Laboratory Assignment 1

**Purpose**

In this lab, you will be introduced to MATLAB by using it to evaluate expressions containing complex numbers. Using MATLAB, you can easily express these complex numbers in either rectangular or polar form. MATLAB’s plotting capabilities will be introduced and used extensively. You will also use M-files and create some simple signal processing functions that are used later in other lab assignments.

**Objectives**

By the end of this laboratory assignment, you should be able to:

* Use MATLAB to perform complex arithmetic
* Generate and plot signals and complex valued functions
* Confidently develop MATLAB M-files and save results of computations from a MATLAB session.

**Reference**

Review Topics:

* Algebra of complex numbers
* Sketching of discrete and continuous time signals
* Vector and matrix algebra

**TUTORIAL PART I**

1. **Starting MATLAB**

Start MATLAB by double (left) clicking on the MATLAB icon on the desktop of your computer. You will be working on the command window, and will enter all command after the prompt: >>

1. **Evaluating Complex Variables and Expressions**

In the following text, information that you enter will be preceded by the MATLAB prompt; information not preceded by a prompt is printed by MATLAB as a computation result or other information.

**Problem 1:**

Express each of the following complex numbers in Cartesian form, i.e., z = x+jy, where x = Re(z) and y = Im(z). Plot part (a) in the complex plane.

* 1. Jej11π/4
  2. (1-j)10

Part a: You can find the Cartesian form by typing the expression, using standard symbols for arithmetic operations.

>> j\*exp(j\*11\*pi/4)

ans =

-0.7071 - 0.7071i

Note that MATLAB has evaluated the expression and echoed the result to the screen, expressed in Cartesian form as the variable **ans**. Also, just like any programming language, **exp(x)** returns ex. Other standard functions, including trigonometric functions, are available; type **help elfun** for a list. Additionally, **pi** is defined as a special variable having the value π, and **j** is defined as √(-1). Any special variable will act as defined until you change its value by assigning a new value to it. For example, to change **pi** to 3, issue the command

>> pi = 3

pi =

3

>> pi

pi =

3

As you can see, **pi** has been changed. If you clear your definition, the old one reappears:

>> clear pi

>> pi

ans =

3.1416

Other variables can be set the same way. For instance, we can set z to the solution of part b:

>> z=(1-j)^10

z =

0 -32.0000i

1. **Plotting Complex-Valued Functions**

Plots in MATLAB are generated using the plot function. **Plot (x, y)** generates a plot where the values of the vector **x** indicate points along the horizontal axis corresponding to the values in the vector **y** that are to be plotted on the vertical axis. Vectors x and y must have the same number of elements.

Since complex values have two components corresponding to **x+jy**, MATLAB provides the **real** and **imag** functions to separate the real and imaginary parts of a complex number:

>> z = 3+4j;

>> zr = real(z);

>> zi = imag(z);

real and imag break z into real and imaginary parts in the variables zr and zi, respectively.

Note that typing a semicolon at the end of the command prevents MATLAB from printing the result on your screen. This will be important when you create large matrices and vectors.

To plot a complex number, we can either plot the real parts vs. the imaginary parts or let MATLAB do it for us. Here, we supply **zr** and **zi** to the plot function:

>> plot(zr,zi,'\*'); grid

and MATLAB generates the plot:



The ‘**\***’ parameter to the plot function tells MATLAB to generate an \* shape for each data point instead of a line display. Since we only plotted one data point, this is extremely useful. The ‘**grid**’ command tells MATLAB to display a grid on the figure. In general you should always label axes on your plot and include a title. **help plot** shows you other characters that can be used as well as the different colors that can be used on the plot.

This plot would be exactly the same if we entered:

>> plot(z,'\*'); grid

since the MATLAB default for plotting complex numbers is to plot the real parts on the horizontal axis and the imaginary parts on the vertical axis.

Multiple sets of parameters can be given to **plot**; each pair of arguments is taken as x and y data pairs.

If you wish to have several plots shown at once on different sets of axes, use the **subplot** command or open new figures by typing **figure** and creating a new plotting window.

**TUTORIAL PART II: Vectors and Matrices**

Matrices and vectors make up the heart of MATLAB computations. In this section, matrix and vector manipulations will be introduced. A vector is a one-dimensional list of values. Vectors hold single signals or lists of data. They can be assigned a name and treated as any other variable in MATLAB; however, operations performed on vectors are done element by element.

As an example of this, consider the function, y = 3x+2. If we want to plot y as a function of x, we first create an x vector containing data points in the range of interest. Suppose the range is 0 to 5, using every integer point. There are ways to create this data set in MATLAB. The first way is to type every pont:

>> x = [0 1 2 3 4 5]

x =

0 1 2 3 4 5

This generates a row vector **x**, i.e., a 1X6 matrix containing six elements, the integers 0 through 5. An easier way to generate this same vector is to use a range generating statement:

x = 0:5

x =

0 1 2 3 4 5

The colon operator acts like the word “to”, in effect generating the function “0 to 5’. A step size of 1 is the default. A different step size - positive, negative, real or integer - can be specified by placing the step value between the beginning and end of the range, as in **k** below:

>> k = 0:0.01:5;

which is a vector containing 501 data points that are 0.01 apart, starting from 0 and ending at 5.

