Islamic University of Technology (IUT)

Organization of Islamic Cooperation (OIC)

Department of Electrical and Electronic Engineering (EEE)

COURSE NO	:	EEE 4416
LAB NO	:	06 (Part A)
TOPIC	:	VARIABLE LENGTH USER-DEFINED FUNCTIONS

USER-DEFINED FUNCTIONS

In Lab 05, you were introduced to functions and learned how to create custom functions in MATLAB. In this lab, we will explore this concept further and delve deeper into function design and usage.

- A user-defined function is typically written in its own .m file, where the file name must match the function name.
- You can call the function from the command window or another script.
- The variables defined inside the functions are local and only exist within the script.
- You can define a function with multiple inputs and outputs.

Let's look at a couple of examples.

```
function area = circular_area(radius)

   % This function calculates the area of a circle given the radius

   area = pi * radius^2;

end
```

```
function [sum, product] = sum_product(a, b)

   sum = a + b;

   product = a * b;

end
```

A function provides a quicker and more efficient way to test the relationship between inputs and outputs. From now on, we will usually rely on such functions to write our programs.

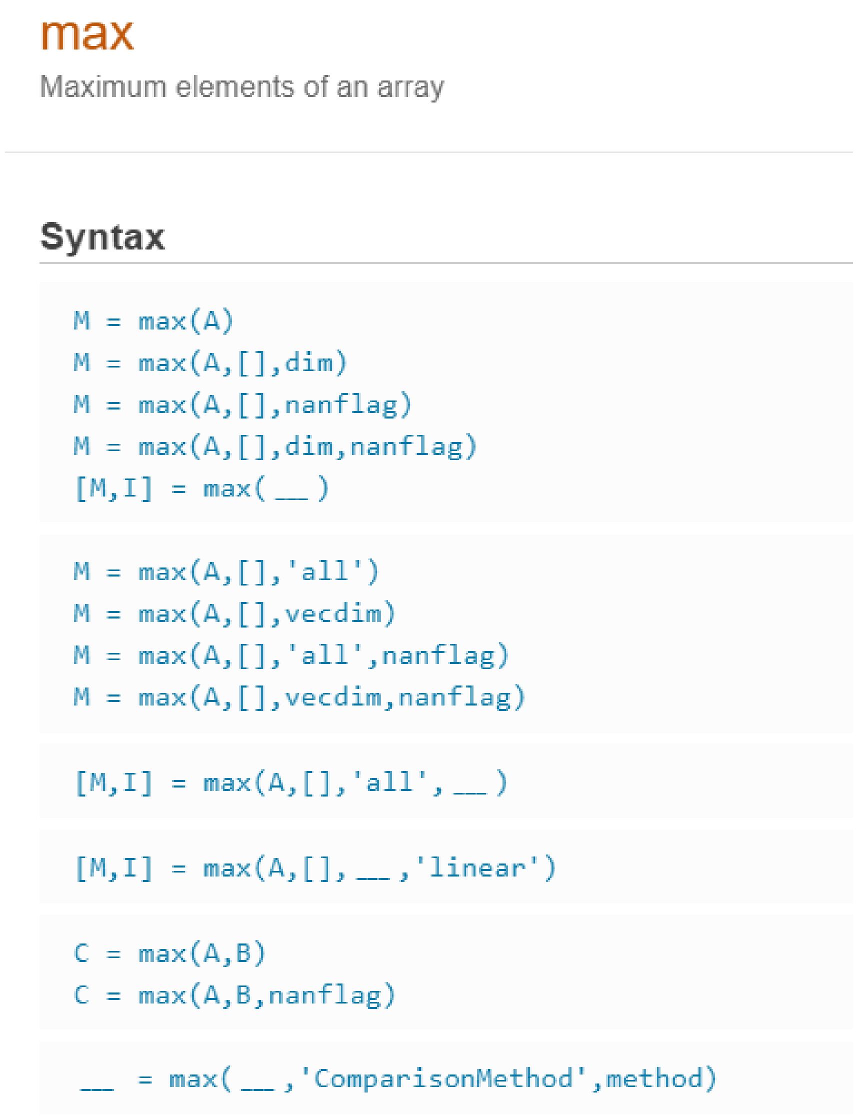
Asif Newaz
Lecturer, EEE, IUT

Variable-Length Arguments

In MATLAB, variable-length input arguments allow you to create functions that can accept a flexible number of inputs. This is useful when the exact number of arguments may vary depending on how the function is called. It allows you to perform a wide range of operations.

You have already observed this behavior when using built-in functions such as max(x).

