# Lab 1: MATLAB Functions and Script Files

The objective of this lab is to introduce you to the basic operations and script files of MATLAB. Practice each new command by completing the examples and exercises. Turn-in the answers for all the exercise problems as your lab report. When answering the problems, indicate the commands you entered and the output displayed. It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document. Similarly, plots can also be pasted to your word document.

# Part 1: Complex Numbers in Matlab

### 1 Complex numbers in Matlab

### 1.1 Complex numbers

One of the strengths of MATLAB is that most of its commands work with complex numbers, too. MATLAB, by default, uses the letter i for the square root of (-1). However, electrical engineers typically prefer using j, and so MATLAB has both predefined. Because of this, you may wish to avoid using i and j as variables if your code deals with complex numbers. That being said, everything in MATLAB is a variable. You may redefine the variables i and j to be anything you like.

**Exercise 1:**

---

1. Enter sqrt(-1) into MATLAB. Does the result make sense?
2. Enter i+j. Does the result make sense?
3. Define z1 = 1+j. Find the magnitude, phase, and real part of z1 using the following operators: abs(), angle(), real(), imag(). Is the phase in degrees or radians?

4. Find the magnitude of z1 + z2, where z2=2*exp( j*3/pi )

---

### 1.2 Complex functions

MATLAB handles complex functions of time in the same way as real ones: defined over a range of time.

**Exercise 2:**

---

Create a signal $x_1 = te^{jt}$ over the range [-10,10] (t=-10:10).
Plot the real and imaginary parts of $x_1$ in one window (using subplot). Notice that one plot is odd, and the other is even. Why is that?

---

## 2. Script Files

Scripts are M-files that contain a sequence of commands that are executed exactly as if they were manually typed into the MATLAB console. Scripts are useful for writing things in MATLAB that you want to save and re-run later. A script file also gives you the ability to go back later and edit your commands, such as when you would like to re-run a function with a different parameter.

To create script files, you need to use a text editor such as Notepad on a Windows PC or emacs on Linux and Mac computers. MATLAB also has an internal editor that you can use within the MATLAB GUI. You can start the editor by clicking on an M-file within the MATLAB file browser. All of these editors are standard tools and will produce plaintext files that MATLAB can read.

Your M-files should each contain a header. In general, a header is a block of commented text that contains the name of the file, a description, and your name and the date. For example:

```
% john smith
% ee235 spring 2020, lab 1
% dampedCosine.m
% produces a plot of a cosine with frequency 1 Hz, with amplitude
% scaled by a decaying exponential (y).
```

<code goes here>

Download the dampedCosine.m script from the course webpage.

Save the script in your current working directory, otherwise MATLAB will not be able to find the file. You need to run the dampedCosine.m script by typing dampedCosine at the MATLAB prompt. Open the script in an editor, and read the code. You may safely ignore the 'diary' commands entirely.

**Exercise 3:**

---

Create a copy of dampedCosine.m, and call it dampedCosine_YourName.m. Open dampedCosine_YourName.m in an editor and give it a proper header.

Edit dampedCosine_YourName.m to create **a second signal that is a cosine with twice the period as the original signal** (think about how you can change the frequency of **cos(2*pi*x)** on line #2).

Add or change commands to plot the two signals together, with the original signal on top and your second signal on the bottom (subplot). You will need to use the subplot and plot commands.

Run your script, and save the output plot. (You can typically just right-click save the image). Turn-in the code and output with your lab report.

---

**Exercise 4:**

---

What is the period of the second signal?
How do the envelopes (envelope: a curve tracing the signal's peaks or overall shape of a signal) of the two signals compare?
What is the difference between the second signal that you just made, and a **third signal which has half the period of the first one**?

---

Download the compexp.m script from the course webpage and save it in your current working directory. This script specifies a complex exponential funtion y(t). This script generates five plots: one 3-D (X,Y,Time), and two pairs of 2-D plots: one pair in Cartesian coordinates (Real and Imaginary axes) and one pair in polar coordinates (radius and angle).

Run compexp.m, and look at the graphs. You can rotate the 3-D graph around by clicking on the "Rotate 3D" toolbar button, then clicking on the graph and dragging the pointer around.

**Exercise 5:**

---

Create a copy of compexp.m called compexp_YourName.m
Add **a second signal** to compexp_YourName.m that has **half the oscillation period** of the original. Generate a new set of 5 plots for the second signal.

Add a third signal to compexp_YourName.m that decays noticeably faster than the original.

Generate a new set of 5 plots for the third signal. The **figure()** command can be used to generate a new figure window for plotting.

Check that your code is commented and has a header.
Run your script, and save the output plots.

