ECE 2312 C24 Project 2 Report

Kara Giordano May 1, 2024

Table of Contents:

Abstract:

This project primarily explores the generation of tones using sine waves in MATLAB, using those sine waves to mimic the sound of a different audio file, and combining the tones with speech files for filtering. Additionally, this report the aural effects of each signal and the way that the addition of a sine tone can significantly change the way we perceive the base signal.

This report uses code and files from Project 1, such as the `PlotAudioTime` and `PlotAudioSpectrogram` functions. Details regarding the code used in this project that were initially created for Project 1 can be found in the Project 1 GitHub repository linked below.

Programs used:

- MATLAB R2023b or later
- Signal Processing Toolbox for MATLAB
- DSP System Toolbox for MATLAB
- Audacity or similar audio processor

Project 1 GitHub repository: https://github.com/ksgiordano/Kara-Giordano-ECE-2312-C24-Project-1

Project 2 GitHub repository: https://github.com/ksgiordano/Kara-Giordano-ECE-2312-C24-Project-2

Part I: Sine Tone Generation

In MATLAB, a sine tone can be generated by simply using the built-in `sin` or `sinpi` functions. By mapping these functions to discrete points using a linearly spaced vector, we can load, save, modify, and plot the sin function. Similarly to Project 1, some variables, such as `SampleRate` and the figures `f1 and` f2 were defined for use later, with `SampleRate` being 44.1kHz for this entire project. Below is the code that generated a 5kHz sine wave that is 5 seconds long, and the plotted outputs.

```
t = linspace(0, 5, SampleRate*5)'; % SampleRate*5 elements linearly spaced going from 0-5
F = 5000; % Sine wave frequency in Hertz
sin5000Hz = sinpi(2*F*t); %Generates sine wave. sinpi includes pi in the function...
% ...automatically and is more accurate than adding a rounded pi within the function
PlotAudioTime(sin5000Hz, SampleRate, f1); %Plots signal over its entire length
PlotAudioTime(sin5000Hz, SampleRate, f1); %Plots signal again...
axis([0 1/5000 -1 1]); %...Sets axis to 1 period of the wave
PlotAudioSpectrogram(sin5000Hz, SampleRate, f2, 8000) %Plots spectrogram of generated sine wave
sound(sin5000Hz, SampleRate); %Plays generated sine wave
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\Kara Giordano SineTone.wav"...
, sin5000Hz, SampleRate) %Writes sine tone to disk
```

Figure 1: Code to generate, plot, and play a 5kHz sine wave. Using the sinpi function gives a more accurate representation of a sine wave than inserting a pi within the sin function. The 5kHz wave is plotted both over its whole length and over 1 period.

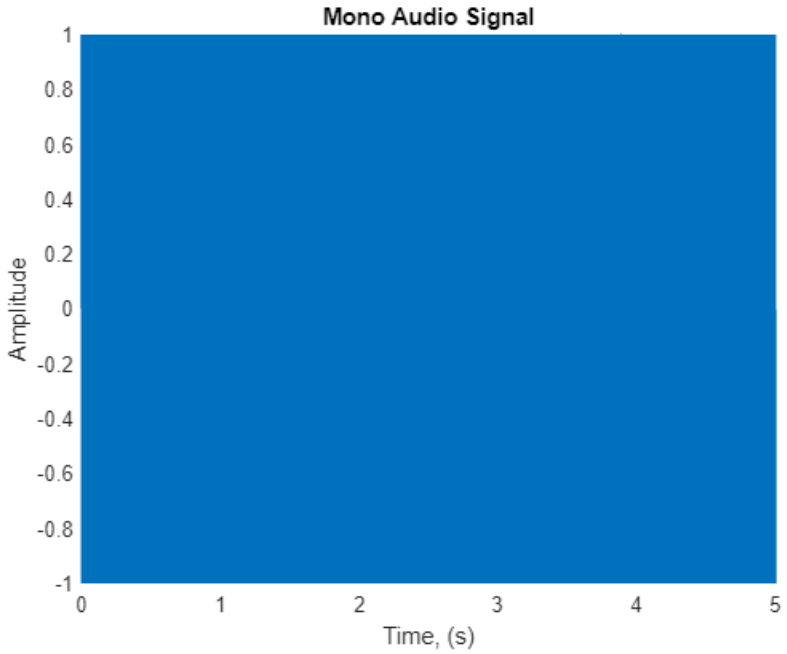


Figure 2: 5kHz sine wave plotted over its entire length. Because the frequency is high and the amplitude of the wave oscillates between -1 and 1, the plot is a solid color and is not useful for analysis of the wave.

