LAB 06

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

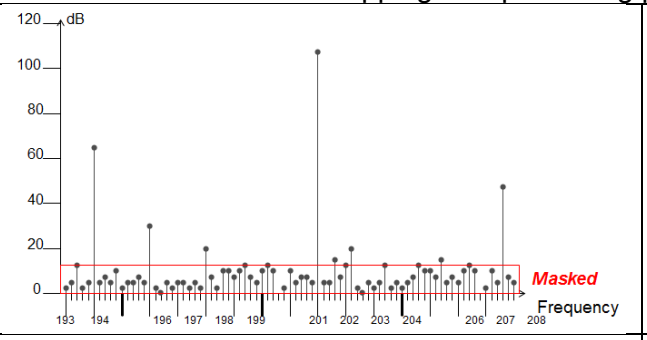
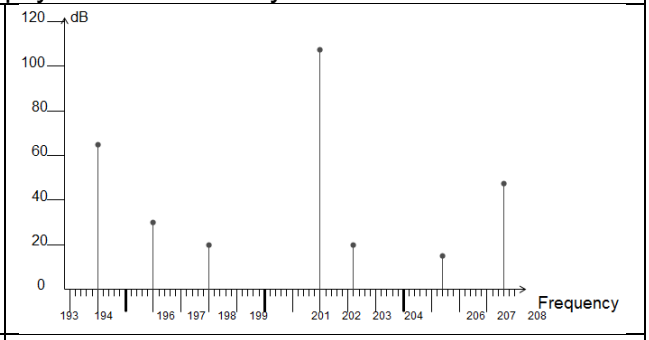
1. Use Matlab to sample three audio files in .WAV format (2/23)
2. Plot the samples in time and frequency domain before any compression (2/23)
3. Analyze the DFT of the samples and drop masked signals
    a. Semi-automatically (2/23)
    b. Write a function to drop masked signals (at least start on 2/23)
4. Perform Huffman Encoding before and after dropping samples (3/2)
5. Analyze efficiency of your compression algorithm (3/2)

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 Freq. (Hz)	Cut-off Freq. (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	10500	12000	2500
24	13500	15500	3500

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.

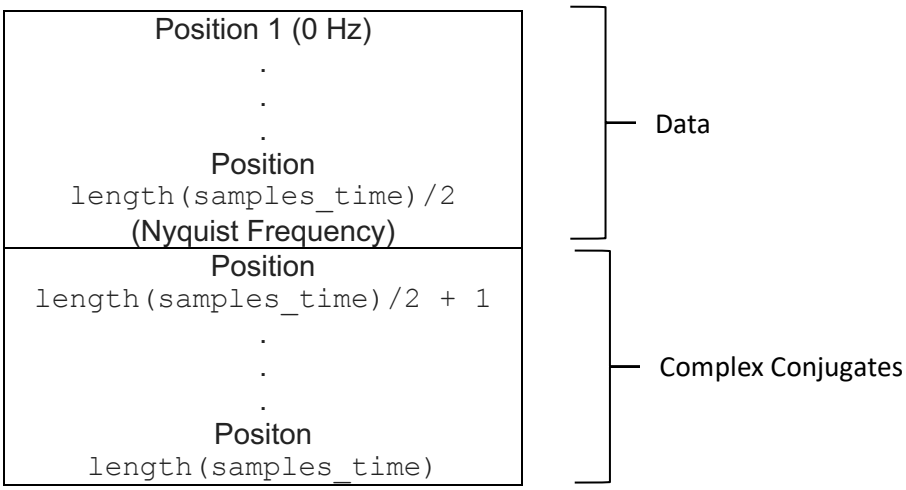
Once you've completed the task of dropping masked tones, you'll run the Huffman encoding algorithm on it and see just how much "redundancy" is now in your audio file that can be reduced

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! In this lab, 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.

