ESE 150 – Lab 03: Data Compression

LAB 03

In this lab we will do the following:

1. Use the Arduino A2D you built in Lab 1 to capture the following waveforms: "square wave" and "triangle wave" in the audio frequency range
2. Import sampled data into Matlab
3. Compress data by reducing sample rate and quantization levels and listen to the results
4. Encode the data using the Huffman Encoding Algorithm
5. Compare the compression ratios for each of the waveforms

Background:

Compression is the concept of storing information using as little computer storage space as possible. It allows us to get the most out of our hard drives and flash drives and allows us to transmit large data files more quickly. It centers around the idea of eliminating redundancy in data and storing only what is necessary to reconstruct the original data.

Recall from lecture that compression algorithms fall into two large categories: Lossy and Lossless. A lossy algorithm will reduce the data to be stored (compress it), but the original signal that was compressed cannot be fully restored. In cases where a lossy algorithm is acceptable, the loss of that data must be acceptable (as it may be in the case when compressing audio data). A lossless algorithm will compress the data (by reducing redundancy in the data file). However, when it is de-compressed, the original data will be fully restored, so nothing will be lost.

In this lab you'll explore lossless compression (through the Huffman Coding Algorithm).

In lab today, you'll sample 2 waveforms: a square wave and a "triangle" wave. You'll then import these sampled waveforms into Matlab and apply lossless compression to them and see what type of results you achieve. The hope is for you to see how compression works and the impact of lossless compression.

ESE 150 – Lab 03: Data Compression

<u>Prelab: Introduction to Matlab</u>

- Matlab is a commonly used software packages in Engineering.
- In ESE 150 we'll use Matlab quite a bit to provide you with intro to Matlab (as you will use this software throughout your engineering career!).
- Matlab is just another programming language, and you should be able to transfer your skills from Java (CIS110, CIS120) to Matlab with modest effort. This lab will help get you started on making the transition.
- Matlab stands for "Matrix Laboratory".
- Central to Matlab is the idea that every piece of data is a Matrix!
- ***If you are comfortable with Matlab, you may jump to Step 9.***

There are 3 options for obtaining or running Matlab. You can run in Ketterer or Detkin lab, run remotely, or download/pickup copy for laptop. The instructions for each option are listed below.

Option 1: Computers in Ketterer or Detkin lab

- Matlab is already installed on all the computers in both labs. Simply search for the program in the Start menu.

Option 2: Remote Access into your SEAS Account

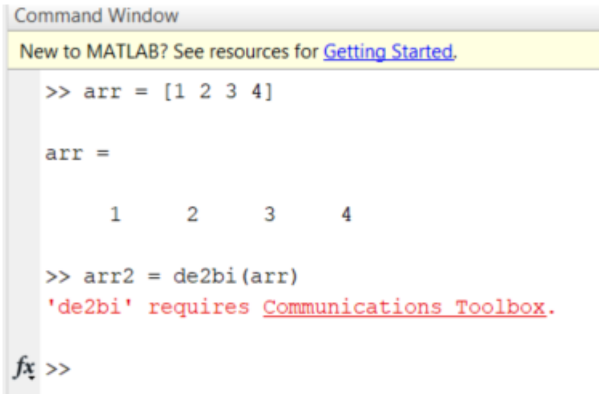
- Click on the following link to find instructions on how to remote access into your SEAS account from your personal computer. https://www.seas.upenn.edu/cets/answers/virtualLab.html
- Once you are logged into your account, search for Matlab in the Start menu.

Option 3: Download Matlab onto Personal Computer

- Click on the following link to find instructions on how to setup Matlab on your personal computer. https://www.seas.upenn.edu/cets/software/matlab/student/
- Download Communication System Toolbox in addition to the Matlab 2019b version software. It is available on https://www.mathworks.com. This toolbox is required to execute a command that you will use in the upcoming lab.
  - If you're not sure you have Communication System Toolbox, open up Matlab. In the command window, type the following commands:

