ESE 150 – Lab 06: Perceptual Coding

LAB 06

In this 2-week lab we will do the following:

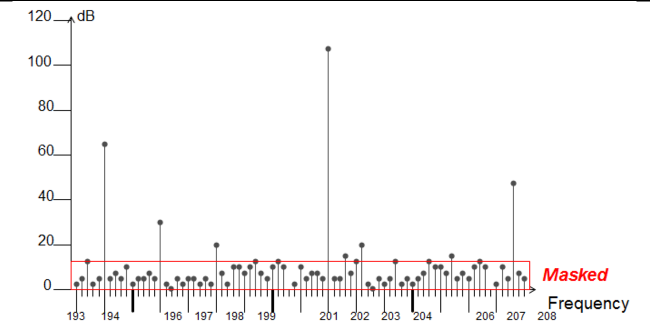
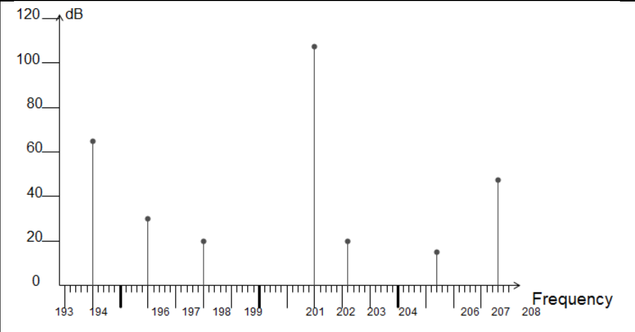
1. Use Matlab to sample three audio files in .WAV format (3/11)
2. Plot the samples in time and frequency domain before any compression (3/11)
3. Analyze the DFT of the samples and drop masked signals
    a. Manually (3/11)
    b. Write a function to drop masked signals (3/18)
4. Perform Huffman Encoding before and after dropping samples (3/18)
5. Analyze efficiency of your compression algorithm (3/18)

You will work in the same teams for the entire lab (both weeks).

Background:

In this lab, you will apply perceptual coding. Recall that perceptual coding is using your knowledge of psychoacoustics (how human's interpret sound) to encode your audio data in a more succinct way. In lab 5 you performed some classic psychoacoustic experiments. One in particular is frequency masking.

Recall that frequency masking is when one tone is so loud that it "drowns out" other tones of the same or nearby frequencies? The pictures below show an example of that happening. Notice that some tones are so loud that other tones that are not as loud are "masked." In the picture on the right, we've used that principal to "drop" or zero-out certain tones. Since they are "masked" no one will be able to hear them (thanks to psychoacoustics), so instead of storing these tones, we drop them (or zero-them-out) to make it so there is less data to store. It's a form of lossy compression that is used in the MP3 algorithm.

Dropping Samples using psychoacoustic analysis	
	
BEFORE psychoacoustic analysis is performed	AFTER psychoacoustic analysis is performed

Remember also that masking can be different for different frequency bins. Recall the table below that shows the 24 critical band frequency bins human hearing is divided into.

Number	Center frequencies Hz	Cut-off frequencies Hz	Bandwidth Hz
		20	
1	50	100	80
2	150	200	100
3	250	300	100
4	350	400	100
5	450	510	110
6	570	630	120
7	700	770	140
8	840	920	150
9	1000	1080	160
10	1170	1270	190
11	1370	1480	210
12	1600	1720	240
13	1850	2000	280
14	2150	2320	320
15	2500	2700	380
16	2900	3150	450
17	3400	3700	550
18	4000	4400	700
19	4800	5300	900
20	5800	6400	1100
21	7000	7700	1300
22	8500	9500	1800
23	10 500	12 000	2500
24	13 500	15 500	3500

You may drop data above band 24.