---

**Exercise 6:**

---

Explain intuitively what the 3-D process looks like in 2-D.

Explain how the 3-D graph is consistent with each of the pairs of 2-D graphs. Explain how the plots of the second and third signals confirm that your code is correct. In

other words: how do your output signals satisfy your goals of “oscillate faster” and “decay faster”?

---

# Part 2: Functions in MATLAB

**Summary:** Learn about simple functions and representing signals in Matlab.

You will learn about MATLAB function files, which we will refer to as m-files and have the file extension `*.m`, the same as script files. You will create a number of functions for manipulating sound signals.

The difference between a function and a script is that functions can return values and take parameters, while scripts are simply a collection of commands. Unlike script files, any variables created in the function are not available after the function has run, that is, the variables are active inside the scope of the function, and the variables cannot be accessed out-of-scope. The MATLAB online help system has a nice write-up about functions and how to handle various things like returning more than one value, checking the number of arguments, etc. To learn more, type the following commands and read the online help:

```
>> help function
>> help script
```

You need to use a text editor to create function files. MATLAB has an internal editor that you can start by clicking "File" and then "New" "m-file".

### note:

Remember that to run custom scripts or functions, the MATLAB working directory needs to be set to the location of those files.

# Sound in MATLAB

### Loading and plotting a sound

---

You will be playing and visualizing a lot of sound files in this course.
Load the built-in sound named ***handel***, plot it, and play it.

TURN THE VOLUME DOWN ON YOUR COMPUTER FIRST!

```
>> load handel;
>> plot(linspace(0,9,73113),y);
>> sound(y); % played at the default sampling rate.
```

What does load do? What does 'linspace' do? Type >>help linspace to see how to use it.

---

Download the sound samples from the class webpage and save them to your working directory. Use **audioread** to load `.wav` files (e.g. **[SS, Fs]=audioread('bleeep.wav')**] , and use **load** to load `.mat` files. Plot each one in turn, and try to guess what it will sound like (the name might help).

On a computer, sounds are represented digitally, which means that only samples of the signal at fixed time intervals are stored. We'll learn more about this later. For now, you just need to know that the time interval Ts (or equivalently the sampling rate Fs=1/Ts) is something you need to keep track of for playing sounds.

Now play each sound. The goal is to learn how the time domain signal sounds. Use the sound command to play a sound signal. You must specify the playback sample rate (Fs), which will be the same as the sample rate of the sound samples on the web site (they are 8000 Hz). For example, if you wanted to play a sound called bell and its sample rate was 8000 Hz, then you would enter the following command,

```
>> load('fall.mat');
>> Fs=8000;
>> sound(fall, Fs);
```

If you use a different value for Fs, you will effectively be doing time scaling. You cannot use **sound(fall)** to play the sound.

When working with sound in MATLAB, it is important to remember that the values of the audio signals are in the range [-1, 1]. Keep this in mind when you are writing your functions. Your functions should expect inputs with values in the range [-1, 1] and anything out of that range will be clipped when you play the sound.

# Function Files

Now you will learn how to make functions that will allow you to modify the sound signals in various ways. Let's create some functions that will let us time scale, reverse, delay, fade, and repeat a sound, and mix two sounds together.

Functions are programs (or routines) that accept input arguments and return output arguments. Each M-file function (or function or M-file for short) has its own area of workspace, separated from the MATLAB base workspace. The base workspace stores variables that you create at the

command line. Variables in the base workspace exist until you clear them or end your MATLAB® session. Functions do not use the base workspace. Every function has its own function workspace. Each function workspace is separate from the base workspace and all other workspaces to protect the integrity of the data. Variables specific to a function workspace are called local variables. Typically, local variables do not remain in memory from one function call to the next.

**Input and output arguments:** the input arguments are listed inside parentheses following the function name. The output arguments are listed inside the brackets on the left side. They are used to transfer the output from the function file. The general form looks like this:

function [outputs] = function_name(inputs)

When a script file is executed, the variables that are used in the calculations within the file must have **assigned values**. The assignment of a value to a variable can be done in three ways.

1. The variable is defined in the script file.

2. The variable is defined in the command prompt.

3. The variable is entered when the script is executed.

Both scripts and functions allow you to reuse sequences of commands by storing them in program files. Scripts are the simplest type of program, since they store commands exactly as you would type them at the command line. Functions provide more flexibility, primarily because you can pass input values and return output values. For example, this function named addnum computes the sum of aa and bb and returns the result (ss).

```
function ss = addnum(aa,bb)
    ss = aa+bb;
end
```

This type of function must be defined within a file, not at the command line. Often, you store a function in its own file. In that case, the best practice is to use the same name for the function and the file (in this example, addnum.m), since MATLAB® associates the program with the file name. Save the file either in the current folder or in a folder on the MATLAB search path.