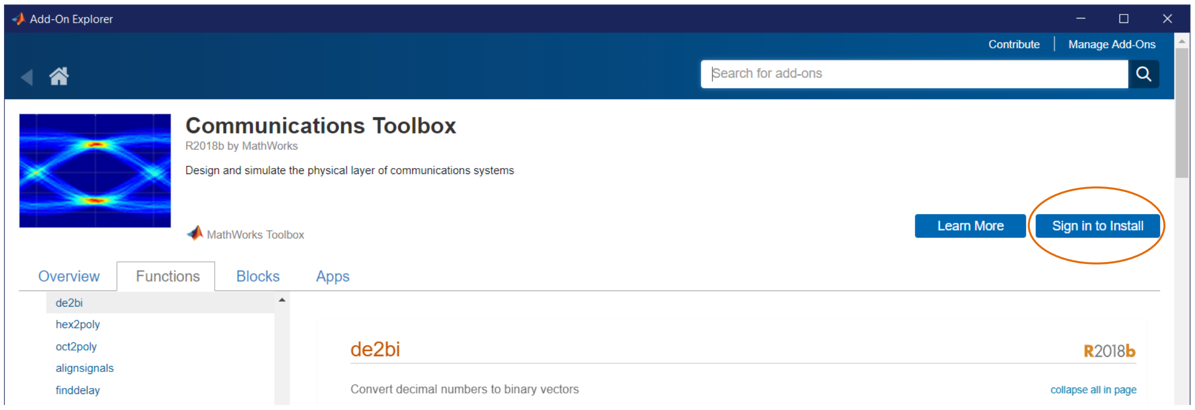
```
arr = [1 2 3 4]
arr2 = de2bi(arr)
```

  - If your computer doesn't have Communications System Toolbox, you will see the following message:



```
Command Window
New to MATLAB? See resources for Getting Started.
>> arr = [1 2 3 4]

arr =

     1     2     3     4

>> arr2 = de2bi(arr)
'de2bi' requires Communications Toolbox.

fx >>
```

  - Click on the underlined portion that says "Communications Toolbox" and a window will open that will let you download the needed software. Click "Sign in to Install", login with your SEAS Matlab account, and install it:

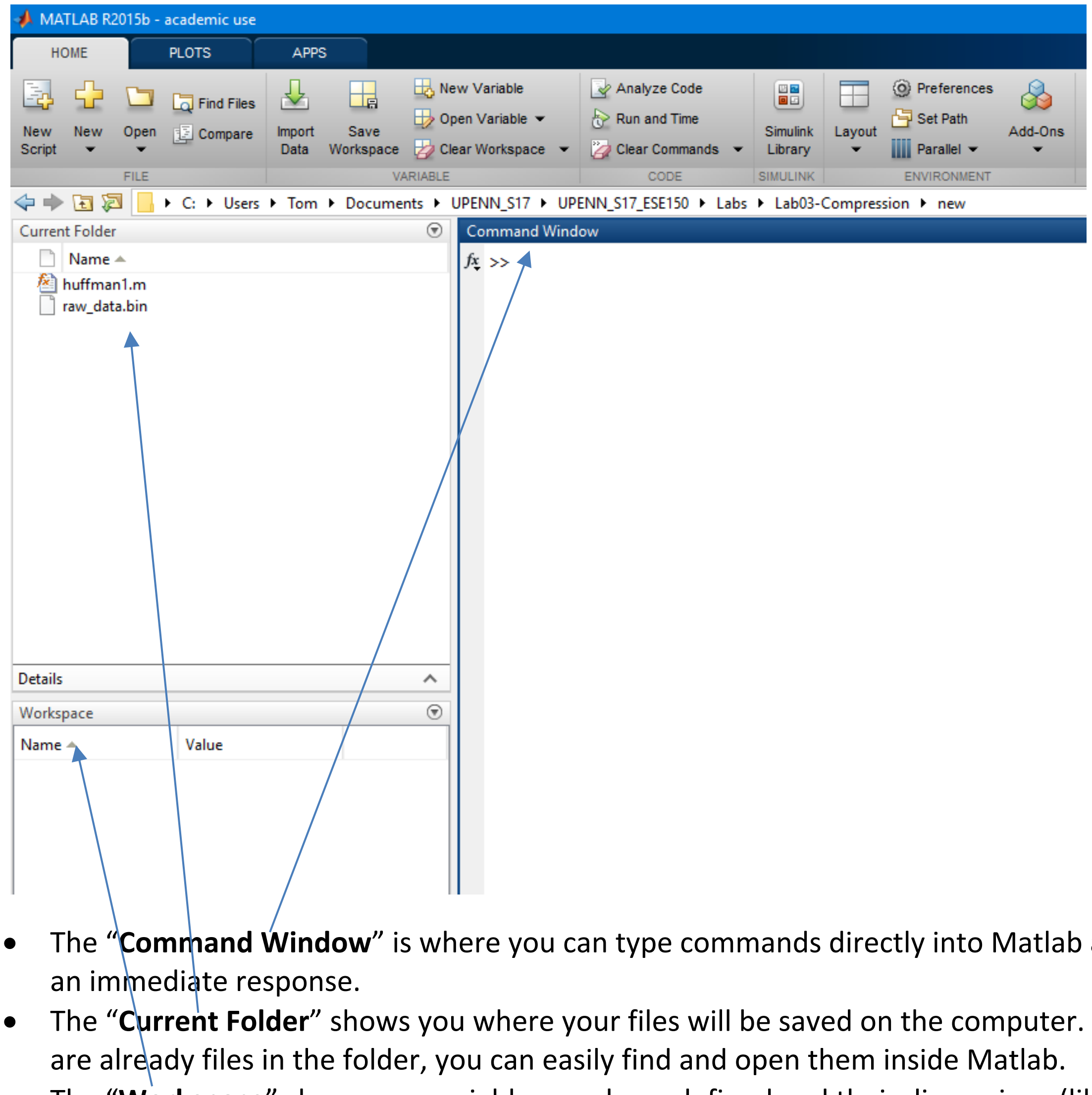


Communications Toolbox
R2018b by MathWorks
Design and simulate the physical layer of communications systems

de2bi
Convert decimal numbers to binary vectors

- If you encounter any issues while installing Matlab, feel free to come to a TA's office hours for help.

1. Open up Matlab.
2. The main screen looks as follows:



- The "**Command Window**" is where you can type commands directly into Matlab and get an immediate response.
- The "**Current Folder**" shows you where your files will be saved on the computer. If there are already files in the folder, you can easily find and open them inside Matlab.
- The "**Workspace**" shows you variables you have defined and their dimensions (like a 2x2 matrix for instance).
  - Anytime you create a variable in either the command window or in a Matlab script, it will be added to the workspace.
  - These are the "values" Matlab knows about at any given time.

3. Creating and manipulating matrices in Matlab:

a. Create a 2x3 matrix named "test" by typing the following in the command window:

```
test = [ 1 2 3 ; 4 5 6 ]
```

i. Notice this creates a matrix with dimensions: 2 rows x 3 columns.
ii. When typing numbers separated by a space, you are adding the numbers to the same row.
iii. When typing a semicolon (;), you are creating a new row of numbers.
iv. Look in the "Workspace" window to see that "test" exists.

b. Print the matrix to the screen by typing:

```
test
```

i. Notice it prints the matrix out to the screen.

c. Print out the element in the matrix in the 2nd row, 3rd column by typing:

```
test(2,3)
```

i. Notice that in Matlab, indices start with the number 1.

d. Print out the entire second row by typing:

```
test(2,:)
```

e. Print out the 3rd column only by typing:

```
test(:,3)
```

f. You can assign the 3rd column to a new matrix by simply typing:

```
test2=test(:,3)
```

This sequence is illustrating some of the specialized Matlab syntax for matrices that is likely different from what you've seen in other programming languages.

4. Generating and Plotting data in Matlab (you must try all the following commands):

a. You can have Matlab generate matrices for you, try the following:

```
x=(0:10)
```

i. This generates 1 row matrix from 0 to 10, with 11 linearly spaced points.
ii. In Matlab, a matrix with only 1 row and more than 1 column is called a vector.

b. You can have Matlab randomly generate a matrix with dimensions of your choosing:

```
y=rand(1, 11)
```

i. This generates a 1 by 11 matrix containing random numbers between 0 and 1

c. Matlab can also easily plot data for you, it takes in matrices to its plotter of course!

```
plot(x, y, '-o')
```

i. Notice, it connects the data points (marked with an o) for you!
ii. If the dimensions of x and y do not agree, it will give you an error.

5. Matrix Math:
    a. You can multiply every element of a matrix by a number quite easily, try:
    ```
    2*test
    ```
        i. This command won't change the matrix *test*, you would have to assign it to itself or a different matrix if you wanted that to happen:
        ```
        test=2*test
        ```
    b. You can "transpose" a matrix by adding an apostrophe after the matrix's name:
    ```
    test'
    ```
        i. Notice this makes the rows the columns!
    c. You can multiple two matrices together (only if they have the same inner dimensions), example:
    ```
    test3=test*test'
    ```
        i. test * test won't work (that's 2x3 * 2x3), the inner dimensions don't agree.
        ii. test * test' will work (that's a 2x3 * 3x2), the inner dimensions: (3) agree.

6. Getting help in Matlab
    a. If you need help using a command in Matlab, type **help *command***. *Example:*
    ```
    help linspace
    ```

7. Getting Help outside of Matlab
    a. The Matlab website (mathworks.com) has documentation on functions and syntax, such as for-loop and if statements.
    b. Often the easiest way to get information on a Matlab function is to google search Matlab and the function itself.
    c. Try it out: Look up documentation on the matlab function `tabulate`.
    You will use this function during the lab.