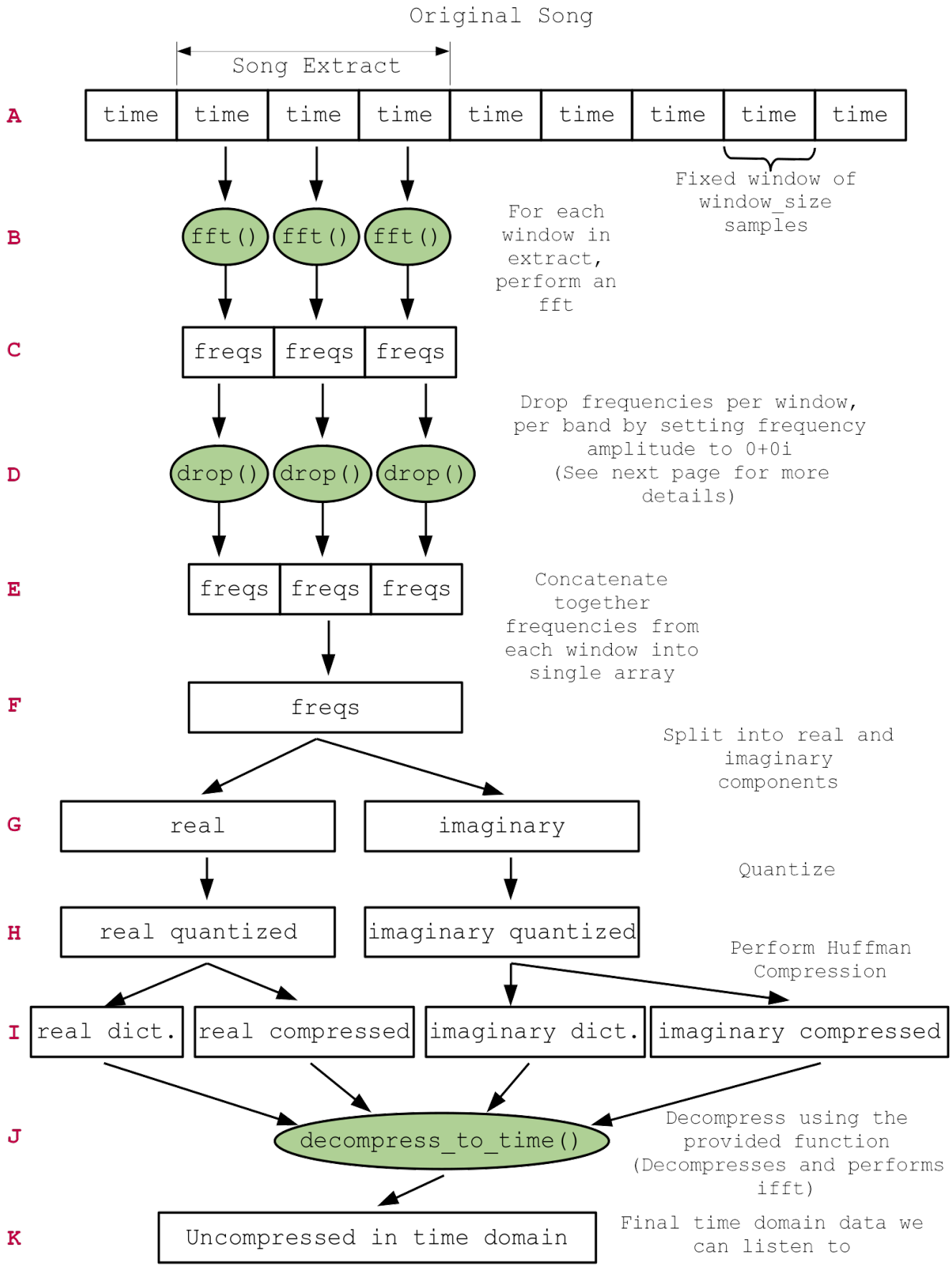
An important concept here is that of symmetric arrays. In order to speed up computation, fft outputs (and ifft takes as input) a symmetric array, where the length of the array is equal to the length of the `samples_time`, the first half of the array contains the Fourier transform data from 0 Hz to the 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 $1 \leq i \leq$ `length(samples_time)/2`)

