# Lab Manual
# Lab Session-2 Control Flow and Functions in Matlab

## Control Flow

1. Conditional Control — if, else, switch.
2. Loop Control — for, while, continue, break.
3. Program Termination — return.

## Scripts and Functions

1. Overview.
2. Functions.
3. Types of Functions.
4. Global Variables.
5. User Created MATLAB function for operations on Signals

**Conditional statements** enable you to select at run time which block of code to execute. The simplest conditional statement is an if statement. For example:

```
yourNumber = input('Enter a number: ');
 if yourNumber < 0
        disp('Negative')
elseif yourNumber > 0
        disp('Positive')
else
        disp('Zero')
end
```

if statements can include alternate choices, using the optional keywords elseif or else. For example:

```
a = randi(100, 1);
if a < 30
    disp('small')
elseif a < 80
    disp('medium')
else
    disp('large')
end
```

when you want to test for equality against a set of known values, use a **SWITCH** statement. For example:

```
 color = 'rose';

switch lower(color)
  case {'red', 'light red', 'rose'}
    disp('color is red')

  case 'blue'
    disp('color is blue')

  case 'white'
    disp('color is white')

  otherwise
    disp('Unknown color.')
end
```

For both if and switch, MATLAB executes the code corresponding to the first true condition, and then exits the code block. Each conditional statement requires the end keyword.

## Array Comparisons in Conditional Statements

When you want to check for equality between two variables, you might use if A == B, ... This is valid MATLAB code, and does what you expect when A and B are scalars. But when A and B are matrices, A == B does not test if they are equal, it tests where they are equal; the result is another matrix of 0s and 1s showing element-by-element equality.

The proper way to check for equality between two variables is to use the **isequal** function:
if isequal(A, B), ...
isequal returns a scalar logical value of 1 (representing true) or 0 (false), instead of a matrix, as the expression to be evaluated by the if function. Using the A and B matrices from above, you get

```
isequal(A,B)
ans =
   logical
         0
```

## Loop Control — for, while, continue, break

The for loop repeats a group of statements a fixed, predetermined number of times. A matching end delimits the statements:

```
for a = 10:20
  fprintf('value of a: %d\n', a);
end
```

When you run the file, it displays the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

It is a good idea to indent the loops for readability, especially when they are nested:

```
 for i = 1:m
            for j = 1:n
               H(i,j) = 1/(i+j);
            end
```

end

**while**

The while loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching end delimits the statements.

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

When you run the file, it displays the following result −

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

**continue**

The continue statement passes control to the next iteration of the for loop or while loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for continue statements in nested loops. That is, execution continues at the beginning of the loop in which the continue statement was encountered.

```
a = 9;
%while loop execution
while a < 20
  a = a + 1;
  if a == 15
     % skip the iteration
     continue;
  end
fprintf('value of a: %d\n', a);
end
```

When you run the file, it displays the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

**break**

The break statement lets you exit early from a for loop or while loop. In nested loops, break exits from the innermost loop only.

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

When you run the file, it displays the following result −

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

**return**

return terminates the current sequence of commands and returns control to the invoking function or to the keyboard. return is also used to terminate keyboard mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a return statement within the called function to force an early termination and to transfer control to the invoking function.

## Scripts and Functions

### Overview

MATLAB provides a powerful programming language, as well as an interactive computational environment. You can enter commands from the language one at a time at the MATLAB command line, or you can write a series of commands to a file that you then execute as you would any MATLAB function. Use the MATLAB Editor or any other text editor to create your own function files. Call these functions as you would any other MATLAB function or command. There are two kinds of program files

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

### Scripts

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate.

For example, create a file called magicrank.m that contains these MATLAB commands:

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
        r(n) = rank(magic(n));
end
bar(r)
```

Typing the statement

```
magicrank
```

causes MATLAB to execute the commands, compute the rank of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables n and r remain in the workspace.

### Functions

Functions are files that can accept input arguments and return output arguments. The names of the file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

The following function named *mymax* should be written in a file named *mymax.m*. It takes five numbers as argument and returns the maximum of the numbers.

Create a function file, named mymax.m and type the following code in it −

```
function max = mymax(n1, n2, n3, n4, n5)