<u>Getting Familiar with Matlab Scripts</u>

- So far, we have been typing commands into the command window to see what Matlab does with the various commands.
- However, we don't want to have to type in the same commands manually over and over.
- To solve this issue, we can store a series of commands in something called a script.
- First, let's create a simple script and run through the commands one by one.
  - This will help us become more familiar with what to do when our code doesn't work in the way we want (debugging).

8. First, let's create a Matlab script!

a. In Matlab, go to the "Home" tab, click on the "New" button, then click "Script".

b. You will now see a blank text file open, called "Untitled".

c. Type the following code:

```
clear

x = 5
y = 10
z = 8

sum = x
sum = sum + y
sum = sum + z

myArr1 = [1 2 3 4; 5 6 7 8]
myArr2 = [5 3 7 4; 1 0 9 7]

myArr1 = myArr1 * 2
sumArr = myArr1 + myArr2
```

d. Save the file to your computer as "myFirstScript.m".

e. Take a moment to read over the script and understand what it does:

i. We are creating multiple variables and, over the course of the script, changing the values of the variables.

ii. For example, the value of the variable *sum* starts as *x*, then we add *y*, and finally we add *z.*

iii. In the end, *sum = x + y + z*, but we did this in multiple steps so we can see how *sum* changes over time.

iv. As a sidenote, the `clear` function deletes all variables that are currently in the workspace.

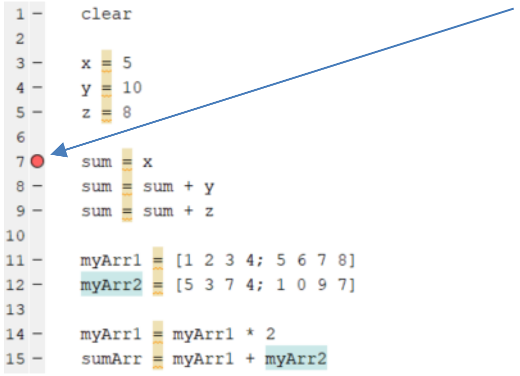
f. Now let's run the script:

i. In the command window, type `clear` to remove all old variables

ii. In the editor tab at the top of the Matlab window, click "Run". Now, your workspace should be full of new variables—these are the *final* values after the script finishes.

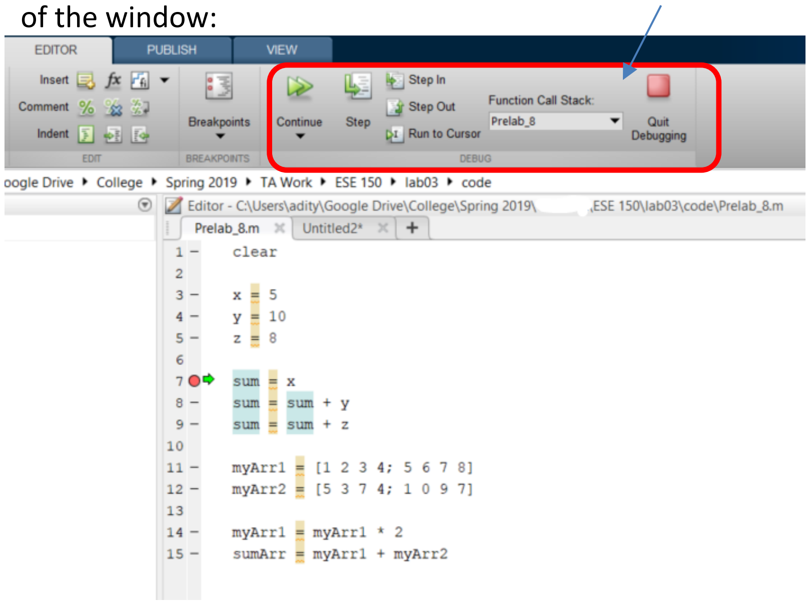
9. What if we want to see how the variables change as our program runs?