Type help **linspace** to learn about another method to generate vectors in MATLAB.

The next step is to evaluate **y**, using **x** as defined above.

y = 3\*x+2

y =

2 5 8 11 14 17

This statement instructs MATLAB to multiply every element in **x** by 3 and then add 2 to every element, storing the results in **y**. Thus 3\***x** is treated as a scalar multiplication of a vector and the 2 is implicitly treated as a vector of the same length as **x** comprising all 2s.

Note that when a dot precedes an operator, as using a **.\*** for multiplication, it implies that each element in the vector (matrix) results from applying that operator to corresponding elements in the first and second vectors (matrices). For example, dot multiplication of two mX1 vectors results in an mX1 vector where each element is the product of the corresponding elements in the first and second vectors. Note that **.\*** is the “dot product”, pr inner product operation from linear algebra. This type of multiplication requires that the vectors or matrices be of the same size and is called pointwise, rather than vector or matrix multiplication.

Since MATLAB is based on matrix operations, it is important to recall that you can only add or subtract matrices having the same dimensions, e.g., the addition of a **3X2** matrix to a **2X3** matrix is undefined. Matrix multiplication requires that the number of columns in the first matrix be the same as the number of rows in the second matrix. For example, multiplication of a **2X5** matrix **A** and a **5X3** matrix **B** results in a **2X3** matrix **C=AB**, whereas the multiplication **BA** is undefined. However, the multiplication **D = B’A** is defined, where **‘** denotes the transpose operation in MATLAB.

1. Generating Complex Functions

Let’s generate values for the complex function f(t) = 3ej3πt for t ranging from 0 to 1 in 0.001 increments. The first step is to create a time variable.

>> t = 0:0.001:1;

Here the semicolon at the end is really important unless you want to see all 1001 values echoed back on your screen.

Next, construct a vector containing value of this function for each time value in **t**:

f = 3\*exp(j\*3\*pi\*t);

The variable **f** now has the complex result of the function evaluation.

It should be pointed out that transcendental functions (e.g. **sin**, **cos**, **exp**) in MATLAB work on a point-by-point basis; in the above command, the function **exp** computes a vector where each element is the exponential of its corresponding element in **j\*3\*pi\*t** (10001 total elements).

1. Accessing Vectors and Matrices

The data in vectors can be viewed and displayed in several different ways: it can be plotted to the screen, printed on paper, and saved electronically. It is not, however, always desirable to access the entire vector at once when displaying the information in it. To access single elements or ranges of a vector, an index element or list that indentified which elements are of interest is needed.

Elements in MATLAB vectors are identified by the vector name and an integer number or index. In MATLAB, only positive integer indices are used. Thus the first element in a row or column vector f is denoted by f(1), the second element by f(2), and so on. To access specific elements in a vector you need to use the name of the variable and the integer index numbers of the elements you wish to access. Range statements can also be used for indices to access the indexed elements. For example,

>> f(25);

>> f(3:10);

>> f(1:2:50);

The first line accesses the 25th element of **f**. The second accesses elements 3 through 10, inclusive; and the third returns the odd numbered elements between 1 and 50.

Elements in matrices require the use of two-dimensional indices for identification and access. For example, **f(3,2)** returns the element in the third row, second column; ranges can also be used for any index. For example, **f(1:3, 4:8)** defines a matrix that is equivalent to a section of the matrix **f** containing the first, second and third rows, and the fourth through eighth columns.

If a **:** is used by itself, it refers to the entire range of that index. For example, a 3X5 matrix could have its fifth column referenced by **f(:,5)**, which means “all rows, 5th column only,” as well as **f(1:3, 5)**, which means rows 1 to 3, 5th column only.

The index number can be another variable as well. This is useful for creating programming loops that execute the same operations on the elements of a matrix.

From other computer programming experiences, you should be familiar with the idea of creating a loop to repeat the same task for different values. Here’s an example of how to do this using MATLAB. Suppose we want to generate an output vector where each element is the sum of the current element and the element from 10 back in an input vector. The task to be repeated is the sum of two elements; we need to repeat this for each element in the vector past 10. The elements of the vector **x** define the input ramp function to be integrated, **y** will hold the result, and **k** is the loop index:

>> x = 3\*(0:0.1:5)+2;

>> y = zeros(size(x));

>> for k = 11:length(x),

y(k)=x(k-10)+x(k);

end

>>

When a loop structure is entered in the command window of MATLAB, the body of the loop does not have a prompt printed; the command line acts the same as with a prompt. Note that the loop will not be executed until the **end** command is entered and followed by a carriage return.

