# 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”, “sine wave”, and “sawtooth 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 memory as possible. It allows us to get the most out of our hard drives and transmit large data files as quickly as possible. It centers around the idea of eliminating redundancy in data and storing only what is absolutely necessary to reconstruct the original data when necessary.

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, nothing will be lost.

In this lab you’ll explore both types of compression: lossy (CS&Q algorithm) and lossless (through the Huffman Coding Algorithm). If you do not remember these algorithms, review the lecture material before beginning lab.

In lab today, you’ll sample 3 waveforms: a sine wave (like in lab 1), a square wave and a “sawtooth” wave. You’ll then import these sampled waveforms in Matlab and apply both lossy and lossless compression to them and see what type of results you achieve. The hope is for you to see how compression works, and see the impacts of lossy vs. lossless compression.

***Prelab: Introduction to Matlab***

- 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 7.***

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.

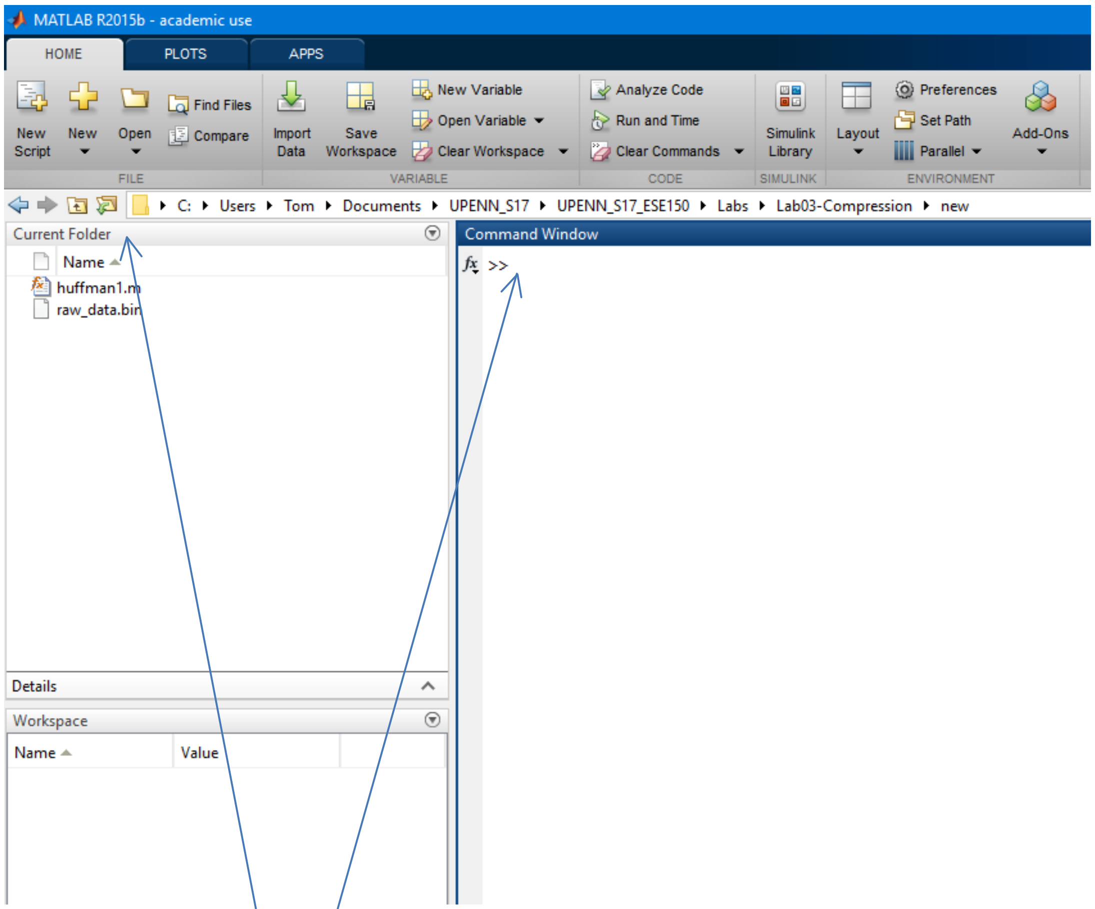
Option 3: Download Matlab onto Personal Computer

- 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 2017 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 execute the following commands, it will prompt you to download the toolbox and provide an appropriate link:

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

  Follow the instructions and it will be added.
- 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 and can allow you to open up files inside Matlab
- The “Workspace” shows you variables you have defined and their dimensions (like a 2x2 matrix for instance)

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. Look in the “Workspace” window to see that “test” exists
   b. Print the matrix to the screen by typing:
      ```
      test
      ```
      i. Notice it spits 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)
      ```
   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)
      ```
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)
      ```
   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. 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. (Hint: The Matlab website (mathworks.com) has documentation on functions and syntax, such as for-loop and if statements. Easiest way is to google search Matlab and the function itself.)
        i. In Matlab, all variables need to be initialized. Create an array called "x1" that contains 100 zeros as starting values. (Hint: Matlab has a built-in function.)
        ii. Create a for-loop that loops from 1 to 100. 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. Use the plot() function to 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 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.
        iv. Plot "y1" and "y2" 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 the general code. 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.