- **a.** This is where "debugging" comes in—we can look closer at what Matlab is doing with our code to see if there is a line where we are making an error.
- **b.** To debug our Matlab code, we need to set "breakpoints" in our code.
  - i. Breakpoints are just parts of our code where we have told Matlab to "pause" running and give us a chance to see what has happened so far
  - ii. Fortunately, setting and using breakpoints isn't too hard!
  - iii. To set a breakpoint, *make sure you have saved your Matlab script!*
  - iv. Then, click on the "—" right beside the number of the line of code you want to debug; the "—" will turn into a red dot.
    1. For example, here I have set a breakpoint at line 7:



```
 1 -   clear
 2
 3 -   x = 5
 4 -   y = 10
 5 -   z = 8
 6
 7 ●   sum = x
 8 -   sum = sum + y
 9 -   sum = sum + z
10
11 -   myArr1 = [1 2 3 4; 5 6 7 8]
12 -   myArr2 = [5 3 7 4; 1 0 9 7]
13
14 -   myArr1 = myArr1 * 2
15 -   sumArr = myArr1 + myArr2
```

    2. Now, when I hit "Run", Matlab will stop **before** running line 7.
  - v. Your turn: **set a breakpoint at the line `sum = x`—your window should look like mine, but your line number might be different.**
- **c.** Now that we've set a breakpoint, let's see what Matlab does:
  - i. First, let's clear the workspace by typing `clear` in the command window.
  - ii. At the top of the editor, press "Run".
  - iii. Matlab should stop running and the following options will appear at the top of the window:



oogle Drive ▸ College ▸ Spring 2019 ▸ TA Work ▸ ESE 150 ▸ lab03 ▸ code

Editor - C:\Users\adity\Google Drive\College\Spring 2019\ ,ESE 150\lab03\code\Prelab_8.m

Prelab_8.m × Untitled2* × +

```
 1 -   clear
 2
 3 -   x = 5
 4 -   y = 10
 5 -   z = 8
 6
 7 ●⇨ sum = x
 8 -   sum = sum + y
 9 -   sum = sum + z
10
11 -   myArr1 = [1 2 3 4; 5 6 7 8]
12 -   myArr2 = [5 3 7 4; 1 0 9 7]
13
14 -   myArr1 = myArr1 * 2
15 -   sumArr = myArr1 + myArr2
```

d. Now that we opened the debugging options, let's "step" through each line:
   i. Look at the workspace—we should see only *x*, *y*, and *z.*
   ii. Notice that the variable *sum* isn't there yet—that's because we stopped **before** running the line: `sum = x`
   iii. Click the "Step" button at the top of the window:
      1. Matlab will run the next line of code.
      2. Now, you should see "sum" in the workspace, and its value should be 5, since we set it equal to *x*.
   iv. Continue to click "Step" and see how older variables change over time and newer variables are added to the workspace.
   v. If at anytime you want to stop stepping and just let Matlab finish the program, click "Continue".
      1. If there are no more breakpoints in your code, the script will finish running as normal.
      2. If there is another breakpoint, Matlab will again offer you the option to step through the code, or just continue.

e. Debugging will be very useful in later labs that use Matlab, so if you have any questions, make sure to ask a TA!

10. The following 3 examples will teach you how to write scripts, plot figures, and print data to a text file in Matlab.

- `a.` Click on the button "New" and select "Script". An untitled blank script will appear above the Command Window.
- `b.` Once you name and save your new script, you should see the file appear in the "Current Folder" panel (left most panel of Matlab). The text above the script displays the path directory and current working environment.
- `c.` The first example involves creating a line plot for a single line.
  - i. In Matlab, all variables need to be initialized. Create an array called "x1" that contains 100 zeros as starting values. (Hint: Use the zeros() function. Remember you can lookup information about MATLAB functions [step 6 and 7].)
  - ii. Create a for-loop that loops from 1 to 100. (Hint: it's probably useful to see an example of a MATLAB for loop; look it up as you did in step 7.) Use the mod() function to determine if the current number is even or odd. If the result of the mod() function indicates that the current number is even, assign the current number to "x1" at the index corresponding to the current number. If the result of the mod() function indicates that the current number is odd, assign the value "0" to "x1" at the index corresponding to the current number.
  - iii. Plot the array "x1". Save or take a screenshot of the resulting plot.
- `d.` The second example involves plotting a sine and cosine on the same plot.
  - i. Create an array "x2" that contains 1000 values between (-2 * pi) and (2 * pi). (Hint: Use the function `linspace`().)
  - ii. Create an array "y1" that takes the sin of "x2" values.
  - iii. Create an array "y2" that takes the cosine of "x2" values.
    1. While you can do this with a for loop; a useful MATLAB function for operations like this is `arrayfun` (Hint: again, look it up).
  - iv. Plot "y1" and "y2" (each vs. "x2", so 2 separate curves) on the same graph. (Hint: It should require only one line of code.) Add the command "figure" before your new plot command to create a separate window from the previous example. Save or take a screenshot of the resulting plot.
- `e.` The third example involves printing the results from Example 2 (y1 and y2) to a text file.

i. Here is the general code:

```
% print to file

fid=fopen(file_name.txt','w');

fprintf(fid, '%f %f \n', [variable1 variable2 …]');

fclose(fid);
```

Make the necessary modifications to your script. Once you execute the code, a text file with your chosen file name should appear in the "Current Folder" panel. The text file and script should be in the same folder.