%This function calculates the maximum of the
% five numbers given as input
max =  n1;
if(n2 > max)
  max = n2;
end
if(n3 > max)
  max = n3;
end
if(n4 > max)
  max = n4;
end
if(n5 > max)
  max = n5;
end
```

The first line of a function starts with the keyword **function**. It gives the name of the function and order of arguments. In our example, the *mymax* function has five input arguments and one output argument.

The comment lines that come right after the function statement provide the help text. These lines are printed when you type −

```
help mymax
```

MATLAB will execute the above statement and return the following result −

```
This function calculates the maximum of the
  five numbers given as input
```

You can call the function as −

```
mymax(34, 78, 89, 23, 11)
```

MATLAB will execute the above statement and return the following result −

```
ans = 89
```

## Anonymous Functions

An anonymous function is like an inline function in traditional programming languages, defined within a single MATLAB statement. It consists of a single MATLAB expression and any number of input and output arguments.

You can define an anonymous function right at the MATLAB command line or within a function or script.

This way you can create simple functions without having to create a file for them.

The syntax for creating an anonymous function from an expression is

```
f = @(arglist)expression
```

Example

In this example, we will write an anonymous function named power, which will take two numbers as input and return first number raised to the power of the second number.

Create a script file and type the following code in it −

```
power = @(x, n) x.^n;
result1 = power(7, 3)
result2 = power(49, 0.5)
result3 = power(10, -10)
result4 = power (4.5, 1.5)
```

When you run the file, it displays −

```
result1 =  343
result2 =  7
result3 =  1.0000e-10
result4 =  9.5459
```

**Primary and Sub-Functions**

Any function other than an anonymous function must be defined within a file. Each function file contains a required primary function that appears first and any number of optional sub-functions that comes after the primary function and used by it.

Primary functions can be called from outside of the file that defines them, either from command line or from other functions, but sub-functions cannot be called from command line or other functions, outside the function file.

Sub-functions are visible only to the primary function and other sub-functions within the function file that defines them.

Example

Let us write a function named quadratic that would calculate the roots of a quadratic equation. The function would take three inputs, the quadratic co-efficient, the linear co-efficient and the constant term. It would return the roots.

The function file quadratic.m will contain the primary function *quadratic* and the sub-function *disc*, which calculates the discriminant.

Create a function file *quadratic.m* and type the following code in it –

```
function [x1,x2] = quadratic(a,b,c)

%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficients of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end   % end of quadratic

function dis = disc(a,b,c)
%function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end   % end of sub-function
```

You can call the above function from command prompt as −

```
quadratic(2,4,-4)
```

MATLAB will execute the above statement and return the following result −

```
ans = 0.7321
```

**Nested Functions**

You can define functions within the body of another function. These are called nested functions. A nested function contains any or all of the components of any other function.

Nested functions are defined within the scope of another function and they share access to the containing function's workspace.

A nested function follows the following syntax −

```
function x = A(p1, p2)
...
B(p2)
  function y = B(p3)
  ...
  end
...
End
```

**Example**

Let us rewrite the function *quadratic*, from previous example, however, this time the disc function will be a nested function.

Create a function file *quadratic2.m* and type the following code in it −

```
function [x1,x2] = quadratic2(a,b,c)
function disc  % nested function
d = sqrt(b^2 - 4*a*c);
end   % end of function disc

disc;
x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end   % end of function quadratic2
```

You can call the above function from command prompt as −

```
quadratic2(2,4,-4)
```

MATLAB will execute the above statement and return the following result −

```
ans =  0.73205
```

**Private Functions**

A private function is a primary function that is visible only to a limited group of other functions. If you do not want to expose the implementation of a function(s), you can create them as private functions.

Private functions reside in **subfolders** with the special name **private**.

They are visible only to functions in the parent folder.

Example

Let us rewrite the *quadratic* function. This time, however, the *disc* function calculating the discriminant, will be a private function.

Create a subfolder named private in working directory. Store the following function file *disc.m* in it −

```
function dis = disc(a,b,c)
%function calculates the discriminant
dis = sqrt(b^2 - 4*a*c);
end      % end of sub-function
```

Create a function quadratic3.m in your working directory and type the following code in it –

```
function [x1,x2] = quadratic3(a,b,c)