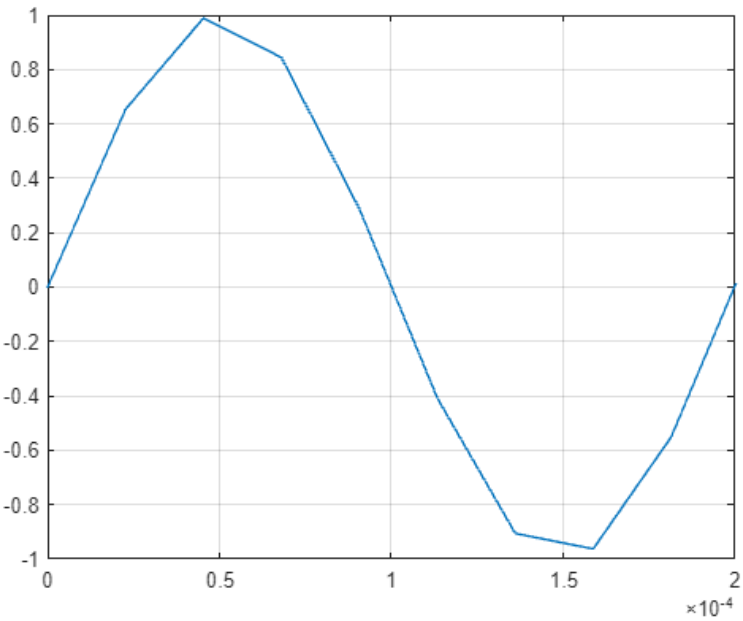


Figure 3: A single period of the 5kHz sine wave plotted. Because the sample rate is limited to 44.1kHz, the function is limited to 8.82 discrete points per cycle. This limited resolution causes what would be a smooth wave if plotted continuously to look somewhat choppy and disjointed.

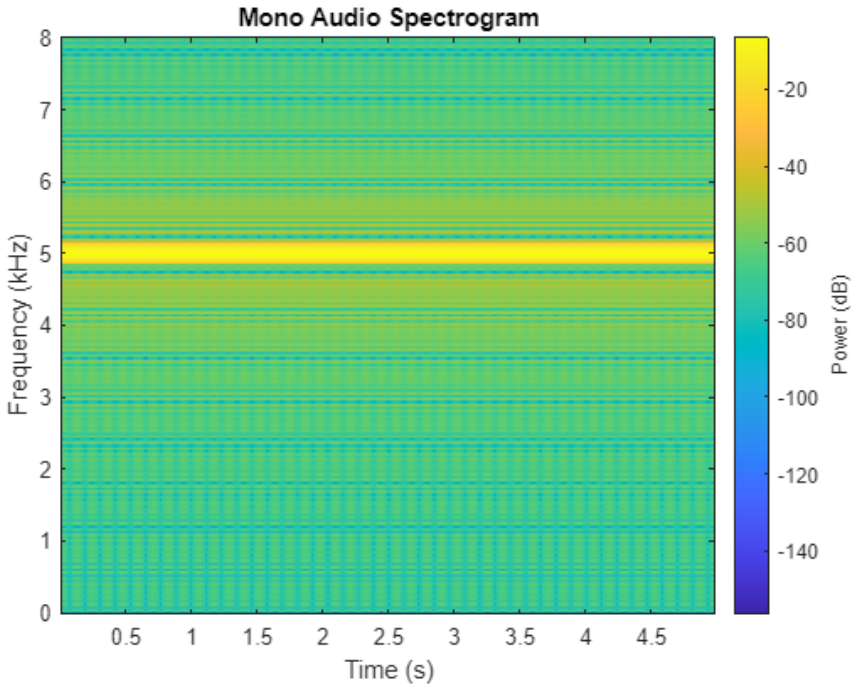


Figure 4: Spectrogram of the 5kHz signal. Because the Fourier transform of a sine wave outputs the frequency of the wave, the spectrogram plots a solid line at 5kHz as expected. The noise at around -40dB to -60dB is likely due to the discrete sampling of the sine wave, with those points connected linearly, rather than a smooth curve.