11. In lecture, you learned about Huffman Coding.

Watch this video for a more in-depth overview of how Huffman Coding Works:

https://www.youtube.com/watch?v=ZdooBTdW5bM

Read this article to see how we implement this:

http://nerdaholyc.blogspot.com/2014/01/a-simple-example-of-huffman-coding-on.html

Prelab Checklist:

1) Matlab Code from Step 10
2) Plots for Example 1 and Example 2 (10.c, 10.d)
3) Text File from Example 3 (10.e)

Lab Procedure:

Lab – Section 1: Gathering Samples for Various Signals

- In Lab 1 you used your Arduino to act as an A2D and sample a perfect sinewave from the lab function generator.
- In this section you'll use your Arduino A2D setup to sample a square wave and a triangle wave.

1. Generate a 300 Hz triangle wave using the function generator:
    a. If you've forgotten how, refer to lab 1, but here are some highlights…
    b. On the function generator select: "waveform" set it to **triangle.**
    c. Click on channel, set the "output load", to "high z".
    d. Set the following parameters for the wave: **300Hz, 5 Vpp, 2.5V offset.**
    e. Check the data on the oscilloscope before continuing to the next part.
2. Set the Arduino up as an A2D, and sample the signal from the function generator:
    a. If you've forgotten how, refer to lab 1, but here are some highlights…
    b. We're going to sample at our fastest rate to acquire lots of samples: 5000 Hz
        a. *This will make the dataset ripe for compression! We are heavily oversampling!*
    c. Here is a reprint of the code you'll need in the Arduino:

```
int incomingAudio[800];
int startTime;
int run = 1;

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600) ;
}
void loop() {
  // put your main code here, to run repeatedly:
  if(run){
    for(int i = 0; i < 800; i++) {
      incomingAudio[i] = analogRead(A0); // read from A0
      delayMicroseconds(XXX) ; // FIX ME!
      // you must fix the line above to sample at 5000 Hz
    }
    for(int i = 0; i < 800; i++) {
      Serial.println(incomingAudio[i]);
      }
    run = 0;
  }
}
```

    d. Capture the samples from the Serial Output Window.
    e. Copy the samples into a column in a new spreadsheet.
       Note: your samples should be between 0 and 1023 (check them!)

3. Repeat steps 1 and 2, but this time change the function generator to output a waveform: SQUARE. Copy the samples into the next column in your spreadsheet.
    a. ***<u>You should now have 2 columns in your spreadsheet, each column should have 800 rows; the #'s in each column should vary from 0 to 1023 (maximum)</u>***
    b. *DO NOT CONTINUE WITH THE LAB if your data doesn't make sense!*
    c. *Save your spreadsheet for submission.*
4. *Since each wave has a frequency of 300 Hz, and you've sampled at 5000 Hz, how many cycles of each waveform do you have? Include your answer in your writeup.*

Lab – Section 2: Importing Sampled Data into Matlab

- In this section you'll learn how to import data into Matlab, namely the sampled data from your Arduino
- You'll use your skills from the Prelab to manipulate and plot the data in Matlab.

1. In the spreadsheet you used to store your samples, highlight the 800 samples for **only** the triangle wave, and "copy" them (using <ctrl> C).
2. In Matlab create an empty matrix, named "**samples_triangle**" by typing the following:

```
samples_triangle=[]
```

3. In the "Workspace" pane, click on your newly created *samples_triangle* matrix
   a. This will bring up an editor, that looks a little like a spreadsheet.
4. Paste your spreadsheet data into Matlab:
   a. RIGHT click on the very first square at the top left, row 1, column 1.
   b. Choose "Paste Excel Data"; once pasted close the variable editor.
   c. Notice in the "workspace" pane, *samples_triangle* is now an 800x1 dimension matrix.
5. Using what you've learned the prelab, plot your samples in Matlab:
   a. Create a new matrix called: *samples_triangle_voltage.*
   b. In that matrix, use matrix math to convert the samples from 0-1023, to 0-5 Volts.
   c. Have Matlab create a vector that represents the "times" the samples were taken.
   d. Plot Voltage vs. Time.
   e. Use the "help" tool for the following commands: title, xlabel, ylabel to add appropriate labels to your plot… (e.g. – ***300 Hz Sine Wave*** is a nice title!).
   f. Use the magnifying glass in the plot window to show only 4 cycles of your wave starting at time 0.
   g. Take a screenshot of the plot for submission.
6. Repeat steps 1-5 for the square wave (create a matrix named: *samples_square*)