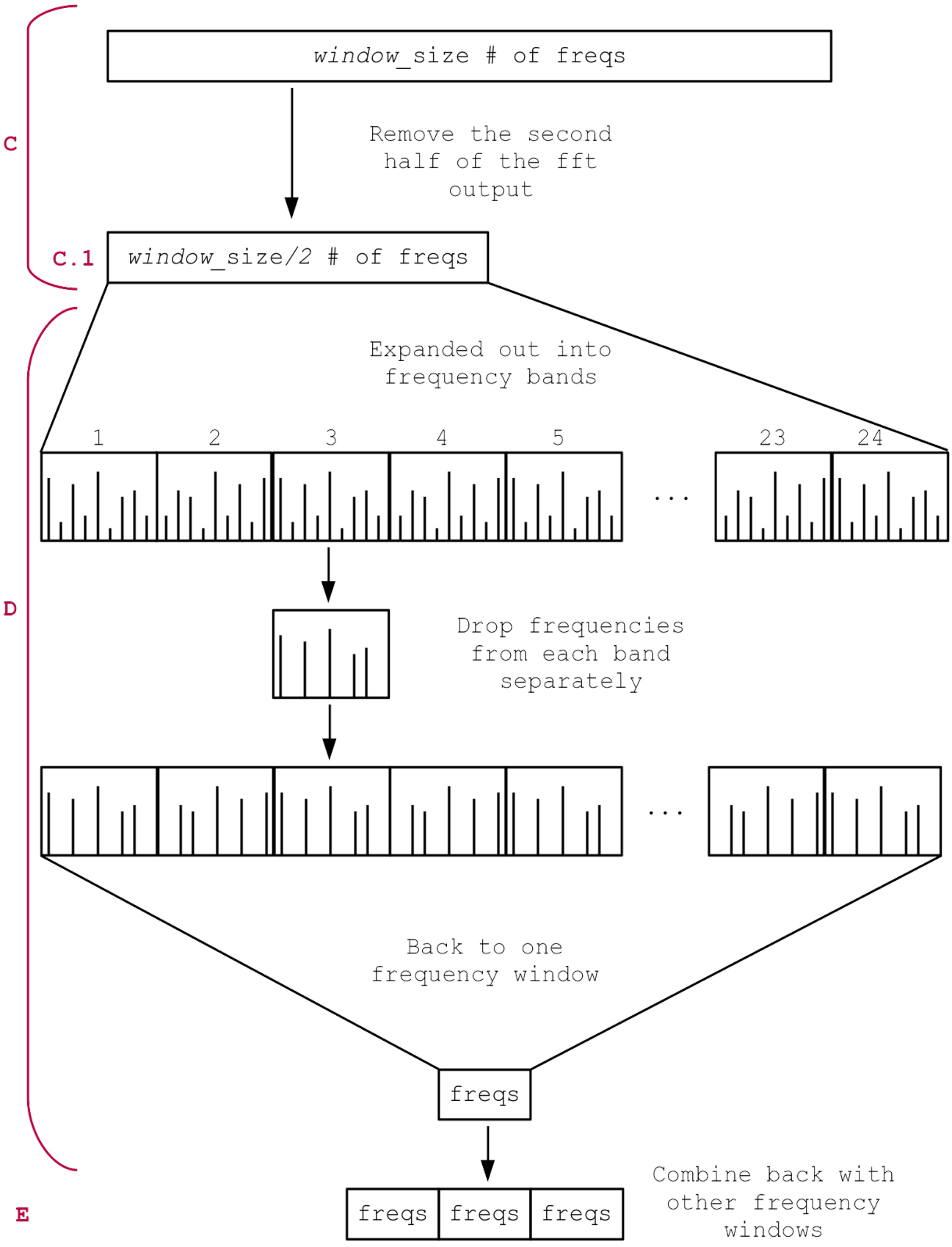


This works fine if your code applies fft to `samples_time` and immediately apply ifft to the result, but we want to be able to change the frequency domain data before converting back to the time domain. So, we will need to edit the first half of the fft output while maintaining the complex conjugates. We will do this by only using the first half of the frequencies during the compression algorithm, and then recalculating the complex conjugates when decompressing. This is handled for you in the code provided below, but make sure you understand why we are only using the first half of the frequencies. You will also need to use this code for section 2.

```
function [samples_freq_symmetric] = calc_conjugate(samples_freq_data)
    freq_conj = zeros(length(samples_freq_data), 1);
    for i = 1:size(samples_freq_data)
        freq_conj(i) = conj(samples_freq_data(i));
    end
    samples_freq_symmetric = vertcat(samples_freq_data, freq_conj);
```

The next two pages provide a master overview of the operations we will perform for compression and decompression. Refer to this diagram as you read through the lab and develop and assemble the components of your compression algorithm.