As you can see, the variable **k** is set to range from 11 to length of **x**; this allows **k** to index all elements in **x**. **k** is set to increment by integers.

Loops generally are not desirable since they take a very long time to run. If possible, rewrite your operations in terms of vector additions and multiplications instead of looping. For example, we can rewrite this problem to use vector addition by creating two new vectors, one which is **x** offset by 10 and the other which is **x** padded with 10 zeros (since we can only add vectors of similar lengths).

>> x = 3\*(0:.1:5)+2;

>> x1 = [zeros(1,10) x];

>> x2 = [x zeros(1,10)];

>> y=x1+x2;

**Storing Results and Using M-files**

Usually you want to save the results that you have generated during a MATLAB session, including data vectors and commands used to process them. This can be accomplished by:

* + - Using the **diary** command to save a record of all command typed. Enter **help diary** to learn how to use this command.
    - Using M-files that you have created using the editor window in MATLAB. Such files have a **.m** extension and contain a list of MATLAB command to be executed when you type the filename.
    - Saving the contents in variables for the next session using the **save** command.

It is recommended that you become familiar with M-files. They are extremely useful and will save you much time and effort. There are two typed of M-file: scripts, which are essentially a series of command typed into a file instead of the command window; and functions, which allow you to create new MATLAB functions.

**Creation of a Function in MATLAB**

A function in MATLAB is a special kind of M-file. The first line defines the function, both giving it a name and indicating what values are to be passed as arguments to the function and those that are to be generated by the function. Once you have created a function, you can use it as you would any built-in function in MATLAB, such as **help** or **plot**.

A Useful feature of function files is that the lines that follow the function definition and begin with a comment symbol (%) are printed when help is requested for your function (**help yourfunction** prints these lines).

Make a function that takes two variables as arguments, adds 1 to the first variable, multiplies the second variable by two and returns the product of the two variables. This function is called **blackbox**. The file is called **balckbox.m**

Solution:

function [output] = blackbox(a,b)

%Adds 1 to argument a and multiplies b by 2

% Returns the product of a nd b in output

%

% if the two variables are not if the same size

%the larger variable is stripped to be the same size as the smaller

%Detremine the lengths of each vector

la = length(a);

lb = length(b);

%Add 1 to a

a = a+1;

%Multiply b by 2

b = b\*2;

%Compare the lengths; truncate the longer vector

if la < lb

output = a.\*b(a:la);

else

output = a(1:lb).\*b;

end

The elements in this file that are required in order for MATLAB to recognize it as a valid function are:

1. The M-file must start with the word function.
2. The variable [output] (could be any other name) must be the function output.

Note that while the function name may be different from the filename, the filename MUST be use to access the function.

**ASSIGNMENTS**

1. Consider the functions p=3sin(x2) + 2cos(y3)and q = 3cos(xy)+2y2, where x is in the range 0 to 5 with 0.01 increments and y = 0.05x+2.01.
   1. Plot y, p and q against x. Grid your plot, label your axes and include a title for your plot.
   2. Determine k = p+q and plot k against x
2. Plot the function y(t) = (1-e2.2t)cos(60πt). Use t from 0 to 0.25 in 0.001 increments.
3. A polynomial function has roots at -2, 2, -2+3j, -2-3j. Determine the polynomial, plot the four roots in the complex plane, and plot the polynomial function for the range -5 to 5 in steps of 0.01. (Label your axes and grid/title your graph). Use help to look up the functions **poly**, **roots**, and **polyval** to help you with this problem.
4. Consider the complex function f(t) = 3e-j2πt+ π/4

On the same plots but different graphs, plot the real and imaginary parts as a function of time from 0 to 3 seconds in 0.01 increments. Also plot the magnitude and phase of f as a function of time on the same plots but different graphs. You may wish to look up the functions **subplot, abs** and **angle**.

1. Generate a file that calculates the sine wave of 5 Hz for 3 seconds using 0.001-second increments and plots the sine wave versus time with all axes labeled. Display the length of the time sample and the length of the sine wave calculated
2. Generate a function called squarer that returns the vector with each element squared. Use this function with the sine wave from the previous problem and display the sum of the elements in both the sine wave vector and the vector composing square of the sine wave.
3. Write a MATLAB function **half** that removes every other element from a vector of arbitrary length, creating a shorter vector made out of only the odd-numbered elements of the original vector; and a MATLAB function **double** that creates a longer vector by adding an additional element between neighboring elements in the original vector. Each new element should equal the average of its neighboring elements. Use only matrix/vector manipulations: do NOT use loops. Test your solution by applying **half** and **double** to the vector **x = [1 2 3 4 5 6 5 4 3 2 1]**. What happens after the execution of **half** followed by **double**, and **double** followed by **half**?