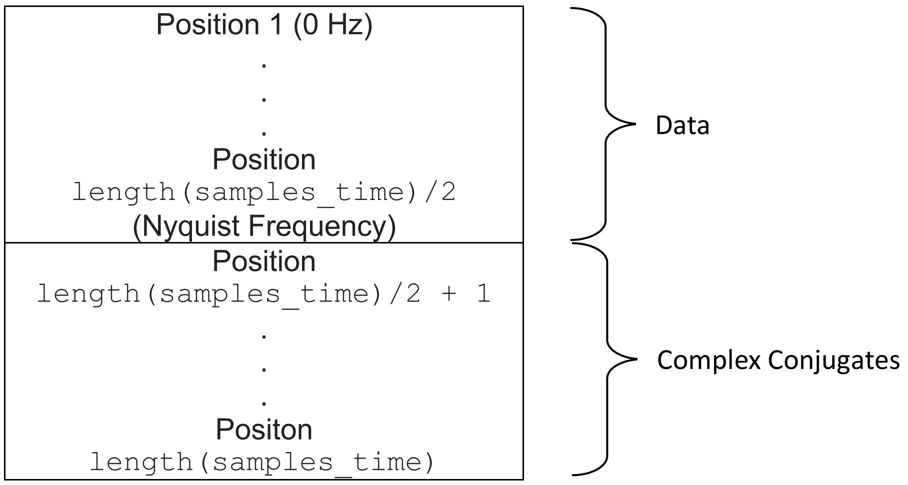
Today in lab, you'll take time-sampled audio and go through it frequency-bin by frequency-bin and perform your own psychoacoustic analysis. You will look for signals that are masked. If they are masked, you will drop them (or zero-them-out). Then you'll convert your frequency domain data back to the time domain and listen to your modified audio file to see if you can notice the difference! The more tones you drop due to masking, the smaller your audio file will become.

Once you've completed the task of dropping masked tones, you'll run the Huffman encoding algorithm on it and see just how much "redundancy" is now in your audio file that can be reduced with standard compression; further reducing your audio sample's file size. By applying a lossy and lossless compression scheme to your original audio file, you'll have gone through a very similar process as to what is done in an MP3 file! Today we'll just concentrate on frequency masking, but other tricks (like temporal masking) are used in the true MP3 algorithm. But this will give us a good sense of what MP3 is doing and how it achieves the compression ratios that it boasts!

MATLAB fft and ifft

For this lab it is very important to understand how MATLAB's fft and ifft functions work. Make sure to read the documentation for these functions, specifically focusing on what their inputs and outputs are.

An important concept here is that of symmetric arrays. In order to speed up computation, fft outputs (and ifft takes an input) a symmetric array, where the length of the array is equal to the length of the `samples_time`, the first half of the array contains the Fourier transform data from 0 Hz to the Nyquist frequency, and the second half of the array contains the complex conjugate of the first half (position `length(samples_time)/2 + 1 + i` contains the complex conjugate of the data in position `i` for all `1≤i≤length(samples_time)/2`)


Position 1 (0 Hz)
Position length(samples_time)/2 (Nyquist Frequency)
Position length(samples_time)/2 + 1
Positon length(samples_time)
Data
Complex Conjugates

This works fine if your code applies fft to `samples_time` and immediately apply ifft to the result, but we want to be able to change the frequency domain data before converting back to the time domain. So, we will need to figure out a way to edit the first half of the fft output while maintaining the complex conjugates. We will do this as follows:

1. Set `samples_freq` equal to the output of fft applied to `samples_time`
2. Create a new variable `samples_freq_data` that stores the value of `samples_freq(1: length(samples_freq)/2)`
3. Perform any operations to this new array as long as the length does not change, given that position 1 in the array corresponds to 0 Hz and position `length(samples_freq)/2` corresponds to the Nyquist frequency.
4. Now compute the complex conjugates by iterating through every element of `samples_freq_data` and storing the complex conjugate of position `i` in position `i` of `samples_freq_conj`
5. Now concatenate `samples_freq_conj` onto the end of `samples_freq_data` to get an array that can have ifft applied to it to get back the time domain data.