Original Song
Song Extract
A
time
time
time
time
time
time
time
time
time
Fixed window of window_size samples
B
fft()
fft()
fft()
For each window in extract, perform an fft
C
freqs
freqs
freqs
D
drop()
drop()
drop()
Drop frequencies per window, per band by setting frequency amplitude to 0+0i (See next page for more details)
E
freqs
freqs
freqs
Concatenate together frequencies from each window into single array
F
freqs
Split into real and imaginary components
G
real
imaginary
Quantize
H
real quantized
imaginary quantized
Perform Huffman Compression
I
real dict.
real compressed
imaginary dict.
imaginary compressed
J
decompress_to_time()
Decompress using the provided function (Decompresses and performs ifft)
K
Uncompressed in time domain
Final time domain data we can listen to


window_size # of freqs
Remove the second half of the fft output
C
C.1
window_size/2 # of freqs
Expanded out into frequency bands
1
2
3
4
5
23
24
D
Drop frequencies from each band separately
Back to one frequency window
freqs
freqs
freqs
freqs
Combine back with other frequency windows
E

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?
    - Assuming non-zero frequencies take 10b on average to encode, if we zero out 50% of the frequencies, what will happen to the efficiency of the compression? How about 90%? Your answer should be expressed as a quantitative result.
    - An alternative to zeroing out the dropped values is storing (Freq, amp) pairs for the non-dropped values. That is, when we zero out frequencies, the position in the array tells us which frequency the amplitude goes with. If we were to only store the non-zero values, we would also need to keep a record of the Frequency associated with each non-zero value.
        - Why might this not be an obvious win? (not always allow us to represent the non-zero frequencies with fewer bits?)
        - Under what circumstances (compression characteristics) would this require fewer bits than the zero'ing strategy?
    - Research the following MATLAB functions or syntax items and write a short (1-2 sentences) explanation in your own words (i.e. do not copy the documentation):
        - … (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()
    - Write MATLAB code using the quantile() function to identify the 25% of the elements with the largest absolute value in the following arrays. The result's order does not matter. Just filter out the bottom 75%.
        - [20 75 100 0 1 50 3 3 7 90 2 12]
        - [1 23 -2 10]

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.
- We will plot the time and frequency representations of the sound in the .WAV file. This section uses 'song 1' as a reference to ensure the plot_time() and plot_dft() functions are working correctly.
- This section serves as preparation for the remainder of the lab.
- Items highlighted in yellow are required for your report.

1. In Lab 4 you developed *plot_time()* and *plot_dft()* functions. For this lab, we provide you with slightly modified versions to ensure the plots are in the format we expect.
2. Download the 3 .WAV files from Canvas/website and put them in a directory where you are running Matlab:
    a. For example, create a directory called: c:\temp\lab6
        i. You can choose any location you want as long as you remember where it is!
    b. Download the .WAV files into c:\temp\lab6 or the directory of your choice.
    c. Start Matlab.
    d. In Matlab's command window, type:
        cd c:\temp\lab6 or cd <directory of your choice>
3. 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
        i. this is A in master diagram.
    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?

4. Plot in the time domain:
    a. The function "audioread" brings the samples into Matlab
    b. Use the provided *plot_time()* function to plot the samples in the time domain. Ensure you understand how this function works.

```
function [samples_time] = plot_time (samples_time, samp_rate, ...
start_time, end_time, figure_n)
samples_short = ...
samples_time((start_time*samp_rate+1):(end_time*samp_rate), 1);
samples_total = length(samples_short);
sample_number = (1:samples_total);
time = arrayfun(@(a) a/samp_rate, sample_number);
figure(figure_n);
plot(time, samples_short);
title('Time Domain Samples');
xlabel('Time (Seconds)');
ylabel('Voltage (Volts)');
end
```

    This function will plot your converted samples in the time-domain, starting at *start_time*, ending at *end_time*, and using figure number *figure_n*. In order to do this, the function takes a subset of the samples based on the given start and end times, calculates the corresponding times for the samples and samples vs. time.
5. Then, call the 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
6. In Lab 4 you created a function called "plot_dft". We provide a variant with the following parameters and functions as explained below:

```
function [freq_half, samples_freq, freq_mag] = ...
    plot_dft(samples_time, samp_rate, figure_n)
samples_freq = fft(samples_time);
samples = length(samples_time); % # of samples
samples_freq_new = abs(samples_freq / samples);
samples_freq_new = ...
samples_freq_new(1:round(length(samples_freq)/2.0),:) * 2;
freqs = (0:(samples - 1));
freqs = freqs * (samp_rate/(samples - 1));
freq_half = freqs(1:round(length(freqs) / 2, 0));
figure(figure_n);
freq_mag = samples_freq_new;
stem(freq_half, freq_mag);
title('Frequency Domain');
xlabel('Frequency (Hertz)');
ylabel('Amplitude (Volts)');
end
```