   NOTE: you can repeat commands in Matlab by pressing the UP arrow

7. Call over a TA to check the data samples you collected.

<u>Lab – Section 3: Variable Group Lossless Compression: Huffman Coding</u>

- Recall the Huffman coding algorithm is a lossless and variable group size algorithm.
- The basic idea of the Huffman algorithm is to first analyze the data one wishes to compress and determine the frequency of occurrence of all the symbols one wishes to compress.
- The more likely a particular symbol occurs in your dataset, the shorter the binary code is that will represent that symbol in its compressed form.
- After the frequency of each "symbol" we want to compress is determined, one encodes (or compresses) the data by replacing each symbol with a more efficient representation of that symbol. These more efficient symbols are unique.
- We'll use Matlab to help us apply the Huffman coding algorithm to our samples.

- We're going to start by applying the Huffman coding algorithm to a small set of data, before we apply it to our samples, so you can understand how it works!

1. Begin in Matlab by creating a vector with only 10 samples as follows:
   ```
   my_samples = [0; 1; 2; 2; 3; 5; 5; 5; 1023; 1023]
   ```
   a. We can see that "5" occurs most frequently (3/10 of the time…30% of the time).
   b. We can see that "2" occurs 20% of the time, same for "1023".
   c. 0 and 1 are the least frequently occurring.
   d. For Huffman, "5" should receive the least bits since it occurs most frequently.
   e. Now, how do we get Matlab to analyze this for us?

2. Use Matlab's tabulate() function to analyze the frequency of occurrence of each symbol
   a. Note, our "symbols" are: 0, 1, 2, … 1023 (unlike lecture, where our symbols were a,b,c)
   ```
   a=tabulate(my_samples)
   ```
   b. Look at what came out, a 6x3 matrix (named a).
      i. The first column is our symbol, the second column is a count of how many times it occurred, the 3rd column is percentage of occurrence!
3. Extract just the probability of each symbol's occurrence (the 3rd column):
   ```
   prob = a(:,3)
   ```
4. Normalize those probabilities to be out of 1 (instead of out of 100%):
   ```
   prob = prob/100
   ```
5. Extract the "unique" symbols in your dataset (0, 1, 3, 5, 1023):
   ```
   symbols = a(:,1)
   ```

6. Build the "Huffman dictionary" for the symbols in your dataset according to their probability of occurrence:
   ```
   dict = huffmandict (symbols,prob)
   ```

   a. Matlab essentially creates a binary tree for your dataset, but it's in matrix form.

b. This function "huffmandict()" actually builds the Huffman dictionary, think of the dictionary as a table you can lookup the binary number for a given symbol.

7. Let's examine the Huffman dictionary:
   a. Double click "dict" in the workspace.
   b. Let's look up the symbol "0" in the dictionary:
      i. Look in Row 1. In the first column, you will see 0, the symbol that we compressed. In the second column, you will see a matrix of four numbers ([0, 0, 0, 1]) that represents the number 0.
      ii. This means that 0 is represented by the binary number 0 0 0 1 after Huffman compression is performed.
   c. In your lab report, write out the binary number that will be used for each symbol in "my_samples".

8. Now let's use the dictionary to compress the samples:

```
samples_compressed = huffmanenco(my_samples,dict)
```

   a. Huffmanenco() examines each of your samples, looks them up in the dictionary, and replaces them with their binary equivalent.
   b. Notice, the first four numbers in the compressed data. They are 0, 0, 0, 1—that represents the Huffman encoded version of 0 that we saw in the dictionary!
   c. Can you manually decode the compressed data using the dictionary? It must be a perfect match to your samples.
   d. Lastly, how many bits (0s and 1s) does your compressed data need? (HINT: look at the matrix size of "samples_compressed" in the workspace!)

9. Let's try decompressing…going backwards:
   `samples_decompressed = huffmandeco (samples_compressed, dict)`
   a. Huffmandeco() just reverses the process
   b. *samples_decompressed* should be equal to your *my_samples* matrix!
   c. Verify these two matrices are equal by using the function `isequal()` (type help to learn how to use it).

