**ADVANCED DIGITAL SIGNAL PROCESSING**

**Lab 1: An Introduction to MATLAB**

# Objective

The objective of this lab is to give a brief general introduction to MATLAB and to familiarize you with the most commonly used functions of MATLAB.

The Lab contains exercises that you should try out as you go along and are intended to be reasonably self-contained. However, the MATLAB manuals contains tutorial introductions and there are built-in demonstrations and tutorial material in the MATLAB software itself.

# 1 Introduction

The name MATLAB stands for matrix laboratory. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include Math and computation; Algorithm development; Data acquisition; Modeling, simulation, and prototyping; Data analysis, exploration, and visualization; Scientific and engineering graphics; Application development, including graphical user interface building.

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of time it would take to write a program in a scalar non-interactive language such as C or Fortran.

## 1.1 Entering and quitting MATLAB

To enter MATLAB, simply double click on the MATLAB icon and to leave MATLAB and return to the PC's operating system, simply type `quit`

## 1.2 The MATLAB Environment

After launching MATLAB, a multipanel window appears containing *Command Window*, *Workspace*, *Current Directory*, and *Command History* panels.

MATLAB has an extensive set of built-in functions as well as additional toolboxes that consist of functions related to more specialized topics like fuzzy logic, neural networks, signal processing, etc. It can be used in two different ways: as a traditional programming environment and as an interactive *calculator mode*. In *calculator mode*, the built-in functions provide a convenient means of performing one-off calculations and graphical plotting. Where as in *programming mode*, it provides a programming environment (editor, debugger, and profiler) that enables the user to write their own functions and scripts.

Expressions consisting of operators and functions that operate on variables are executed at the command-line prompt $\gg$ in the *Command Window*. All of the variables that are created are stored in the workspace and are visible in the *Workspace Window*.

Information about any function or toolbox is available via the command-line help function (or from the Help Browser launched from the Help menu):

$\gg$ `help sum`

If you need to find a build-in function for a particular operation, you may type,

$\gg$ `lookfor operation`

Example: If you like to find which function is used to find inverse of a matrix, you may type,

$\gg$ `lookfor inverse`

similarly, if you want to know which function is used to find transpose of a matrix, you type,

$\gg$ `lookfor transpose`

In MATLAB, everything that can be done using the GUI interface can also be accomplished using a command-line equivalent. The command-line equivalent is useful as it can be placed into scripts that can be executed automatically.

## 1.3 Functions and Scripts

Functions and scripts in MATLAB are just text files with `.m` extension. User-defined functions can be used to extend the capabilities of MATLAB beyond its basic functions. A user-defined function is treated just like any other function. Scripts are sequences of expressions that are automatically executed instead of being manually executed one at a time at the command-line. A script uses variables in the (base) workspace, while each function has its own (local) workspace for its variables isolated from other workspaces. Functions communicate only through their input and output arguments. A function is distinguished from a script by placing the keyword function as the first term of the first line in the file.

# 2 Matrices

A matrix is a rectangular or square array of numbers. MATLAB allows two dimensional array [A Matrix] and one dimensional array [A Row/Column vector]

One can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions in M-files.

For example: Explicit list of elements

```
≫ A = [1 2 3; 4 5 6; 7 8 9]
≫ A =
```

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

Some built-in functions that create matrices are:

- Ones ≫ `A = ones(4,4)`
- Zeros ≫ `X = zeros(5,5)`
- Rand ≫ `Y = rand(6,6)`
- Randn ≫ `Z = 10*randn(3,3)`

A `fix()` function can be used to obtain only integer values from `rand` and `randn` functions. i.e.

≫ `Z = fix(10*rand(4,4))`

## 2.1 Row Vector

To create a `vector` is pretty similar. Example below creates row vector;

≫ `V = [1 2 3 4 5]`

or

≫ `V = 1:1:5`

or

≫ `V = 1:5`

After pressing "return" or "enter" key on the keyboard, the value of `V` should have echoed back to you. To suppress the echo, use a semi-colon (`;`) following the command. For example,

≫ `V = [1 2 3 4 5];`

Another useful built-in function is `linspace`, which creates a linearly spaced vector of elements. For example, we may generate a vector `D` of 100 elements between 0 and 50 as follows,

≫ `D = linspace(0, 50, 100)`

## 2.2 Column Vector

The example below will create column vector, just by transposing the row vector, the command `transpose` or ' can be used to do this.