a. Input: samples_time – the output of audioread
b. Input: samp_rate – the sampling rate: e.g.: 44100
c. Input: figure_n – figure number to create the plot on
d. Output: samples_freq – the unmodified output of the fft() function
e. The function itself plots only the valid frequencies: up to ½ sample rate
f. Important note: Here we use the stem() function instead of plot(). The stem() function plots the discrete sequence of the input as stems that extend from the x-axis. This is better for visualizing discrete frequencies returned by fft().

7. Plot the entire song in the frequency domain:
   a. Use the "plot_dft()" function.
   b. For the input of the 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. Do you see the two spikes you expect?
   e. Notice that our function also returns the corresponding magnitudes to sample_freq.
8. Listen to the audio just to make sure…
   a. Plug your headphones into you computer.
   b. Use Matlab to perform a D2A on the data you sampled from the .WAV file! Type the following in the MATLAB command line:

```
audioplayer(samples_time, samp_rate)
```

To begin playing audio

```
play(player)
```

To stop audio

```
stop(player)
```

Put your headphone on and listen! Does it sound right??

   c. Verify that this step is working before proceeding.
9. Turn both of these graphs in with your report!

Lab Section 2 – Performing a psychoacoustic analysis

- 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.
- This section corresponds to doing steps C and D from the diagram manually.

1. In Section 1, you imported "song1".
2. Create a new file named *mainsection2.m* Write all of you MATLAB script code in this file, the majority of your code should be in this file so you can put it into your report.
3. Use the code from Section 1 to import "song2" into Matlab (writing this code in mainsection2.m rather than in the command line). Use the 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.
4. 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` (this is A)
        ii. In previous years some people had problems with running out of memory when running code for this lab. If this happens to you:
            1. Ensure you are only using a piece of the song rather than the whole thing (as per instructions).
            2. If it is still taking too long to run, shorten the length of the extract taken while writing code and debugging (e.g., use a smaller set of samples like 100,000)
    2. Use your plot_dft() function to obtain a frequency plot for the 500,000 samples.
        i. Save this plot for your lab submission.
        ii. Make sure to save the values returned by plot_dft() as `samples_freq` and `freq_mag`
    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?
            1. This is a preliminary version of C/D.
    4. Remember, you are looking at the complex magnitude of the frequency domain BUT, recall that the 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 the 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 first. Make it easy to change the percentage of frequencies dropped in the code, you will want to experiment with different values here.

i. Write a matlab function to perform this dropping. You will be applying this to other bands in later steps.

ii. Create a new variable `samples_freq_extract` that stores the value of: `samples_freq(1:length(samples_freq)/2)` where `samples_freq` is the return of plot_dft from Step 4.2.

1. This is step C.1.

iii. You should determine the frequencies in critical band 2 to drop based on the complex magnitude, but samples_freq should still be complex numbers, even after dropping frequencies.

- Hint: note that plot_dft() is returning `freq_mag` with the magnitude of the complex amplitudes of all the frequencies.
- Hint: use quantile() to determine the threshold for dropping.

iv. Drop (set to zero) frequencies (in `samples_freq_extract)` 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.

1. Look at the indices in `freq_mag` corresponding to critical band 2.
2. Drop frequencies that have a magnitude less than the dropping threshold determined in 5.iii.
3. This is a preliminary version of D.

6. You've done this for just the 2nd frequency bin. Generalize your code to work on all critical bands.

i. Create an array with the thresholds of all bands

ii. Use this array to figure out where the indices should start and stop for dropping within freq_mag (and correspondingly in samples_freq). It is suggested to create a helper function with the signature
`function [indices] = bands2indices(freq_domain, cutoff_freqs)`
that returns an array the same size as `cutoff_freqs` which indicates the indices of `freq_domain` that delimit each frequency band.

iii. Apply your generalized code to drop frequencies for all bands. Here are some tips to help you get it right on the first try:

1. Consider first creating a helper function that drops the frequencies on one band at a time:
`function [freq_mag] = drop_freq_band(in_freq_mag, percent_dropped)`
where `in_freq_mag` is a slice of the FFT that represents a particular frequency band. This will be very similar to the function you created in the prelab.
2. `bands2indices()` can be used to formulate the slices to pass to `drop_freq_band()`. Concatenate the results of `drop_freq_band()` and you will have the original `freq_mag` but with the masked frequencies removed.