It is also highly recommended to add the tag `‘symmetric’` when performing ifft on data that has been modified to ensure the output is real.

ESE 150 – Lab 06: Perceptual Coding

Prelab

- Read through lab
- Make sure you read the background. This knowledge will be very important for the lab.
- Questions:
    - Explain why and how zeroing out (setting values to 0) frequencies in a signal will lead to compression during Huffman encoding?
    - Particularly, if we zero out 50% of the frequencies, what will happen to the efficiency of the compression? How about 90%?
    - What is an alternative to zeroing out values? What are the advantages and disadvantages of this method? Would it result in better compression rates?
    - Research the following MATLAB functions or syntax items and write a short (1-2 sentences) explanation:
        - … (Yes, an ellipsis is an important part of MATLAB syntax)
        - %% vs % (Focus on explaining MATLAB sections)
        - quantizenumeric()
        - real()
        - imag()
        - ceil() and floor()
        - vertcat()
        - zeros()
        - abs() for complex values
        - conj()
        - quantile() [added 3/11]
- Programming:
    - We want to write a function `calc_conjugate(samples_freq_data)` that automates the process in steps 4 and 5 of the fft and ifft background above. This function takes as input frequency data of size n and returns an array of size 2n where the first half of the array contains the data from `samples_freq_data` and the second half contains the complex conjugates of the first half. If you have any trouble with complex numbers in MATLAB make sure to post a question on Piazza or ask a TA.
- Bring headphones to lab.

Lab Procedure – Lab Section 1 – Using Matlab to Sample .WAV files

- In this section of the lab, we'll import a .WAV files directly into Matlab! This will give you "perfect" PCM audio data that you can then perform a psychoacoustic analysis on.
    - a .WAV file contains time-sampled audio similar to what you collected in lab on the arduino in previous weeks
- We will plot the time and frequency representations of the sound in the .WAV file. This section uses 'song 1' as a reference to verify your plot_time() and plot_dft() functions you wrote in earlier labs.
- This section serves as preparation for the remainder of the lab and ensure that your plot_time() and plot_dft() functions are correct.
- Items highlighted yellow are required for you report

1. Download the 3 .WAV files from last lab and put them in a directory where you are running Matlab:
    a. For example, create a directory called: c:\temp\lab6
        i. If you can't create that folder on our lab computers, you can choose any other location you'd like, as long as you know exactly where it is!
    b. Download the .WAV files into c:\temp\lab6
    c. Start Matlab
    d. In Matlab's command window, type:
    ```
    cd c:\temp\lab6
    ```
2. Import the .WAV files directly into matlab:
    a. In Matlab's command window type:
    ```
    [samples_time, samp_rate]=audioread('song1_300-600Hz.wav')
    ```
    b. This has imported the PCM quality .WAV file into a matrix called: samples_time
    c. It has also determined the sampling rate that the .WAV file was created with
    d. *Look carefully at the value of "samp_rate", is it what you expect for a .WAV file?*
3. Plot in the time domain:
    a. The function "audioread" brings the samples into Matlab
    b. Ensure your plot_time() function has the following parameters:
    ```
    function [samples_time] = plot_time (samples_time, samp_rate, …
    start_time, end_time, figure_n)
    ```

    This should plot your converted samples in the time-domain, starting at *start_time*, ending at *end_time*, and using figure number *figure_n*. Also, make sure you do not use a hardcoded sample rate anywhere in your code, it should be parameterized on *samp_rate*.
4. Then, call your function plot_time() on your imported .WAV file (song1) to plot 2 periods of the 600 Hz wave.
    a. save this labeled plot for your report