When the code is run and the wave is played using the `sound` function, it sounds like a high-pitched constant tone for 5 seconds. Even though it is made up of discrete points making the signal appear choppy when plotted over 1 period, it sounds like a continuous, pure sine wave with a high sample rate.

Part II: Chirp Signal Generation

Generating a chirp signal has some similarities to simply generating a sine wave. First, the frequency vector is created, and is a linearly spaced vector that scales from 0-8kHz over the length of `SampleRate*5` to create a 5-second-long chirp. Using the `sinpi` function, 2 times the cumulative sum of `F` divided by the sample rate outputs a chirp. Below is the code used to generate, play, plot, and save the chirp, as well as the output spectrogram.

```
% Generate the chirp signal using the sinpi functionn
F = linspace(0, 8000, SampleRate*5)'; %Frequency linearly goes from 0-8000Hz
sin8000HzChirp = sinpi(2*cumsum(F)/SampleRate); %Generates chirp
PlotAudioSpectrogram(sin8000HzChirp, SampleRate, f1, 10000) %plots spectrogram
sound(sin8000HzChirp, SampleRate); %Plays chirp
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\Kara Giordano Chirp.wav"...
    , sin8000HzChirp, SampleRate) %Writes chirp to disk
```

*Figure 5: Code used to generate, plot, play, and save the 0-8kHz sine chirp. A linearly spaced vector with SampleRate*5 elements is used like in the 5kHz sine tone generation done earlier.*

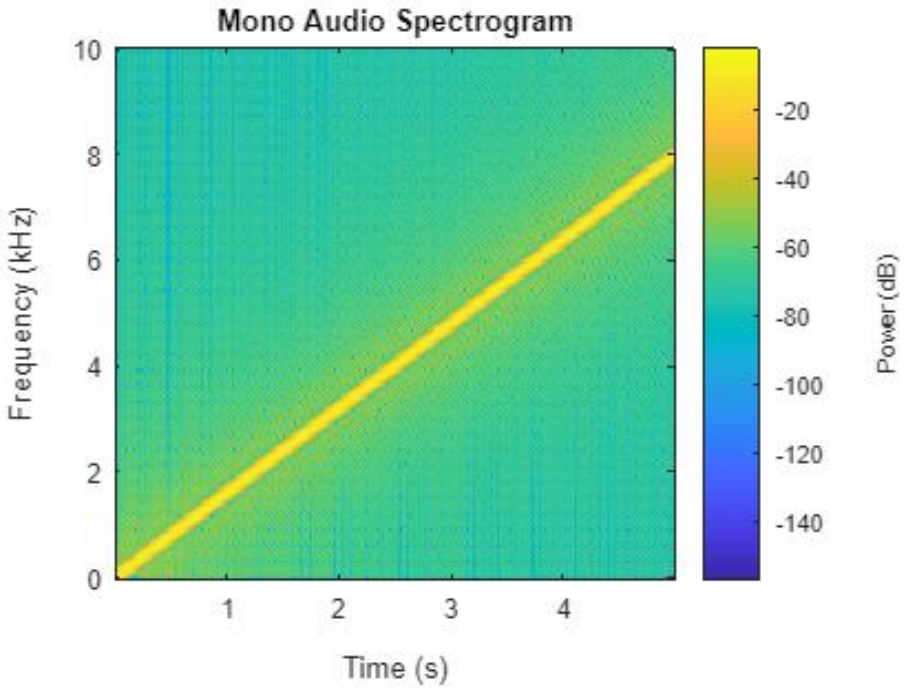


Figure 6: Spectrogram of the 0-8kHz sine chirp. As expected, it is made up a linear increase of frequency between 0 and 8kHz over five seconds. The slight noise around the linear chirp is probably due to being a discrete signal with a limited sampling rate, like in the 5kHz tone.