3. Ensure that a frequency on the edge of two frequency bands only gets considered once. If your code considers it in both bands, the length of your output will not match the input. Off-by-one indexing is a common mistake.
4. Test that dropping 0% of frequencies gives the input as the output. Use `isequal()`. Similarly, test that dropping 100% of frequencies gives a 0 vector.

iv. In the end, you should have a top-level function `function [new_freq_mag] = drop_freqs(freq_half, freq_mag, band_cutoffs, drop_portion)` that composes your helper functions together. The function should work on an input of any size, provided that `freq_half` and `freq_mag` are the same length.

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

5. 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. Remember you must use the method discussed in the background/prelab to handle the complex conjugates required by ifft. Use the `calc_conjugate`() function provided to recompute the complex conjugates after dropping frequencies from the first half of the frequency samples.
   3. 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(calc_conjugate(samples_freq_extracted_dropped),
'symmetric')
```

   4. 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.
   5. Play 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?
   6. Show your TA your plot showing the tones dropped and your before and after frequency samples and answer some questions.
      i. This is your Exit Checkoff for February 23rd lab, but you should continue on to Section 3 as time permits.

Lab Section 3 - Automating psychoacoustic analysis

- In the previous section, you only dropped frequencies for a small portion of the sound wave. The goal of this section is to drop frequencies from all bands for the entirety of the song.
- This will build off of your Lab 5 prelab.
- This section corresponds to steps A through F on the diagram.
- Make sure to read through the entire rest of the lab before beginning this section.

1. Build on your code from Section 2 to remove masked frequencies.
   a. You will need to create a separate Matlab function that contains the logic in the plot_dft() function and analyzes the transformed signal for masking, frequency bin by frequency bin.
   b. Call this function "drop_samples" (this will perform A through F) and give it the following ins/outs:

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

   c. Create a new file called `mainsection3.m` to test your function.
   d. 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.
         Note: When you extract a subset of the song to feed in as `samples_time`, make the length of `samples_time` a multiple of `window_size` so that you don't have to deal with a window that is shorter than `window_size`. Additionally, ensure your code does not double-count any samples on the boundary of two windows. The first window is indices 1 through `window_size` inclusive. The second is indices `window_size + 1` through `2*window_size` inclusive. And so on.
      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` (at the end of step F) 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. This is B.
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. This is D. Start with your code from Section 2 of this lab and refine as necessary. For simplicity, critical band 24 should include all frequencies from the lower end of critical band 24 to the Nyquist frequency.
    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. Repeat these steps for each window in the `samples_time` and concatenate these separate window results together to produce the expected `windowed_freqs` result.

e. 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.
    ii. You should also try dropping different fractions of frequencies and see if the number of frequencies dropped is what you expect.

2. Use the following filtered_frequencies_to_time code to convert the results of Step 1 to time domain and listen to the results. In the next sections we will add Huffman compression and decompression in between, but it is useful to validate that you can convert this data back and to see how it sounds before adding in the Huffman compression and decompression.

```
function [samples_time] =
filtered_frequencies_to_time(windowed_freq_mag, window_size)
windows = floor(length(windowed_freq_mag)/(window_size/2));
freq_win_size = window_size/2;
samples_time = [];
for i=0:(windows-1)
    window = windowed_freq_mag((i*freq_win_size +
1):((i+1)*freq_win_size));
    window_time = ifft(calc_conjugate(window), "symmetric");
    samples_time = [samples_time; window_time];
end
end
```

3. For this decoding function to work, your encoding function needs to produce data compatible with it.

Lab Section 4 – Using Huffman Encoding

- In this section of the lab, we provide code similar to your Huffman function from your previous labs to compare the compression ratios before and after dropping samples. This code also includes quantization of the data.
- This section corresponds to steps G through I on the diagram.

1. We provide 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_freqs(windowed_freqs,word_bits,frac_bits)
% First quantize freqs after masked freqs have been droppped
r = real(windowed_freqs);
i = imag(windowed_freqs);
rq = quantizenumeric(r, 1, word_bits, frac_bits, 'nearest');
iq = quantizenumeric(i, 1, word_bits, frac_bits, 'nearest');

% Now compress
[compressed_real, dict_real, avglen_real] = compress_huff(rq);
[compressed_imag, dict_imag, avglen_imag] = compress_huff(iq);
symbols_real = dict_real(:, 2);
symbols_imag = dict_imag(:, 2);
end

function [samples_compressed, dict, avglen] =
compress_huff(my_samples)
% Helper function for compress_freqs
a = tabulate(my_samples);
prob = a(:,3);
prob = prob/100;
symbols = a(:,1);
[dict, avglen] = huffmandict(symbols, prob);
samples_compressed = huffmanenco(my_samples, dict);

end
```