%this function returns the roots of
% a quadratic equation.
% It takes 3 input arguments
% which are the co-efficient of x2, x and the
%constant term
% It returns the roots
d = disc(a,b,c);

x1 = (-b + d) / (2*a);
x2 = (-b - d) / (2*a);
end      % end of quadratic3
```

You can call the above function from command prompt as −

```
quadratic3(2,4,-4)
```

MATLAB will execute the above statement and return the following result −

```
ans =  0.73205
```

## Global Variables

Global variables can be shared by more than one function. For this, you need to declare the variable as global in all the functions.

If you want to access that variable from the base workspace, then declare the variable at the command line.

The global declaration must occur before the variable is actually used in a function. It is a good practice to use capital letters for the names of global variables to distinguish them from other variables.

## Example

Let us create a function file named average.m and type the following code in it −

```
function avg = average(nums)
global TOTAL
avg = sum(nums)/TOTAL;
end
```

Create a script file and type the following code in it −

```
global TOTAL;
TOTAL = 10;
n = [34, 45, 25, 45, 33, 19, 40, 34, 38, 42];
av = average(n)
```

When you run the file, it will display the following result −

```
av =  35.500
```

## **User-defined functions for operations on signals - Examples**

### **Step Signal**

For the step discrete signal, let us call the function stepsignal. We will pass to stepsignal the time when the signal should start. We will call this time Sindex. We will also pass to it the starting and the ending of the time interval which we will call Lindex and Rindex for left index and right index. The function will return the signal *x*(*n*) and its index. The function is

```
function[xofn, index]= stepsignal(Sindex, Lindex, Rindex)
index = [Lindex: Rindex];
xofn = [(index – Sindex) >= 0];
```

### **Impulse Signal**

For the impulse signal, let us call the function impulsesignal. We will pass to it the point of application of the impulse signal, Sindex, and the range, Lindex and Rindex. The function will send to us the impulse signal *x*(*n*) and its index. The function is

```
function[xofn, index]= impulsesignal(Sindex, Lindex, Rindex)
index = [Lindex:Rindex];
xofn = [(index – Sindex) == 0];
```

### **xreflected**

The reflection of the signal *x*(*n*) is implemented using the MATLAB function fliplr. We will use fliplr to flip the sample values for *x*(*n*) and to flip the time index. The function will be called xreflected and will receive the original *x*(*n*) and the original index. It will give back the reflected *x*(*n*) and the new index. The function is

```
function [xnew, nnew] = xreflected(xold, nold);
xnew = fliplr(xold);
nnew = –fliplr(nold);
```

### **x1plusx2**

The function to add two discrete signals *x*1(*n*) and *x*2(*n*) will be called x1plusx2. It will receive the original signals and their original indices and return the sum of the two signals and the index for the sum. This function is

```
function[x, n] = x1plusx2 (x1orig, x2orig, n1orig, n2orig)
n = min(min(n1orig), min(n2orig)): max(max(n1orig), max(n2orig));
x1i = zeros(1, length(n));
```

```
x2i= x1i
x1i (find( (n >= min(n1orig))&(n <= max(n1orig))== 1))= x1orig;
x2i (find( (n >= min(n2orig))&(n<= max(n2orig)) == 1))= x2orig;
x = x1i+ x2i;
```

**x1timesx2**

The function to multiply $x1(n)$ and $x2(n)$ is similar to the x1plusx2 function
and is given next.

```
function[x, n] = x1timesx2 (x1orig, x2orig, n1orig, n2orig)
n = min(min(n1orig), min(n2orig)): max(max(n1orig), max(n2orig));
x1i = zeros(1, length(n));
x2i= x1i
x1i (find( (n >= min(n1orig))&(n <= max(n1orig))== 1))= x1orig;
x2i (find( (n >= min(n2orig))&(n<= max(n2orig)) == 1))= x2orig;
x = x1i. * x2i;
```

**xshifted**

A shifted version of $x(n)$ is $x(n - n0)$

$$xshift(n) = x(n - n0\ )$$

If $n - n0 = m$, then $xshift(m + n0) = x(m)$. This indicates that the sample values are not affected but the index is changed by adding the index shift $n0$. We will call the function xshifted and pass to it the original $x(n)$, the original index $n1$, and the amount of shift $n0$.

```
function[xnew, nnew] = xshifted (xold, nold, n0)
nnew = nold + n0;
xnew = xold;
```