Prelab Checklist:

1) Matlab Code from Step 7
2) Plots for Example 1 and Example 2 (7.c, 7.d)
3) Text File from Example 3 (7.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 sinewave (again), a squarewave, and a triangular (sawtooth) wave

1. Generate a 300 Hz sinewave 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 **sine**
    c. Click on channel, set the "output load", to "high z"
    d. Set the following parameters for the wave: **300Hz, 4 Vpp, 2V 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 way 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.
4. Repeat steps 1 and 2, but this time change the function generator to output a waveform: SAWTOOTH. Copy the samples into the next column in your spreadsheet.
   a. ***You should now have 3 columns in your spreadsheet, each column should have 800 rows; the #'s in each column should vary from 0 to 1023 (maximum)***
   b. *DO NOT CONTINUE WITH THE LAB if your data doesn't make sense!*
5. *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.*

***<u>Lab – Section 2: Importing Sampled Data into Matlab</u>***

- 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 last section to manipulate and plot the data in Matlab

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

   ```
   samples_sine=[]
   ```

3. In the "Workspace" pane, click on your newly created *samples_sine* 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_sine* 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_sine_voltage*
   b. In that matrix, use matrix math to convert the samples from 0-1023, to 0-4 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.
6. Repeat steps 1-5 for the square wave (create a matrix named: *samples_square* )
   ***a. <u>NOTE: you can repeat commands in Matlab by pressing the UP arrow</u>***
7. Repeat steps 1-5 for the sawtooth wave (create a matrix named: *samples_saw* )

***Fixed Group Lossy Compression: CS&Q***

- Recall CS&Q compression is a lossy fixed group algorithm
- CS&Q = Coarser Sampling AND/OR Quantization
- We'll see the effect of CS&Q in terms of acceptable loss of our data

***Lab – Section 3:* Re-sampling Data**

1. Recall, that you sampled your 300 Hz signals at 5000 Hz, (5000 samples/sec).
2. If we delete every other sample from our sampled data, we'd cut our sampling rate in half
3. In Matlab, it is easy to delete elements from a matrix, try the following:
   make a copy of the sine_matrix:
   ```
   samples_sine_2500 = samples_sine
   ```
   Delete every other element of our new matrix:
   ```
   samples_sine_2500(2:2:end)=[]
   ```
   *What does this command do?*
   It starts at the 2nd element of the "samples_sine_2500 matrix" and replaces every 2nd element with a [] or "blank" matrix. So it effectively deletes every other element from the matrix.
4. Plot the new data (voltage vs. time). Have we lost any necessary data?
   a. Recall from lecture what the "sampling rate" must be for a 300 Hz signal
   b. Could we go further? How low could we go and still be able to reconstruct our original signal without harm?
   c. Show the matlab commands necessary & the plotted data
   d. Since we over-sampled, this form of compression can be quite helpful!
5. With your most compressed form of the data, how much have you reduced the data compared to the original 5000 Hz sampled data?

## *Lab – Section 4:* Re-quantizing Data

1. Recall that the samples from the Arduino can range from 0-1023; how many bits would be required in binary format to represent 1024 levels? What does that mean for its resolution?
2. As we did in the spreadsheet last week, we can use Matlab to perform quantization.
3. The first step towards doing this, is to convert our samples into binary:

   ```
   samples_sine_binary = de2bi(samples_sine, 'left-msb')
   ```

   ***de2bi*** converts decimal #'s into binary #s, arranging so 'left' is the most significant digit

4. Does your new data line up with the samples? Compare the data in your matrices to be sure. Print out the first element of ***samples_sine***. Then print out ***samples_sine_binary***, are they equivalent? Ensure that they are before continuing!
5. *How many bytes does your matrix contain? Recall 8 bits = 1 byte*

6. Re-quantize to 8-bit resolution:

   ```
   samples_sine_binary_8b=samples_sine_binary(:,1:8)
   ```

   *Can you tell what this has done? Look carefully at (:,1:8), think about what it means!*

7. *How many bytes does your re-quantized matrix contain? Recall 8 bits = 1 byte*
   a. What is the compression ratio? **Compression ratio= # bits out / # bits in**