When listening to the chirp tone, it sounds like the frequency increases logarithmically rather than linearly. The signal sounds like it quickly increases in frequency, and perceived volume and this growth slows over time as the signal is played. It seems like our brains might have a harder time distinguishing between frequencies when the frequency is higher than 3-4kHz, which might cause this perceived logarithmic increase in pitch and volume in a linear signal. Perhaps if a signal increases in frequency exponentially it would be perceived as a linear increase over time.

Part III: Some Fun with Sine Tones

For this part of the report, we took a five tone sequence that was frequently used in the movie Close Encounters of the Third Kind (CETK) by Steven Spielberg. Using Audacity, I recorded the computer's audio while playing the video by using the built-in loopback recording feature. Next, the file was saved as "CETK 5 Tones.wav" for ease of use later. Using the `audioread` function in MATLAB, followed by the `PlotAudioTime` and `PlotAudioSpectrogram`, we can see what this signal looks like when plotted in time and when plotted in time with frequency:

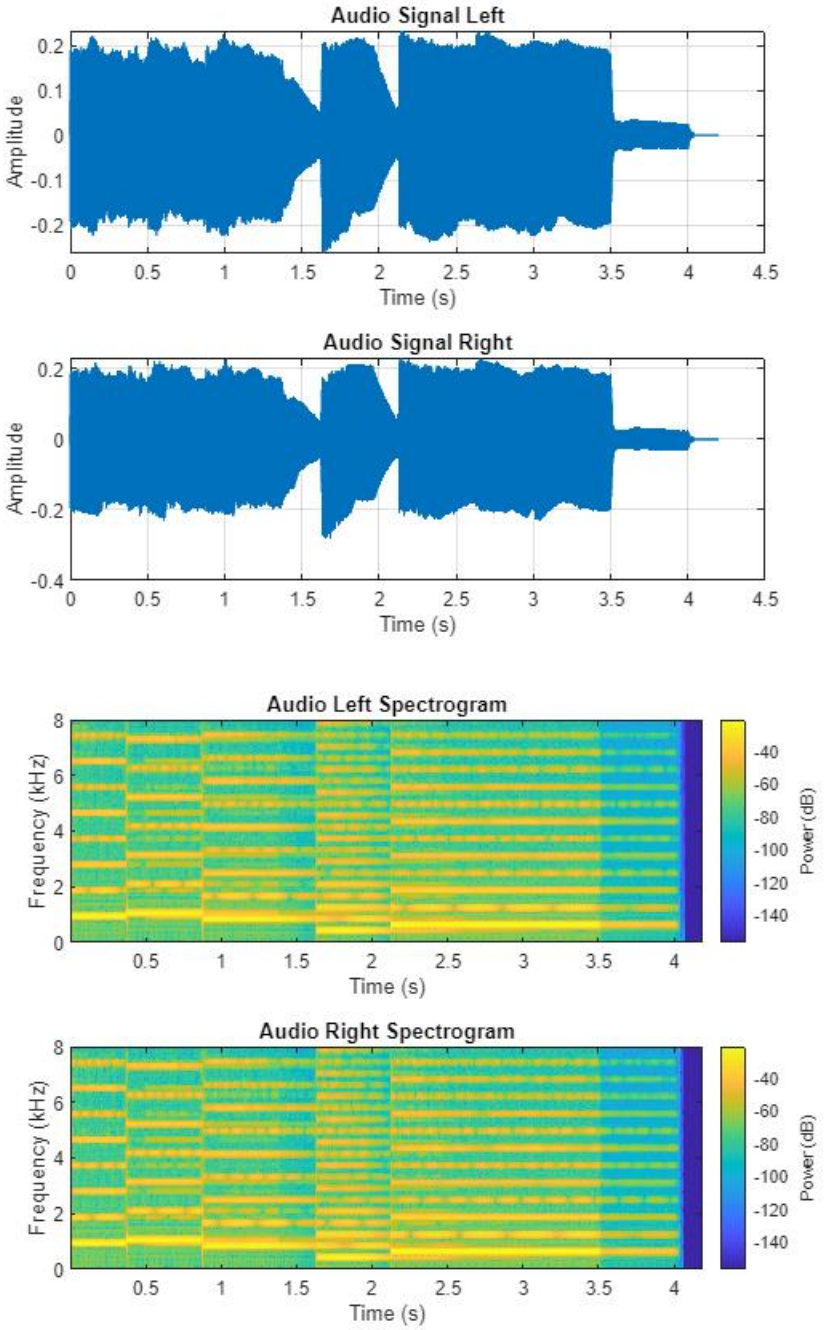


Figure 7: Time and spectrogram plots for the 5-tone file for CETK. Each note is made up of 8-12 individual frequencies when plotted from 0-8kHz. For each of the notes, it seems like the second lowest frequency changes in volume over time periodically.

For each of the notes, they were made up of multiple individual frequency tones, with the lowest frequency being the loudest. Some of the higher frequencies seem to oscillate in volume over time while the frequency stays constant. Visually, we can guess what each frequency is, however it is much more accurate to do this analytically in MATLAB.

To determine the frequencies and volume of the frequencies in the signal, a spectrum analyzer can be utilized. Below is the `GetFreqPwr` function, which takes an input of the audio file, the sample rate, and the start/end samples of the note. The function first creates a spectrum analyzer object with set parameters, then the peak finder is enabled to find 8 peaks in the audio data for a good balance between processing time and accuracy. For finding the peaks of different notes in one signal, the spectrum analyzer is only used on the frequency and volume vectors of the measured peaks are then sent to the output of the function.

```
function [F, Pwr] =  GetFreqPwr(Audio, Fs, SampleStart, SampleEnd)
    scope = spectrumAnalyzer(SampleRate = Fs,...
         AveragingMethod = "vbw", PlotAsTwoSidedSpectrum = false,...
         FrequencySpan = "start-and-stop-frequencies", StartFrequency = 0,...