   Put the above functions in the same file and save it as compress_freqs.m
2. The logic for this function is as follows:
   a. Split windowed_freqs into the real and imaginary components. This is G.
   b. Quantize the real and imaginary components separately. This is H.
      i. Quantization is based on the `word_bits` and `frac_bits` arguments.
   c. Use the logic from the Huffman function from Lab 3 to compress the real and imaginary components separately. This is I.

   d. Returns 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.
   e. Make sure to investigate the huffmandict function and its return values.
3. 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. Make sure to run `drop_samples()` on the excerpt before running `compress_freqs()`. Later parts of this lab will explain how to modify long excerpts so Huffman compression runs faster.
   a. Use `word_bits`=16 and `frac_bits`=12 when quantizing the data with `compress_freqs()`.

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.
- This section corresponds to J and K in the diagram.

1. We are providing you with a function called `decompress_to_time` that will take the output of the 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. 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_freqs() 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.

Lab Section 6 – Putting It All Together

- 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 new file called `mainsection6.m` to contain your code and tests for the remainder of the lab.
2. 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). This is A.
    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_()` function and the `window_size` you chose earlier. This is A—F.
    g. Now call compress_freqs() form Section 4 on the result of drop_samples(). This is G-I.
3. Create a new section to decompress the song:
    a. Decompress using `decompress_to_time()`. This is J.
    b. Plot this decompressed data in the time domain. This is K.
4. 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`.
    a. Read this to better understand the word and frac bits in quantizenumeric(): https://www.mathworks.com/help/dsp/ug/concepts-and-terminology.html
5. 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 to 16 bits. First calculate the total size given 16b per value. For instance, how many bytes for 1 second of audio? How many bytes for the entire input?
    b. As for the compressed data size, first include the bytes required for the Huffman-encoded data. However, also make sure to include the data required to store the dictionaries, which are still needed for decoding. Instead of directly measuring the size of each dictionary, try to think about the data requirements for storing the dictionaries. (How many entries in the dictionary? For each entry, how many bits are needed for the key (input symbol) and value (symbol's encoding)?)
    c. Calculate the compression ratio.
6. With a fixed window_size of 2048, fill out the following table

Percent Frequencies Dropped	# Frequencies Dropped	Compression Ratio	Quality? (subjective)
0%			5
10%			
25%			
50%			
75%			

For quality – rate 1—5, where 1 is unrecognizable and 5 is essentially indistinguishable from 0% dropped.

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

window_size	# Frequencies Dropped	Compression Ratio
1024		
2048		
4096		

8. 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 a sample window? For example, why is it wrong to take the Fourier transform of the 2048 sample window and remove the 512 frequencies with the lowest magnitudes? (512 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.
5. Lab report should include error analysis as noted in the “Formal Lab Report” specification. Error analysis is something you should think about in any experiment and data analysis. It’s worth thinking about it, even if the conclusion is there is none. When you think about it, you usually realize there are sources of errors. Often there are good reasons to dismiss errors, but being thoughtful and deliberate about what errors might be present and the potential magnitude of those errors is good in not fooling yourself and is important in communicating clearly to others the potential (non-)impact of various error sources.

   Pointing out your source of data and assumptions you are making about that is valuable (you didn't add any errors, but you do know about the process that likely produced that original .wav data you started with and errors that may exist there.) You performed computations and manipulated data. Could any of that introduced any error? What can you say about it? You made judgements in tuning your algorithm.

HOW TO TURN IN THE LAB

- There will be no weekly lab writeup for either lab session.
- Include answers to prelab questions.
- 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 14th.
- 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(s).
- 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 the formal lab report.