5. In Lab 4 you created a function called "plot_dft". Modify it to have the following parameters and functions as explained below:
```
function [samples_freq, freq_mag] = ...
    plot_dft(samples_time, samp_rate, figure_n)
```
   a. Input: samples_time – should be the output of audioread
   b. Input: samp_rate – should take in the sampling rate: e.g.: 44100
   c. Output: samples_freq – should be the unmodified output of the fft() function
      i. Recall, you take the absolute value of it, don't return that version! That's only the magnitude, not the phase information.
   d. The function itself should plot only the valid frequencies: up to ½ sample rate
   e. Important note: Use the stem() function instead of plot(). The stem() function plots the discrete sequence of your input as stems that extend from the x-axis. The inputs of the stem() function are identical to those of the plot() function.
6. Plot in the frequency domain:
   a. Use your "plot_dft()" function.
   b. For the input of your plot_dft() function, you should input the raw sound file (from audioread()).
   c. Save this plot for your report, you may want to save multiple zoom levels so you can see that the spikes are at the correct locations.
   d. If all your functions are working properly, do you see the two spikes you expect?
   e. Modify your function to also return the corresponding magnitudes to sample_freq. Make sure the indices of sample_freq and magnitudes are perfectly matched.
7. Listen to the audio just to make sure…
   a. Plug your headphones into the lab PC
   b. Use Matlab to perform a D2A on the data you sampled from the .WAV file! Type the following in the MATLAB command line:
```
soundsc(samples_time, samp_rate)
```
      Put your headphone on and listen! Does it sound right??
   c. Verify that this step is working before proceeding.
8. Turn both of these graphs in with your report!
   a. Make sure the 2 functions are correct before continuing, have a TA look at them if you are unsure.

<u>Lab Section 2 – Performing a psychoacoustic analysis</u>

- In this section you'll actually examine the data in the frequency domain and look for samples to drop.
- For this section, the removal of frequencies from certain sections of the song will be done manually. This will serve as a basis for writing the actual algorithm in the next section.

2. In Section 1, you imported "song1" which is a test file to make sure your plot_dft() function is working
3. Create a new file named *mainsection2.m* Write all of you MATLAB script code in this file, <u>the majority of your code should be in this file so you can put it into your report.</u>
4. Use the code from Section 1 to import "song2" into Matlab (writing this code in mainsection2.m rather than in the command line). Use your plot_dft() function to plot the frequencies in song2. This will serve as a visual to show what frequencies are present in the signal and the corresponding magnitudes. Listen to the audio before continuing to ensure it sounds correct.
    1. You'll also need to save this plot for your lab submission
5. In this step, you will be performing psychoacoustic analysis on a smaller section of song 2.
    1. Write MATLAB code to isolate a consecutive sequence of 500,000 samples starting from the 100,000th sample of song 2. What is the duration for the 500,000 samples in seconds?
        i. Call this extracted sequence `samples_time_extract`
    2. Use your plot_dft() function to obtain a frequency plot for the 500,000 samples.
        i. Save this plot for your lab submission.
    3. Do you see any masking opportunities?
        i. Zoom in on the 2nd critical band. Refer to the critical band frequency bin chart in the introduction section to determine the lower and upper frequencies for the 2nd critical band.
        ii. Are there any frequencies that can be dropped from the 2nd critical band? If so, where and why?
    4. Remember, you are looking at the complex magnitude of the frequency domain <u>BUT</u>, recall that your function: plot_dft() actually returns the complete frequency domain data
    5. For tones that are being masked, zero-out those samples in the matrix that was returned by your plot_dft() function (meaning, set them to zero). Use the magnitudes to determine a threshold for dropping frequencies. Try to drop 50% of the samples.
        i. Remember you must use the method discussed in the background/prelab to handle the complex conjugates required by ifft
        ii. Create a new variable that stores the value of `samples_freq(1:length(samples_freq)/2)`
        iii. Drop frequencies from this new array given that position 1 in the array corresponds to 0 Hz and position `length(samples_time_extract)/2` corresponds to the Nyquist frequency.
        iv. Create a new variable `samples_freq_extracted_dropped` and set it equal to the function you wrote in the prelab that calculates the complex conjugates required for ifft (do not ifft your data yet through).
    6. You've done this for just the 2nd frequency bin. Repeat parts 5 for critical bands 3-8.