You can call the function from the command line, using the same syntax rules that apply to functions installed with MATLAB. For instance, calculate the sum of 5 and 10.

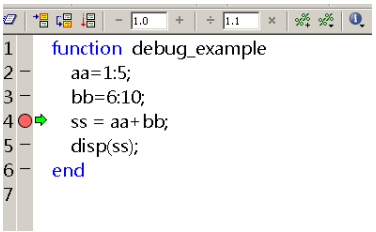
```
>>a = 5; b=10;
>>y = addnum(a,b)

y =

   15
```

Or call the function directly: >>y = addnum(5, 10)

**Debug a MATLAB Program：** To debug your MATLAB® program graphically, use the Editor. Before you begin debugging, make sure that your program is saved and that the program and any files it calls exist on your search path or in the current folder. If you run a file with unsaved changes from within the Editor, then the file is automatically saved before it runs.

**Set Breakpoint：** Set breakpoints to pause the execution of a MATLAB file so you can examine the value or variables where you think a problem could be. **It is very useful to check the intermediate results at a breakpoint and you can examine the variables in the Workspace of Matlab. For example, you can check the size of a variable or the actual values in a vector/matrix to see if they are what you expected**.

You can set breakpoints using the Editor. To add a standard breakpoint in the Editor, click the breakpoint alley at an executable line where you want to set the breakpoint. The breakpoint alley is the narrow column on the left side of the Editor, to the right of the line number. Executable lines are indicated by a dash (—) in the breakpoint alley. For example, click the breakpoint alley next to line 4 in the code below to add a breakpoint at that line.




After setting breakpoints, run the file from the Command Window or the Editor. The prompt in the Command Window changes to K>> indicating that MATLAB is in debug mode and that the keyboard is in control. This is to remind you that you are currently debugging so all the variables you see are "local." You can now click on the various debugging menu items (Step, Step into, Step out, etc.) and see what they do. MATLAB pauses at the first breakpoint in the program. In the Editor, a green arrow just to the right of the breakpoint indicates the pause. The program does not execute the line where the pause occurs until it resumes running. Now you can go back to the command window to check the values of vector bb by typing in:

k>>bb

bb =

6 7 8 9 10

To remove a breakpoint, click on the "red dot" that represents the breakpoint (in the left column). To remove all breakpoints, run >> dbclear all.

There are many functions built into MATLAB. One that will be useful here is **`fliplr`**, which is a one step way of time reversing a signal. Try this with the bell sound.

Another function that we created for you is timescale.m, which you can use to speed up or slow down a signal. Download it and give it a try. Notice that it also changes the pitch of a sound -- why?

Download the function fade.m, make sure you save it as `fade.m`. Start MATLAB, and go to the directory where you saved the function. You can see and change your current directory at the top of the MATLAB screen. Enter "help fade" at the MATLAB prompt. If you did everything correctly, you should see the help text (in the `.m` file) in response to `help fade`. Notice that we've now added a new command to MATLAB that can be used as if it were a built-in function.

Enter the following commands at the MATLAB prompt:

```
>> time = 0:.01:1;
>> y = cos(time .* pi .* 25);
>> plot(time, fade(y));
```

You can see in the plot that fade does indeed fade-out the cosine wave. You can use this function on audio signals as well.

## Exercise 1

### Fader

1. Modify the fade.m function so that you can adjust the slope of the ramp which will affect the level of the fade. Change the last command line to y = (1 – level*(t)) .* x; and save your script file. The variable `level` (which is already in the parameter list for you in the function) is used to represent the strength of the fade as a decimal fraction.
2. Set the level to 0, 0.2, 0.5, 0.7, and 1 respectively to observe the effects of changing the slope of the ramp function. For example:

   ```
   >> time = 0:.01:1;

   >> y = cos(time .* pi .* 25);

   >> plot(time, fade(y, 0.2)); % level=0.2
   ```

3. Save your plots and include them into your lab report to show that your modified fade function works.

## Exercise 2

**For** Loops

The for loop allows us to repeat certain commands. If you want to repeat some action in a predetermined way, you can use the **for** loop. All of the loop structures in Matlab are started with a keyword such as **for**, or **while** and they all end with the word **end**. The **for** loop is written around some set of statements, and

you must tell Matlab where to start and where to end. Basically, you give a vector in the "**for**" statement, and Matlab will loop through for each value in the vector:

For example, a simple loop will go around four times each time changing a loop variable, **j**:

```
>> for j=1:4, j, end;
```

## Repeater

1. Create a function that repeats a sound N times. Use a `for` loop for this. Inside the **`for`** loop (see below) you will need to concatenate sound signals. For example, if you have a vectors `x`, you can do concatenation like this:

```
>> x = [1 4 2 2 3];
>> z = x;
>> z = [z x];
z =

     1     4     2     2     3     1     4     2     2     3
```

To concatenate two matrices, they must have compatible sizes. In other words, when you concatenate matrices horizontally, they must have ***the same number of rows*** (using **',' or a space ' '**),. When you concatenate them vertically (using **';'** ), they must have **the same number of columns**.

