (n*(n+1))/2;             % calculating the n-th triangular number

    if val<=r                        % if the value is less than or equal to r
        out = [out, val];            % storing the number in the output array
        n= n+1;                      % incrementing the integer to get 2nd, 3rd, 4th, ... triangular numbers
    else
        break                        % if the value obtained is greater than r, break out of the loop
    end                              % first end to close if

end                                  % second end to close while

end                                  % third end to close function
```

Name	Value
ans	1

```
>> triangular_numbers(10)

ans =

     1     3     6    10

>> triangular_numbers(50)

ans =

     1     3     6    10    15    21    28    36    45

>> triangular_numbers(3)

ans =

     1     3

>> triangular_numbers(200)

ans =

  Columns 1 through 12

     1     3     6    10    15    21    28    36    45    55    66    78

  Columns 13 through 19

    91   105   120   136   153   171   190

>> triangular_numbers(2)

ans =
```

Asif Newaz
Lecturer, EEE, IUT

Example – 03

Every positive integer has a unique prime factorization. For example,

- 8 = 2 * 2 * 2
- 12 = 2 * 2 * 3

Here, (2,2,2) or (2,2,3) are called the prime factors of the integer 8 and 12 consecutively. There is a built-in function already provided in MATLAB to do this task for us.

⇨ *factor(n)*

Now, say, I want to find all the divisors of a number (excluding the number itself). For example,

- 8 = 1, 2, 4
- 12 = 1, 2, 3, 4, 6

Here, 12 is divisible by 1, 2, 3, 4, 6.

Unfortunately, MATLAB doesn't provide a built-in function to obtain that. So, we'll try to write our own function. Later in other codes, you'll be able to re-use your defined function just as the built-in functions that you've already created. That will prove a great advantage in many programming exercises later.

Proper divisors of a positive integer n are all its positive divisors excluding n itself. Let's try and create a user-defined function called 'proper_divisors' that will take an integer as input and return its proper divisors as output in an array.

Method

- You need to check whether the integer n is divisible by numbers 1, 2, 3, … so on.
- You can use the modulo operator to check the remainder.
- If *mod (n, 5) == 0*, then n is divisible by 5.
- You can use a for loop to check each number one by one. A better approach is to use vectorization. You can directly perform the modulo operation on an array. The following code will return a logical array whether n is divisible by numbers 1 to 10 or not.
  - *mod (n, 1:10) == 0*
- The highest number that can be a proper divisor of n is n/2. That is, 5 is the highest divisor of 10. So, you can narrow your search using this concept. It will reduce the runtime of your code.

```
function out = proper_divisors(n)

a= 1:n/2;
b= mod(n,a)==0;
out= a(b);

end
```