≫ C = [1 2 3 4 5]'

The semicolons (`;`) in matrix indicate the end of a row. All rows have to be the same length. Lets

## 2.3 Indexing

Once a matrix is created, one can refer to specific elements in it. MATLAB indexes matrices by row and column. Let `A = [1,2,3;4,5,6;7,8,9]` be a matrix.

≫ `A =`

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix}$$

The command `A(3,1)` will represent the element in the (3rd row, 1st column) which is 7. `A(2:3,1:2)` gives the elements in rows 2-3, and columns 1-2, so the result would be:

≫ `A =`

$$\begin{matrix} 4 & 5 \\ 7 & 8 \end{matrix}$$

`A(1:3,2)` gives the elements in rows 1-3, and the second column, that is, the entire second column. The same operation could be accomplished using command; `A(:,2)`. One can get a whole row of a matrix with command `c(1,:)`. This literally tells MATLAB to take the first row, all columns.

## 2.4 Operations

To multiply, divide, or do usuall mathematical operation when one has a matrix or vector that is acting as a set of data points,

```
.*
./
.^
```

The `.` before mathematical operators tells MATLAB to multiply each element in the matrix instead of trying to do matrix multiplication or division.

## 2.5 Built-in Functions

MATLAB has a lot of built-in functions. To use a function, just type `functionname(arguments)` with the appropriate name and arguments. For example, `s = sum(c)` which sums all the columns in a matrix `c` and returns a row vector of the sums. The function `d = det(c)` takes a matrix `c` and returns the determinant of

it. Typing `help commandname` gives help on a specific command. For example: ≫ `help det`

- `who` command displays all the variables that are currently defined.
- `whos` command displays the variables, their sizes, and some other information
- `pi` is a function that returns the value of pi

## 3 Random Integers

The `randint` function generates random integer matrices whose entries are in a range that one specifies. A special case generates random binary matrices.

For example, the command below generates a 5-by-4 matrix, that is 5 rows and 4 columns containing random integers ranging between 2 and 10.

≫ `c = randint(5,4,[2,10])`

If one desires the range to be [0,10] instead of [2,10], he can use either of the commands below. They produce different numerical results, but use the same distribution.

≫ `d = randint(5,4,[0,10]);`

≫ `e = randint(5,4,11);`

## 4 Plotting

The basic syntax to get a plot in MATLAB is `plot(x,y)`. The `x` value always come before the `y` value, and they represent variables in which data is stored. The `x` represents *x-axis* value and `y` represents *y-axis* value. The `hold on` command is used to hold the current plot, so that one can add plots on top of one another, until one resets it by typing `hold off`.

The command `plot(x1,y1,x2,y2)` is used to plot multiple values and to specify the color and linetype of a plot, the command is `plot(x1,y1,'w*')` to get white $*$ markers for each data point.

The results of signal processing computations in MATLAB are huge matrices. For example, frequency responses, which one would like to display in graphical form. We will go through a simple example of generating data, plotting the graph, putting labels on the axes etc. We will plot the graph of `y = sin(t);` for `t` going from 0 to 20 seconds. First we need to generate a vector `t` containing the values of time that we want to use. Type,

≫ `t=linspace(0,20,100);`

which generates vector `t` into a row vector of 100 values going from 0 to 20. Then type,

≫ `ys = sin(t);`

which generates the values of `sine`. To plot the set of points type,

≫ `plot(t,ys);`

The commands to add titles, labels and legends to the plots are; `title('This is a Title')`, `xlabel('My X-axis')`, `ylabel('My Y-axis')`, `legend('Series 1', 'Series 2')`

The basic graph can be prettied up using the following self explanatory sequence of commands,

≫ `title('Your roll number')`

≫ `xlabel('Time in seconds')`

≫ `ylabel('sin(t)')`

≫ `grid`

More than one set of points can be plotted on the same axes, and more than one graphs can be displayed on the screen at once.

The command `subplot` is used to split the plot into a bunch of smaller plots.

≫ `subplot(r,c,n)`

The above command splits the plot window into `r` rows and `c` columns and sets the current plot to plot number `n`. For example, subplot(2,1,1) splits the plot window into *2 rows* in a *1 column* and plots on top. To plot at bottom, the command is `subplot(2,1,2)`.