         StopFrequency = 8000); %Creates spectrumAnalyzer obect with specified parameters
    scope.PeakFinder.Enabled = true; %Enables peak finding
    scope.PeakFinder.LabelPeaks = true; %Enables peak finder labels
    scope.PeakFinder.NumPeaks = 8; %Sets number of peaks to be found

    t = SampleStart:SampleEnd; %Integer vector from beginning to end of sample
    scope(Audio(t,:)); %Run the spectrum analyzer on the audio signal
    data = getMeasurementsData(scope); %Creates table of measurements of the peaks
    F = data.PeakFinder.Frequency; %Gets the frequency values from the peakfinder
    Pwr = data.PeakFinder.Value; %Gets the power values in dB from the peakfinder
    Pwr = 10.^(Pwr/10); %Converts dB power to a linear
end
```

Figure 8: GetFreqPwr function in MATLAB. In addition to getting the frequency of peaks and the volume of those peaks, the function also plots the spectrum data of the signal between 0 and 8kHz. The spectrum analyzer only considers the information between the given start and end samples of the signal, making it useful for analyzing notes.

Next, I made a function called `SineGen` to take the measured frequencies and volumes and generate a tone using those frequencies to mimic a note in the original sound file. By inputting the frequency and power vectors from `GetFreqPwr`, the length of the desired output in samples, and the sample rate, the function outputs a generated vector consisting of 8 sine waves. The function first makes a SampleRate by 8 matrix for each of the separate tones, then the sum of those vectors is taken. Following this, a linear attenuation on the beginning and end of the wave that's 100 samples long is added to prevent popping noises when switching between notes. Finally, the function takes that vector and normalizes the peak amplitude to +/- 1.

```
function OutWave = SineGen (F, Pwr, SampleLength, Fs)
t = linspace(0, SampleLength/Fs, SampleLength)';
Wave = zeros(SampleLength, 8); %Samplelength x 8 matrix of zeros
    for v=1:8
    Wave(:,v) = Pwr(v, 1) * sinpi(2*F(v, 1)*t); %Each column of the matrix, v, is the frequency at position v
    end
Wave = sum(Wave,2); %Sums each vector in Wave matrix
%Adding slight attenuation at beginning and end of wave...
t = linspace(0, 1, 100)';
t = paddata(t, SampleLength, FillValue=1);
Wave = Wave.*t;
t = linspace(1, 0, 100)';
t = paddata(t, SampleLength, FillValue=1, Side="leading");
Wave = Wave.*t;
%...prevents pops and clicking when switching notes
OutWave = 2*((Wave-min(Wave))/ (max(Wave) - min(Wave)))-1; %Normalizes output for max. amplitude to be +/-1
end
```

Figure 9: SinGen function in MATLAB. Running this function multiple times to give multiple note outputs allows us to construct a signal close to that of the original CETK file.