For example, if x is a column vector, you have to use a semicolon ';'between two vectors.

```
>> x = [1 4 2 2 3]';
>> z = x;
>> z = [z;x];
```

Create a Matlab function called 'repeat.m'. The first line of your function might look like this:

```
function [ out ] = repeat(in, N)

…
Out=[];     % Return an empty array
for i=1:N

    out=[out in]; % to concatenate two vectors if "in"
                  % is a row vector. You may have to
                  % use out=[out;in] if "in" is a column vector.

end;
…
```

where 'in' is an input signal to be replicated and 'N' is the number of repeats. 'out' should be an output vector containing N replicas of 'in'.

Write a function using the 'for' loop to do the repeat operation.

2. Call the repeat function from your command window and demonstrate your repeater using N=3. For example,

```
>> load('fall');ss=repeat(fall,3);plot(ss);
```

3. Optional: Add an argument that lets you insert silence in between each repetition.

# Exercise 3

## Delay (Shift)

1. Create a function to time-delay a signal. Because we are working with digital data, you can do this by simply adding zeros (zero pad) (e.g. **delayed_y=[zeros(1,1000), y];**) in the front (note: the size of the vector containing zeros must match up the vector to be padded). Type 'help zeros' to see what it does.

2. The inputs to the function should be the signal, its sampling frequency and the amount of time-delay in seconds. The number of zeros to add will depend on the time-delay and the sample rate. The sound signals from the course webpage have a sample rate of 8,000 Hz, but it is a good coding style not to assume this and to still have the sample rate (Fs) be an input to the function in case you wanted to change it later.

```
function [ out ] = delay_signal(signal,Fs,td)
% Input arguments:
% signal: a sound signal to add delays
% Fs: sampling rate of the sound signal in Hz
% td: the amount of delay in seconds

N=fix(Fs*td);  % calculate the number of zeros needed.
…
```

3. Demonstrate that your delay function works by plotting the original and delayed signal together with the subplot command. To plot your delayed signal with the correct time axis, you need to create an appropriate time vector t. For example, `t=(0:length(s)-1)/Fs.`

# Exercise 4

# If-statement in Matlab

Sometimes it is necessary to have some code that is only executed when a condition is satisfied. In MATLAB this is done using an if statement.

The simplest form is:

```
if logical_expression
   % Code here is only executed if logical_expression is true
   % otherwise execution continues after the end statement
end
```

For example:

```
x=4;
```

```
if x >= 0
        y = sqrt(x)
end
```

There may be more than one condition.

```
if logical_expression1
   % Code here is only executed if logical_expression1 is true
   % otherwise execution continues after the end statement
elseif logical_expression2
   % Code here is only executed if logical_expression1 is false
   % and logical_expression2 is true
end
```

For example:

```
x=3;
if x < 0
        y = -1
elseif x > 2
        y = 2
end
```

## Mixer

Create a function that can mix two sounds together (sum of their magnitude values sample by sample); your function should be able to handle inputs **that are not the same length**. Please use the if statements to check their lengths and then perform appropriate operations to add two sound signals with different dimensions (lengths) but the same sampling rates. You need to do zero padding to the shorter vector.

For example:

```
N=length(x)- length(y); % To find N, use their difference in length if x is longer than y.

y=[ y, zeros(1,N)];% N is the number zeros needed. For column vectors, please try to use y=[ y;zeros(N,1)];

mixed_signal=x+y;  % to mix "x" and "y" if they have the same length.
```

The output values cannot be outside of the range (-1, 1), so you will have to re-scale them based on the maximum sample value (use the 'max'—maximum and 'abs'—absolute value functions in Matlab). One option is to re-scale the summed sound if it goes out of this range. What happens if you let the sounds go out of this range and you try to play them with the `sound` command?

To normalize a signal to the range of [-1, 1], you can try to use:

```
>> x = [1 -4 2 2 3];
>> y = x/(max(abs(x)));
```

Demonstrate that your mixer function works by playing a mix of two sounds and **plot the two sound signals and their mixed output signal on three subplots of a single graph**.

**You must include a copy of each source code file you wrote. You can copy and paste your function into a Word file instead of submitting separate .m files.**