## 5 Logical Conditions

MATLAB has an interesting way of using logic. Let M be matrix,

≫ `M = [1 2 3 4 5 6]`

≫ `M>3`

This will return `[0 0 0 1 1 1]`, which means 3 elements of `M` are greater than 3.

A logical array of 1 (true) and 0 (false) values is returned as a result of applying logical operators to arrays;

≫ `a = [4 0 -2 7 0]`

`a =`

$$4 \quad 0 \quad -2 \quad 7 \quad 0$$

To know elements that are greater than 0,

≫ `a > 0`

`ans =`

$$1 \quad 0 \quad 0 \quad 1 \quad 0$$

To know any element that is equal to 7,

≫ `a == 7`

```
ans =
     0     0     0     1     0
```

To know any element that is not equal to 0,

```
≫ a ~= 0
ans =
     1     0     1     1     0
```

We can also perform logical AND, OR and NOT operation,

```
≫ (a >= 0) & (a <= 4)
ans =
     1     1     0     0     1

≫ (a < 0) | (a > 4)
ans =
     0     0     1     1     0

≫  ((a < 0) | (a > 4))
ans =
     1     1     0     0     1
```

A logical array can be used just like an index array to select and change the elements of an array;

```
≫ a(a > 0)
ans =
     4     7
```

The above example shows you the values of the vector according to the condition specified.

```
≫ a(a == 7) = 8
ans =
     4     0    -2     8     0
```

The above example replaces the specific value with the one specified.

```
≫ a(a ~= 0) = a(a ~= 0) + 1
ans =
     5     0    -1     9     0
```

The above example adds one to non zero value.

Typing `help ops` will list all of the logical functions and operators. Some basic and useful functions are shown below.

```
≫ zeros(3) % 3 by 3 matrix of zeros
≫ ones(3) % 3 by 3 unity matrix
≫ diag(3,2) % diagonal matrix
≫ triu(3) % upper triangle of matrix
≫ tril(3) % lower triangle of matrix
≫ rand(3,3) % random numbers
≫ magic(3) % 3 by 3 magic square
≫ toeplitz(3) % toeplitz matrix
≫ sin(a) % Trigonometric Function
≫ cos(a) % Trigonometric Function
≫ tan(a) % Trigonometric Function
≫ exp(a) % Exponential Function
≫ log(a) % Logarithmic Function
≫ rem(3,2) % Remainder after division
≫ sqrt(a) % Square root
≫ abs(a) % Absolute value
≫ sign(a) % Sign of scalar
≫ round(a) % Rounding off value
≫ floor(a) % Round to minus infinity
≫ ceil(a) % Round to plus infinity
≫ max(x) % Maximum Value
≫ sum(x) % Sum of elements
≫ median(x) % Median of values
≫ any(x) % True if any element is ≠ 0
≫ all(x) % True if all elements are ≠ 0
≫ min(x) % Minimum Value
≫ prod(x) % Product of values
≫ mean(x) % Mean of values
≫ sort(x) % Sort the values
≫ std(x) % Standard Deviation
```

# 6 Control Structures

In MATLAB, `FOR` loops iterate over the columns of an array, and logical expressions are used for conditional evaluation in `IF` statements and `WHILE` loops.

## 6.1 FOR Loop

```
≫   for i = 1:3, i, end

i =
     1

i =
     2

i =
     3

≫   for i = 5:-2:1, i, end
```

```
i =
   5

i =
   3

i =
   1
```

Any type of array can be used; e.g., a character array:

```
≫   chararray = 'abc'

chararray =
   abc

≫   for i = chararray, i, end

i =
   a

i =
   b

i =
   c
```

Because any type of array can be used in a FOR loop, using FOR loops can greatly slow down the execution speed of MATLAB as compared to the equivalent array operation.

### 6.2 IF statement

```
≫   n = input('Enter a Number:  ');
≫   if n > 0
≫     disp('Positive value.')
≫   elseif n < 0
≫     disp('Negative value.')
≫   else
≫     disp('Its Zero.')
≫   end
```

### 6.3 WHILE Loop

```
≫   n = 3;
≫   while n > 0
≫     n = n - 1
≫   end

n =
   2

n =
   1

n =
   0
```

### 6.4 DO-WHILE Loop

```
≫   done = 0;
≫   while ~done
≫     n = n + 1
≫     if n >= 3, done = 1; end
≫   end

n =
   1

n =
   2

n =
   3
```