Using both the `SineGen` and `GetFreqPower` functions, the following code was executed to generate each note. For the 3rd, 4th, and 5th note, linear attenuation was added to represent the original signal more accurately. The lengths of each note and the attenuations were estimated using Audacity, and are not 100% accurate to the original file, but are rather close.

```
[AudioData, SampleRate] = audioread("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\CETK 5 Tones.wav");
PlotAudioSpectrogram(AudioData, 44100,f1, 8000)

[Freqs, Powers] = GetFreqPwr(AudioData, SampleRate, 1, 16650);
Note1 = SineGen(Freqs, Powers, 16650, SampleRate);

[Freqs, Powers] = GetFreqPwr(AudioData, SampleRate, 16651, 38700);
Note2 = SineGen(Freqs, Powers, 22050, SampleRate);

[Freqs, Powers] = GetFreqPwr(AudioData, SampleRate, 38701, 71800);
Note3 = SineGen(Freqs, Powers, 33100, SampleRate);
t = linspace(1, 0.1, 71800-60750)'; %Linearly spaced vector from 1 to 0.1 in 71800-60750 samples
t = paddata(t, size(Note3,1), FillValue=1, Side="leading"); %Pads t with 1s to equal length of Note5
Note3 = t.*Note3; %Multiply Note3 by t for attenuation

[Freqs, Powers] = GetFreqPwr(AudioData, SampleRate, 71801, 93800);
Note4 = SineGen(Freqs, Powers, 22000, SampleRate);
t = linspace(1, 0.1, 93800-86500)'; %Linearly spaced vector from 1 to 0.1 in 93800-86500 samples
t = paddata(t, size(Note4,1), FillValue=1, Side="leading"); %Pads t with 1s to equal length of Note4
Note4 = t.*Note4; %Multiply Note4 by t for attenuation

[Freqs, Powers] = GetFreqPwr(AudioData, SampleRate, 93801, 178000);
Note5 = SineGen(Freqs, Powers, 84200, SampleRate);
t = linspace(1, 0.1, 1000)'; %Linearly spaced vector from 1 to 0.1 in 1000 samples
t = paddata(t, 61000, FillValue=1, Side="leading"); %
t = paddata(t, size(Note5,1), FillValue=0.1, Side="trailing");
Note5=t.*Note5;

WavOut = cat(1,[Note1;Note2;Note3;Note4;Note5]); %Puts each note into 1 vector one after another
PlotAudioTime(WavOut, SampleRate, f2);
PlotAudioSpectrogram(WavOut, SampleRate, f2, 8000);
sound(WavOut, SampleRate); %Plays cetk tones
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\Kara Giordano cetk.wav", WavOut, SampleRate)
pause(4.5)
```

Figure 10: Code executed to generate each note individually then concatenates each note, plotting the generated CETK tone sequence in time and on a spectrum before playing the sequence and writing it to the disk. For ease of comparison, the original signal is also plotted.