7. Let's see if deletion affected the quality of the sound.

6. Convert your modified frequency-domain data back to the time domain
   1. As you'll recall, there is an "inverse" DFT function in mathematics and in Matlab as well
   2. Assuming you've made a matrix called: `samples_freq_extracted_dropped` that contains the frequency domain data, but with the samples that were masked set to zero, convert it back to the time-domain by typing:
      ```
      samples_time_dropped = …
      ifft(samples_freq_extracted_dropped, 'symmetric')
      ```
   3. What you'll get back in "samples_time_dropped" will be your audio file but back in the time domain. Matlab has performed the inverse DFT for you.
   4. Play this back the two sets of samples (i.e. before and after dropping; `samples_time_extracted, samples_time_dropped`), can you hear a difference? Did the dropping of the masked signals change anything?
   5. Show your TA your plot showing the tones dropped and your before and after frequency samples and answer some questions. This is your Exit Checkoff for March 11th lab.

Lab Section 3 - Automating psychoacoustic analysis

- In the previous section, you only dropped frequencies from a small selection of critical bands for a small portion of the sound wave. The goal of this section is to drop frequencies from all bandwidths for the entirety of the song.
- This will build off of your lab 5 prelab.
- <u>Make sure to read through the entire rest of the lab before beginning this section</u>

1. Remove masked frequencies as in Section 2 but in an automated way.
   a. Visually inspecting the fft() created matrix is going to be laborious
   b. You can automate this process in Matlab
   c. You will need to create a separate Matlab function that contains the logic in your plot_dft() function and analyzes the transformed signal for masking, frequency bin by frequency bin
   d. Call this function "drop_samples" and give it the following ins/outs

```
   function [windowed_freqs] = drop_samples ( samples_time,
drop_fraction, window_size )
```

   e. Create a new file called `mainsection3.m` to test your function
   f. Here's an overview of the logic:
      i. The input variable `window_size` will control how many samples are allocated to each window. We are using the 'window' variable to isolate smaller sections of the signal, similar to Section 2. To simplify the logic, `window_size` will only be a power of 2.
      ii. The input variable `drop_fraction` should be a value between 0 and 1 to specify what fraction of the frequencies in a band should be dropped.
      iii. The output `windowed_freqs` should contain the frequencies after dropping such that position `1` to position `window_size/2` should contain the frequencies that represent the first window of `samples_time` and position `window_size/2 + 1` to position `window_size` should contain the frequencies that represent the second window of `samples_time`
         1. Remember from the background information that we only need to store/drop frequencies from the first half of the fft output, which is why each window takes up half of `window_size` rather than the entire `window_size`
      iv. Take an excerpt of `samples_time` with length equal to `window_size`
      v. Obtain the Fourier transform of each windowed section.
      vi. For each transformed section of the song, go through all 24 bandwidths to delete masked frequencies based on the `drop_fraction` and the magnitudes. The logic will be similar to section 2 of this lab.
         1. Thresholds used for dropping are likely different for each band.
         2. MATLAB function quantile() may be useful to get the threshold values.

vii. Have your function also print out the original number of frequencies produced by all uses of fft (not including aliased data) and the total number of frequencies dropped.

viii. Make sure you store the data in `windowed_freqs` as discussed earlier

ix. Try dropping different fractions of frequencies and see if the number of frequencies dropped is what you expect.

g. This will take your time. You'll have to think through the code on your own, and you want to spend some time experimenting with code, the window size, the thresholds, and perhaps other parameters you identify.

i. Using MATLAB's debugging tools and breakpoints will be very helpful for this section. A tutorial for breakpoints was provided in lab 3.