The DO-WHILE loop ensures that the statements in the loop are evaluated at least once even if the logical expression is not true. The logical expression in a conditional statement must evaluate to a single logical scalar value. To convert a logical vector to a scalar, the functions `any` and `all` can be used.

```
≫   a = [5 0 -1 9 0];
≫   a > 0

ans =
   1 0 0 1 0

≫   any(a > 0)

ans =
   1

≫   all(a > 0)

ans =
   0
```

## 7 Array Manipulations

1. How to create a row vector that increments by 1?
   [ Answer: `r = 1:10` ]
2. How to create a column vector that increments by 1?
   [ Answer: `c = transpose(1:10)` ]

3. How to create a row vector that increments by a specific value?
   [ Answer: `r = 1:2:20` ]

4. How to create a row vector that decrements by a specific value?
   [ Answer: `r = 20:-1:1` ]

5. How to create a row vector with equally spaced points? Let's create a vector that goes from 0 to 100 with 21 equally spaced points.
   [ Answer: `r = linspace(0,100,21)` ]

6. How to create a row vector of zeros?
   [ Answer: `r = zeros(1,10)` ]

7. How to create a row vector of ones?
   [ Answer: `r = ones(1,10)` ]

8. How to append a vector? lets say we add 11 to the end of the vector `r` already defined.
   [ Answer: `p = [r 11]` ]

9. How to append two vectors together? lets say we have already defined vectors `r` and `p`.
   [ Answer: `j = [r p]` ]

10. How to remove a particular entry from a vector? lets say we want to remove the 4th entry from already defined vector `r`.
    [ Answer: `r(4) = []` ]

11. How to replace a particular entry with a different entry within a vector? lets say we want to replace the 4th entry from row vector `r` with the value of 100.
    [ Answer: `r(4) = 100` ]

12. How to remove the last element or entry from a row vector `r`?
    [ Answer: `r(end) = []` ]

13. How to remove a series of elements or entries from a row vector `r`?
    [ Answer: `r(3:6) = []` ]

14. How to get the number of elements or entries within a row vector `r`?
    [ Answer: `s = length(r)` ]

15. How to remove repeating elements within a row vector `r`?
    [ Answer: `s = unique(r)` ]

16. How to compare two vectors namely `r` and `s` for equality?
    [ Answer: `q = isequal(r,s)` ]

17. How to determine whether the vector `r` is empty?
    [ Answer: `s = isempty(r)` ]

18. How to know whether any element or entry in a vector `r` has a non-zero? The output of 0 means no non-zero and the output of 1 means the vector has a non-zero.
    [ Answer: `s = any(r)` ]

19. How to know whether all elements or entries in a vector `r` are non-zero? The output of 0 means the vector has a zero and the output of 1 means the vector has no zero.
    [ Answer: `s = all(r)` ]

20. How to find the element or entry the repeats maximum in a vector `r`?
    [ Answer: `s = mode(r)` ]

# 8 Exercise

1. Create a row and column vector containing random 5 elements. Assign the row vector to a variable `L` and column vector to `M`. Investigate the impact of following commands.
   (a) L(2)
   (b) sum = L + M'
   (c) diff = L – M'
   (d) prod = L*M
   (e) ab = [L M']
   (f) ab(2:6)
   (g) ac = [L ; M']

2. Create two 3-by-3 matrix's containing random integer elements. Assign the one matrix to a variable `A` and other to `B`. Investigate the impact of following commands.
   (a) A+B
   (b) A*B
   (c) A' – B
   (d) 2A + 4B
   (e) A(1,:)
   (f) A(2,:)
   (g) B(:,1)
   (h) B(:,3)

3. Create a singular matrix [whose determinant is zero], find its inverse, and the transpose of the matrix. [Hint: use `det()`, `inv()`, `transpose()` function]

4. Create $4 \times 4$ matrix and assign its diagonal elements to a variable, lets say `v`. [Hint: use `diag()` function]

5. Find the maximum element of the matrix: `A = [12,2,3;10,11,8;9,17,5]` [Hint: use `max()` and `A(:)` function]

6. Let `Z = [1 1 1;2 2 2;3 3 3]` be a matrix, delete its 2nd entire row. [Hint: `Z(:,2)`]

7. Write a program using `else if` structure to display even and odd numbers from 1 to 20.

8. Generate and plot a sine as well as cosine wave on same figure using `hold on` function. Repeat the same task and display the sine and cosine wave on two same graphs using `subplot` function.