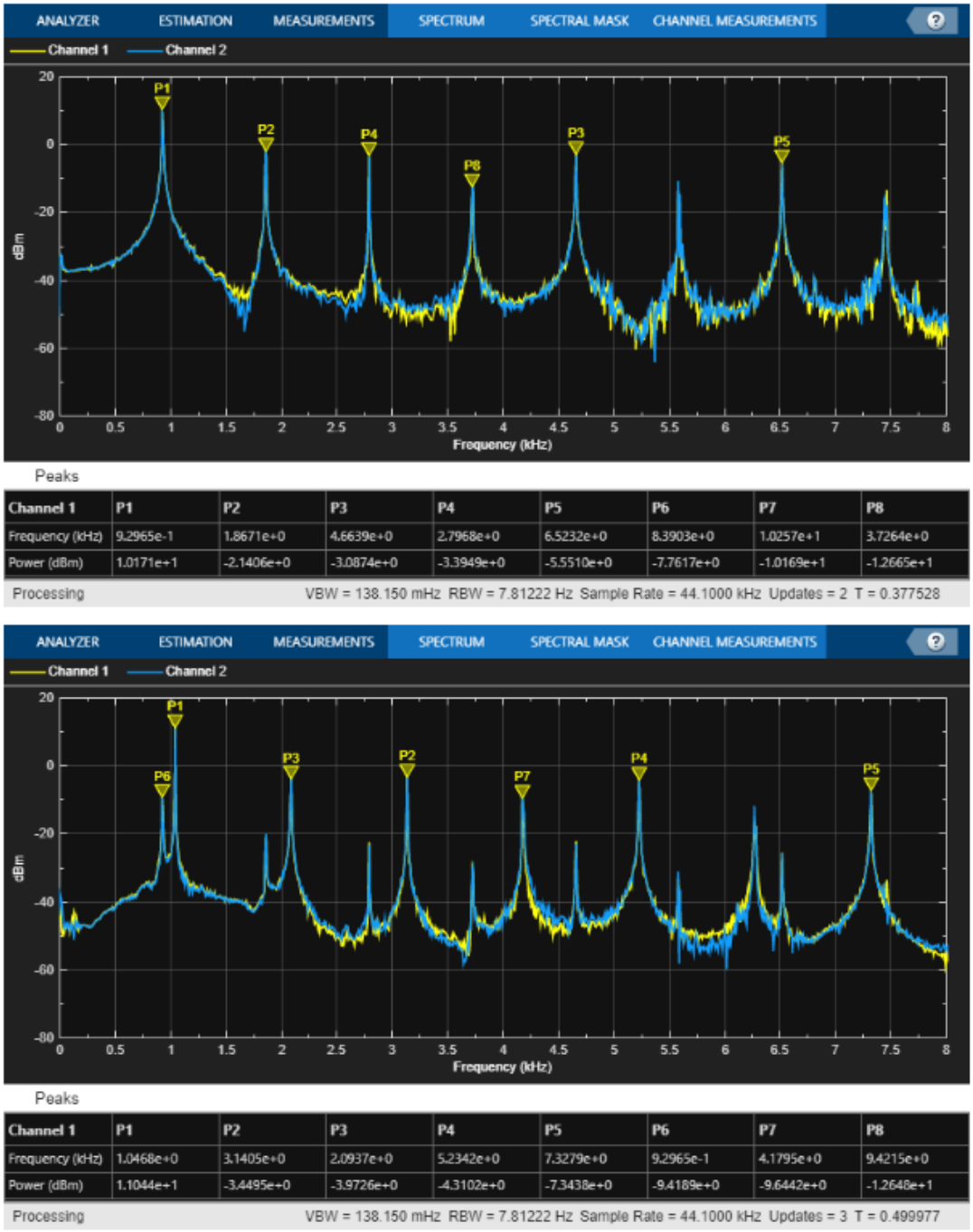
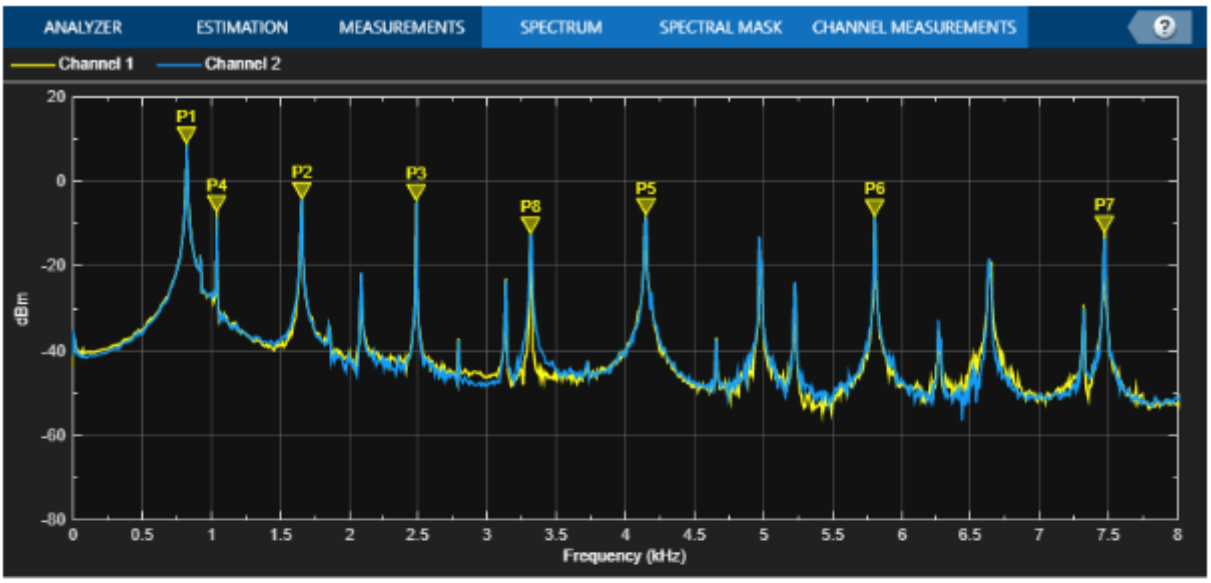