10. We had to type a lot of commands to get the data compressed—let's make that automatic:
    a. We will be making our own function in Matlab!
    b. Click on the "New Script" button at the top of the Matlab screen.
    c. Enter all the commands you just used above in steps 2-8 (skip step 7).
    d. At the very top of the file add a line like this:

```
function [samples_compressed, dict] = compress_huff(my_samples)
```

    e. The name of our function is "compress_huff".
    f. It takes as input a matrix: my_samples
    g. It returns two output matrices: samples_compressed and dict, which is the dictionary used to compress the samples.
       i. MATLAB functions can return multiple values; this may be a feature you haven't seen before.
       ii. By assigning the variables used in the function return signature (that's the **`[samples_compressed, dict]`** in this case), you are returning them.
       iii. As before, search for "MATLAB function" to see documentation and examples.
    h. At the very end of the file, add the following line:

```
end
```

    i. Lastly, save the file with the exact same name as your function: *compress_huff.m*
    j. Now, whenever we call the function, we will receive two matrices back—our compressed data (consisting of 1s and 0s) and the dictionary used to compress the original data.
    k. How do you call your function? From the command window, type:

```
[your_compressed_values, your_dictionary] =
    compress_huff(your_matrix_to_compress_here)
```

all on one line.

For example, if you are compressing square wave data, you might do the following:

```
[compressed_sq, dict_sq] = compress_huff(samples_square_voltage)
```

This will create two Matlab matrices—one called compressed_sq, containing the Huffman encoded data, and another called dict_sq, containing the Huffman dictionary for the square data.

Lab – Section 4: Compressing Wave Data with Huffman

- In this section you'll apply your compression function to the two signals you've sampled from Section 1.

1. Now that you have a function that will compress your data, try it out on your samples!
   a. Run your compress_huff function on your `samples_triangle_voltage` and `samples_square_voltage` matrices.
2. Calculate the compression ratio (binary bits out / binary bits in) for each sample set: square and triangle
   a. In our case, each voltage sample value takes 10 bits.
   b. Include your two compression ratios in your lab writeup.
   c. **Which one gets the best ratio? Why do you think that is?**
      i. Hint: run `tabulate`() on the voltage samples for each set and look at the frequency distribution.
3. Now, use the Huffman dictionary returned by your compress_huff function, along with your compressed data, to decompress the triangle waves:
   a. Remember, we used a command to decompress the compressed data.
   b. Also, the `isequal` function can help you compare matrices.
   c. Report whether your decompressed data matched your original samples.
4. Before leaving lab, demonstrate your compression/decompression to a TA. This is the Lab Exit Check-off.
5. ***Optional:*** run one last experiment. In the command window, create a Huffman dictionary for the triangle wave. Now, apply that dictionary to the square wave samples…does it do as good a job at compressing the data?
   a. What does this tell you about the Huffman dictionary? Is a generic one useful?
   b. It's possible this will fail in encoding. If so, why does it fail?
6. Make sure all Matlab code, graphs, and data collected are accessible to both partners.
7. Clean up your lab station before leaving.

ESE 150 – Lab 03: Data Compression

Postlab: Questions

1. How effective would Huffman Compression be for each of the following. Explain why? (in some cases the the answer depends on personal habits, so your explanation is central to what the "correct" answer is.)
    a. World Cup Soccer (Football to non-Americans) scores
    b. NFL (American Football) scores
    c. What you pay for lunch each day
    d. Ambient outdoor temperature when you arrive at SEAS quad each weekday
    e. Hours of sleep you get each night (quantized to whole hours)
    f. Digits of π (pi)
    g. Digits of sqrt(2)
2. Beyond audio, what are applications where Huffman Coding is applicable and likely to be effective? [at least 3]
3. What are applications where Huffman Coding is not likely to be effective? [at least 3]

HOW TO TURN IN THE LAB

- Upload a PDF document to canvas containing:
    - All items in the Prelab Checklist
    - Matlab code for your `compress_huff` function
    - Answers to the following questions from the lab sections:
        - Section 1, Question 4
        - Section 3, Question 7c
        - Section 3, Question 8d
        - Section 4, Question 2b
        - Section 4, Question 2c
    - If you answered these optional questions, include your responses as well:
        - Section 4, Question 5a
        - Section 4, Question 5b
    - Plots for voltage samples (Section 2—for triangle and square)
    - Answers to postlab questions
    - Label the sections and tell us what questions you are answering!
- Upload a separate .XLSX to canvas
    - All sample data for each waveform: square, sawtooth