- It provides the maximum of the array x.
- But if x is a matrix, the same function by default performs column-wise max operation.
- Moreover, it also takes addition input like max(x, [ ], 2) to perform row-wise max operation. It performs column-wise if max(x, [ ], 1) is provided – which is set to default.
- It takes an additional 3rd argument and performs the max operation on the whole matrix if max(x, [ ], 'all') is given as input.
- There are other options such as 'nanflag' to handle nan values.
- It can also provide multiple outputs – the highest value as well as the associated index.



max

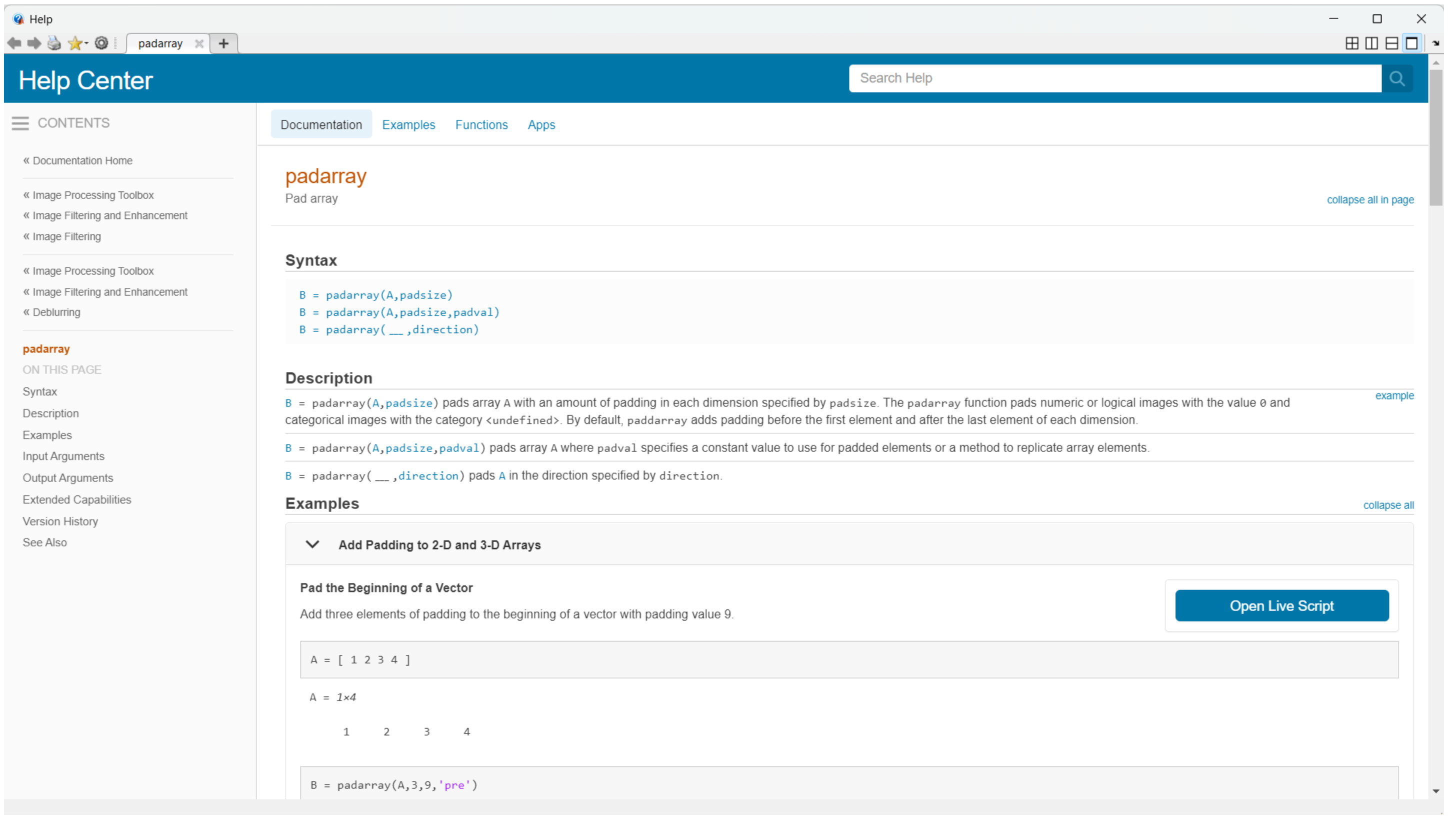
Maximum elements of an array

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)
```

As you can see, the function can take 1, 2, 3, or 4 input arguments. Out of this, one argument is **compulsory** – that is the input array. Other arguments are **optional** – they may be provided or not. You have to maintain the sequence while placing the arguments.

Let's solve exercise – 01 to understand the concept more clearly. To solve the problem, you can take the help of the *'padarray'* function which is built into MATLAB.



padarray
Pad array

Syntax

```
B = padarray(A,padsize)
B = padarray(A,padsize,padval)
B = padarray( __ ,direction)
```

Description

`B = padarray(A,padsize)` pads array `A` with an amount of padding in each dimension specified by `padsize`. The `padarray` function pads numeric or logical images with the value `0` and categorical images with the category `<undefined>`. By default, `paddarray` adds padding before the first element and after the last element of each dimension.

`B = padarray(A,padsize,padval)` pads array `A` where `padval` specifies a constant value to use for padded elements or a method to replicate array elements.

`B = padarray( __ ,direction)` pads `A` in the direction specified by `direction`.

Examples

Add Padding to 2-D and 3-D Arrays

Pad the Beginning of a Vector

Add three elements of padding to the beginning of a vector with padding value 9.