<u>Lab Section 4 – Using Huffman Encoding</u>

- In this section of the lab, you'll use logic similar to your Huffman function from your previous labs to compare the compression ratios before and after dropping samples.

1. Create a new function called `compress_freqs` and give it the following ins/outs:

```
function [compressed_real, compressed_imag, dict_real, …
avglen_real, symbols_real, dict_imag, avglen_imag, …
symbols_imag] = compress_huff(windowed_freqs)
```

2. The logic for this function is as follows:
   a. Split windowed_freqs into the real and imaginary components
   b. Use the logic from the Huffman function from lab 4 to compress the real and imaginary components separately.
   c. Make sure to return all of the outputs above, they are all outputs from the process Huffman compression and will be used later to either decompress the data or compute compression ratios
3. <u>Test this function on small excerpts of the song (128 or 256 samples) because unmodified long excerpts will take a very long time to run</u>. Make sure the run `drop_freqs()` on the excerpt before running `compress_freqs()`. Later parts of this lab will explain how to modify long excerpts so Huffman compression runs faster.

Lab Section 5 – Decompressing

- In this section we provide code to perform the inverse of the Sections 3 and 4 code to retrieve time domain data that we can listen to.

    1. We are providing you with a function called `decompress_to_time` that will take the output of your Huffman function as input:

```
function [samples_time] = decompress_to_time(compressed_real, ...
  compressed_imag, dict_real, dict_imag, window_size)

%Decompress the real component
uncompressed_real = huffmandeco(compressed_real, dict_real);

%Decompress the imaginary component
uncompressed_imag = huffmandeco(compressed_imag, dict_imag);

%Combine the real and imaginary components
windowed_freqs = uncompressed_real + sqrt(-1)*uncompressed_imag;

%Allocate an empty vector for vertcat
samples_time = zeros(0, 1);

%Frequency window size is half the window_size
freq_window_size = window_size/2;

%Calculate the number of windows based on the freq_window_size
windows = length(windowed_freqs)/freq_window_size;

%Iterate through all windows
for w = 1:windows
  %Extract one window of frequencies
  freq_extract = windowed_freqs(((w-1)*freq_window_size)+1: ...
    w*freq_window_size, :);

  %Calculate complex conjugates
  ifft_data = calc_conjugate(freq_extract);

  %IFFT
  time_samples_window = ifft(ifft_data, 'symmetric');

  %Adds time_samples_window for this window to the running list of
  %time samples
  samples_time = vertcat(samples_time, time_samples_window);
end
```

2. The logic for this function is as follows:
    a. Decompress the real and imaginary components using the corresponding dictionary.
    b. Combine the real and imaginary components into one complex array.
    c. Next it iterates through each window of frequencies; remember that we are only storing half of the number of frequencies as the length of the window.
    d. Take the extract of the frequencies corresponding to the current window.
    e. This calls the function you wrote in the prelab to compute the complex conjugate data needed for ifft
    f. ifft the data and add the time domain data to the running array of time samples.
    g. This assumes the window size is a multiple of 2 and the length of the uncompressed data is equally divisible by the half of the window size.
3. For this decoding function to work, your encoding function needs to produce data compatible with it.
    a. If the output of your compress_huff function does not produce listenable audio, first try running the first few lines of the function to decompress your data and make sure it matches the data you had prior to Huffman compression (Matlab allows you to check if two matrices are equal with ==)
    b. Also make sure that the length of your data after decompression is an integer multiple of your window_size
    c. If both a and b are satisfied, and the output still does not sound correct, then the issue is likely with your code from section 3 rather than how your data interfaces with this decompression function

<u>Lab Section 6 – Putting It All Together</u>

- In this section we will use the functions we have written so far in the lab to compress a song, calculate compression ratios, and decompress the song to listen for quality loss.