8. Convert the samples back to decimal form by using the **bi2de**() function:

   ```
   samples_sine_8b = bi2de(samples_sine_binary_8b, 'left-msb')
   ```

   a. If your data is now in 8-bit form (we've dumped the two least significant bits), what is the new range of your data? It will no longer go from 0-1023

9. Plot the 8-bit sampled data to the screen (voltage vs. time)
10. Listen to the original and the 8-bit sampled versions!
    a. Yes, Matlab can be a D2A for us!! It can play sounds out over the sound card of the computer, by typing the following:

    ```
    soundsc(samples_sine_voltage, 5000, [0 5])
    ```

    Note, your signal must be converted to voltage. Also, you must tell it the sampling frequency (5000Hz) and the range of voltage [0 5] volts.
    b. Put the above in a **FOR** loop (*type help* for to learn how to use for loops), make it loop 25 times so you can hear the difference
    c. Compare it to the 10-bit sampled data
11. Make a 3-bit version of your sampled data ($2^3$=8 levels), how does that sound?
12. ***Repeat the above for the square wave…how many bits does it need???***

## ***Lab – Section 5: Variable Group Lossless Compression: Huffman Coding***

- 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.

1. Consider the data we are compressing…samples ranging from 0-1023.
    a. We realize no matter what symbol we have: 0, 1, 2, … 1023, we encode them in binary with 10-bits. So even for the number 0, we must use 10 zero's to encode it. This way, when we read the data from a file, we read it in groups of 10, so we know how to interpret each sample in our file.
    b. This waste of bits, makes this type of data ripe for compression! Perhaps 10 bits isn't necessary to store each sample!
2. Watch this video on Huffman Coding Works:

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

**Read this article to see how we actually implement this:**

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

- 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 actually 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 class 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. Let's 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 # for a given symbol.

7. Let's examine the Huffman dictionary:
   a. Let's look up the symbol "0" in the dictionary:
   b. Notice, from the matrix, that symbol "0" is in the 1st row of the dictionary matrix:

   ```
   dict{1,:}
   ```

   c. The {} brackets are used, because "dict" is a matrix of other matrices! This asks it to print out the matrix in the 1st row
   d. Inspect the result:

   ```
   ans =

     0

   ans =

   0     0     0     1
   ```

   e. This indicates that the symbol "0" will be encoded with binary: 0 0 0 1
   f. In your lab report, write out the binary # that will be used for each symbol

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 very first #'s in the compressed data? They are 0, 0, 0, 1 ; that's the # 0!
   c. Can you manually decode the compressed data? It must be a perfect match to your samples!
   d. Lastly, how many bits does your compressed data need? (HINT: look at the matrix size in "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. Can you make your own function in Matlab?
    a. Yes! Click on the "New Script" button at the top of the Matlab screen
    b. Enter all the commands you just used above in steps: 2-8 (skip step ~~6~~ 7)
    c. At the very top of the file add a line like this:

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

    d. The name of our function is "compress_huff"
    e. It takes as input a matrix: my_samples
    f. It returns as its output a matrix: samples_compressed
    g. At the very end of the file, add the following line:

    ```
    end
    ```

    h. Lastly, save the file with the exact same name as your function: *compress_huff*
    i. How to call your function? From the command window, type:

    ```
    compress_huff(your_matrix_to_compress_here)
    ```

***<u>Lab – Section 6: Compressing Music Data with Huffman</u>***

- In this section you'll apply your compression function to the three 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!
2. Calculate the compression ratio (binary bits out / binary bits in) for each sample set: sine, square, sawtooth
    a. *Remember, you know how many bits it actually takes to represent your matrix (recall the binary conversion of the uncompressed data in Section 4)*
    b. **Which one gets the best ratio? WHY do you think that is?**
3. Don't forget to decompress your data and make sure it's a match to the original (remember, it's lossless, so everything must line up!)
4. Run one last experiment. In the command window, create a Huffman dictionary for the sine 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. Imagine a Huffman dictionary for: rock/classical/country…is this a good idea?
5. Before leaving lab, demonstrate your compression/decompression to a TA for a wave of his or her choosing and answer a few questions. This is the Lab Exit Check-off.

***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? [identify at least 3]
3. What are applications where Huffman Coding is not likely to be effective? [identify at least 3]

**<u>HOW TO TURN IN THE LAB</u>**

- Upload a PDF document to canvas containing:
  - Final prelab Matlab demo (Step 7)
  - All Matlab Code
  - Answers to ALL questions in each lab section (Format: x.y (x = section and y = step number)
    - 1.5, 3.4b, 3.5, 4.1, 4.8a, 4.12, 5.8c&d, 6.2b, and 6.4a&b
  - All graphs generated
  - Answers to postlab questions
  - Label the sections and tell us what questions you are answering!
- Upload a separate .XLSX to canvas
  - All samples for each waveform: sine/square/sawtooth