```
A = [ 1 2 3 4 ]
```

```
A = 1×4

     1     2     3     4
```

```
B = padarray(A,3,9,'pre')
```

As you can see from the figure, the *padarray* function takes 2, 3, or 4 arguments. Here, at least 2 arguments are compulsory. MATLAB will throw an error if you give only one input. You have to provide both the array and the 'padsize'. There are two optional arguments - 'padval' and 'direction'. You can find details of these parameters in the discussion section of the documentation. Also, take a look at the examples provided to familiarize yourself with different ways you can use the function.

Now, that you have seen and understood the concept, let us try to create custom functions with such variable-length arguments.

- MATLAB provides two special keywords for handling variable-length arguments:
  - varargin - short for "variable arguments in"
  - varargout - for "variable arguments out"

Internally, varargin is a **cell array** containing all the extra inputs. You access the input elements using {} brackets, like varargin{1}.

- nargin - is a function that stands for "number of input arguments". It is used to determine how many input arguments have been passed to that function when it is called.
- nargout - stands for "number of output arguments". It's used to determine how many output arguments the user has requested.

Example – 01

Let's start by creating a simple user-defined function with variable-length arguments.

For this problem, you will try to find the maximum of an array or integers provided inside the function input arguments. The built-in 'max' function can take an array as input or two input integers and return the highest among them. However, for your function, you want to allow the user to provide as many input integers as they want.

i. If the user provides one input – say, an array, the function will return the maximum of that array.
ii. If the user provides two or more integers as inputs, the function will return the maximum of all the integer values.
iii. If the user provides two or more arrays as inputs, the function will return the maximum of all the integer values.

```
function out = vf_01(varargin)

        v=varargin

        n=nargin

        x= cell2mat(varargin)

        out= max(x)

end
```

Try the following -

```
    ⇨ vf_01 ([2, 4, 10])
    ⇨ vf_01 (4, 10, 20, -200)
    ⇨ vf_01 ([6, 7], [3, 8], [9, 12])
    ⇨ vf_01( )
```

- Check the last test case. If you don't provide any input, your function will provide an empty array as output. It will **not provide an error**, unlike the 'max' function. **Why is that?**
- The reason behind this is that the input argument is written as varargin – any number of inputs. This includes no input as well. So, all the input arguments are optional here. You have not fixed anything.
- To modify this, say, you want to place at least one compulsory argument, you need to define your function in the following way –

```
function out = vf_02(a, varargin)
        a
        v=varargin
        n=nargin

        x= [a, cell2mat(varargin)]
        out= max(x)
end
```

Here, the 1st input parameter is defined – that means it is fixed. Then the function can take any number of arguments. Check the test cases now. It will provide the same output except for the last case – which will throw an error.

Example – 02

In the previous lecture, you have seen how to generate triangular numbers. Let's create a similar user-defined function that takes variable numbers of input arguments.

The goal of the function is to generate the **n-th Figurate number**. Figurate numbers (or polygonal numbers) are numbers that can be represented as regular geometric patterns (dots arranged in the shape of polygons).

Type	**Name**	**Formula**	**Sequence**
2-gonal	Linear	P(n) = n	1, 2, 3, 4, ...
3-gonal	Triangular	T(n) = n(n+1)	1, 3, 6, 10, 15, ...
4-gonal	Square	S(n) = n^2	1, 4, 9, 16, 25, ...
5-gonal	Pentagonal	P(n) = (n(3n−1))/2	1, 5, 12, 22, 35, ...
6-gonal	Hexagonal	H(n) = n(2n−1)	1, 6, 15, 28, 45, ...
7-gonal	Heptagonal	H(n) = (n(5n−3))/2	1, 7, 18, 34, 55, ...
8-gonal	Octagonal	O(n) = n(3n−2)	1, 8, 21, 40, 65, ...

The table above shows different figurate numbers and their associated formula. The general formula for the n-th figurate number for an s-gon (polygon with s sides) –

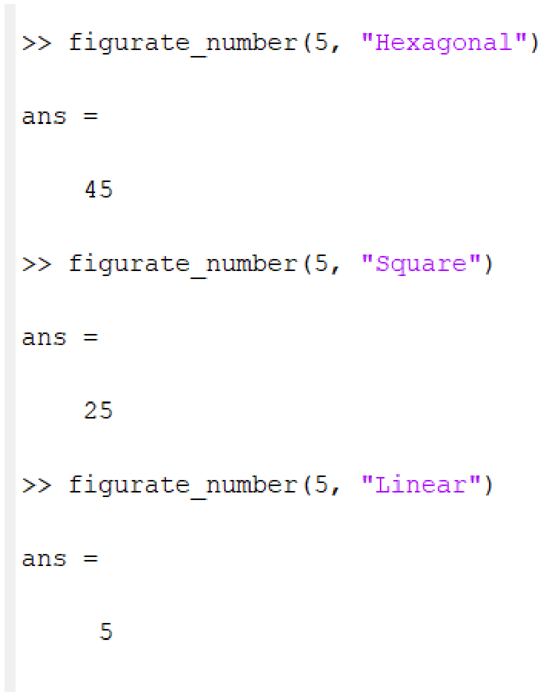
$$P\ (n, s) = \frac{(s-2) * n * (n-1)}{2} + n$$

Now, for this problem, you have to create a function that takes one, two (or three) inputs. These are as follows -

i. The first argument is the value n which indicates the n-th figurate number that you need to provide as output. This argument is compulsory.
ii. The second argument is optional. It takes a **string** indicating the name of the figurate number, e.g., 'hexagonal', 'square', etc. It is optional – so the user may provide it or may not. If not provided, your function will assume a default value – which is 'triangular'.