1. Create a function called quantize_freqs that has the following ins/outs:
```
function [quantized_freqs] = …
quantize_freqs(windowed_freqs, word_bits, frac_bits)
```
2. The logic for this function should be as follows:
    a. Split apart the real and imaginary components of the frequency array
    b. Quantize the real and imaginary components separately based on the input `word_bits` and `frac_bits` (Hint: Look back at the functions you researched for the prelab, `word_bits` corresponds to word length and `frac_bits` corresponds to fraction length)
    c. Combine the real and imaginary components back into a single frequency array to be returned by the function
3. Create a new file called `mainsection6.m` to contain your code and tests for the remainder of the lab.
4. Create a section to contain your compression code
    a. Read in the audio from song3
    b. Choose a `window_size` that is a power of 2

   c. Take a long excerpt of the `samples_time` that is a whole number multiple of the `window_size` (Start with around 1 minute of the song and shorten this if your computer takes more than 5-10 minutes to process it)
   d. Plot this excerpt in the time domain
   e. Plot the frequencies for this excerpt (You do not need to save the output from `plot_dft`)
   f. Drop frequencies from the excerpt using your `drop_freqs()` function and the `window_size` you chose earlier
   g. Quantize the frequencies using the function you just wrote
   h. Compress the frequencies using your `compress_huff()` function.
5. Create a new section to decompress the song
   a. Decompress using `decompress_to_time()`
   b. Plot this decompressed data in the time domain
6. Set your `window_size` to 2048 and do not drop any frequencies from your song. Figure out what combination of `word_bits` and `frac_bits` will best balance sound quality and processing time. It should not take more than 5-10 minutes to compress and decompress the song. Make sure to look back at the documentation for `quantizenumeric()` to understand the meaning of `word_bits` and `frac_bits`
7. Create a new section to compute the compression ratio (# bits to store the original data divided by the # of bits to store the compressed data) of your algorithm
   a. For the original data size, you should assume the data is quantized the same as the quantization you chose in step 6.
   b. For the compressed data size, make sure to include the data required to store the dictionaries, but instead of directly measuring the size of each dictionary, try to think about what data needs to be stored for the dictionaries.
   c. Display the compression ratio
8. With a fixed window_size of 2048, fill out the following table

Percent Frequencies Dropped	# Frequencies Dropped	Compression Ratio
0%		
10%		
25%		
50%		
75%		

9. With a fixed 10% frequencies dropped, fill out the following table:

window_size	# Frequencies Dropped	Compression Ratio
1024		
2048		
4096		

10. Include any plots that were produced by your code to generate the above tables with proper labels (You only have to include two plots for the unmodified excerpt, one for the frequency domain and one for the time domain)

Post Lab Questions:

1. Why is it inaccurate to remove frequencies directly from the Fourier of the entire song? For example, why is it wrong to take the Fourier transform of the entire song and remove the 500,000 frequencies with the lowest magnitudes? (500,000 was arbitrarily chosen)
2. Provide 2 qualitative examples to why the method mentioned in the previous question is incorrect. In other words, what situations in a song could easily highlight the inaccuracy of the previous method?
3. How did changing the window size affect the resulting quality of the song after decompressing? Changing the percent of frequencies dropped?
4. Compare the time & frequency plots of your original sound wave and modified sound wave when the window_size is 2048 and 25% of frequencies are dropped.

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

- There will be no weekly lab writeup for either lab session.
- Include all items highlighted in yellow.
- Include all code you wrote. The code should be in appendices rather than the main part of the report.
- Include answers to post lab questions.
- Be sure to include all necessary plots.
- Each student will produce an individual formal lab report that is due Sunday, March 24th.
- Include the academic integrity statement indicating individual for lab report.
  - I, <*your name*> certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this report.
  - You can review the Code of Academic Integrity here: http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html
- Include a section in the lab report acknowledging contribution of your partner.
- See course web page for Formal Lab Report specification and expectations.
- See the specific rubric on canvas for the formal writeup for this lab.
- Upload a PDF document to canvas containing for formal lab report.