```
function out = figurate_number(n, varargin)

a= ["Linear", "Triangular", "Square", "Pentagonal", "Hexagonal", "Heptagonal", "Octagonal"];

    if nargin == 1
        s= 3;
    elseif nargin == 2
        x = varargin{1};        % the input string is taken into variable x
        s = find(a==x) + 1;
    else
        out= "Too many input arguments";
        return
    end

    out = (((s-2)*n*(n-1))/2) + n;
end
```

- Check different outputs with different combinations.
- What would have happened if you had not used the 'return' command inside the 'else'?
- What would have happened if the user mistakenly wrote 'Liner' instead of 'Linear'? Modify your code with an appropriate error message.



```
>> figurate_number(5, "Hexagonal")

ans =

    45

>> figurate_number(5, "Square")

ans =

    25

>> figurate_number(5, "Linear")

ans =

     5
```

Example – 03

Now, let's extend the previous question and create a function that takes three inputs (1 compulsory same as before, and 2 optional arguments). The additional 3rd argument is a string termed ‘all’ that indicates to provide a list of all the figurate numbers up to n.

iii. The string 'all'. If this argument is included, the function should return two outputs:
- The figurate number (as usual)
- A list containing all the figurate numbers up to n (the first argument)

```
function [val, arr] = figurate_number_v3(n, varargin)

      a= ["Linear", "Triangular", "Square", "Pentagonal", "Hexagonal", "Heptagonal", "Octagonal"];

if nargin == 1
      s= 3;

elseif nargin == 2
      x = varargin{1};          % the input string is taken into variable x
      s = find(a==x) + 1;

elseif nargin ==3
      x = varargin{1};          % the input string is taken into variable x
      s = find(a==x) + 1;

      if isequal(varargin{2}, 'all')
            arr=[];
            for i=1:n
                  a= (((s-2)*n*(i-1))/2) + i;
                  arr= [arr, a];
            end
      else
            val = 'Undefined input';
            return
      end

else
      val= "Too many input arguments";
      return
end

val = (((s-2)*n*(n-1))/2) + n;

end
```

Now, there are two output arguments. One is always present and the other one is optional – will provide a value only if the 3rd input is supplied.

- Check the output from the above function. To obtain two outputs, you need to call the function with two output variables inside an array, separated by a comma.
- Check the last two commands. It is proving an error. How can you fix that?

```
>> figurate_number_v3(10, 'Pentagonal')

ans =

   145

>> figurate_number_v3(10, 'Pentagonal', 'all')

ans =

   145

>> [w,q]=figurate_number_v3(10, 'Pentagonal', 'all')

w =

   145


q =

     1    17    33    49    65    81    97   113   129   145

>> [w,q]=figurate_number_v3(10, 'Square', 'all')

w =

   100


q =

     1    12    23    34    45    56    67    78    89   100
```

```
>> [w,q]=figurate_number_v3(10, 'Square')
Output argument "arr" (and possibly others) not assigned a value in the
execution with "figurate_number_v3" function.

>> [w,q]=figurate_number_v3(10, 'all')
Output argument "arr" (and possibly others) not assigned a value in the
execution with "figurate_number_v3" function.
```