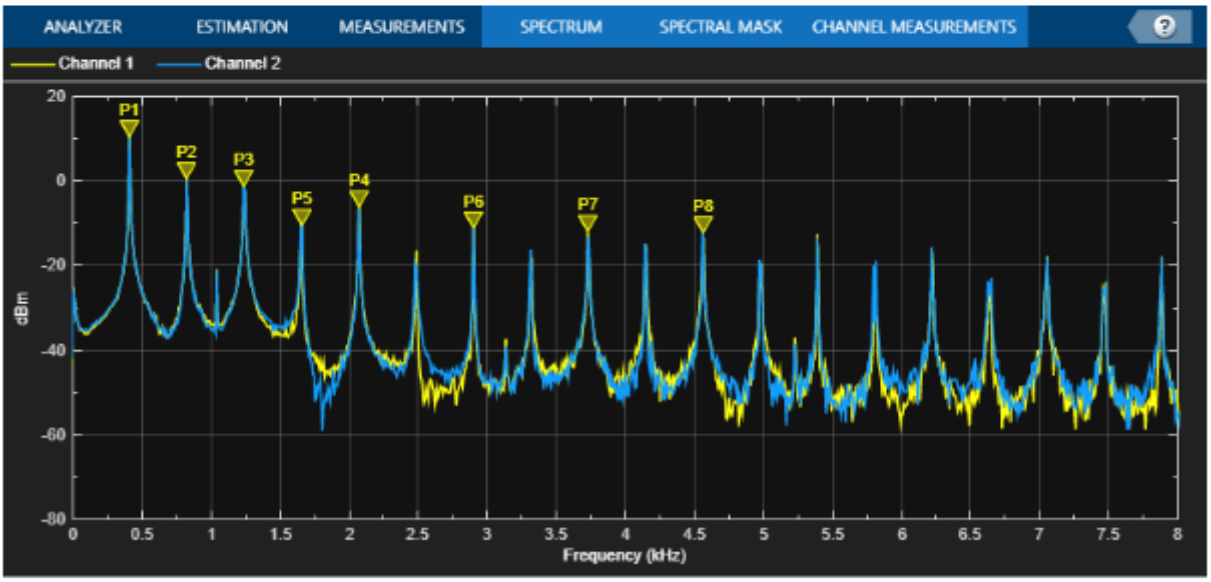
Figure 11: Notes 1 and 2 plotted on the Spectrum Analyzer. Peaks with the frequencies and power computed are labeled in the plots, with their frequency and power displayed below.



Peaks

Channel 1	P1	P2	P3	P4	P5	P6	P7	P8
Frequency (kHz)	8.2810e-1	1.6640e+0	2.4921e+0	1.0468e+0	4.1561e+0	5.8123e+0	7.4763e+0	3.3202e+0
Power (dBm)	8.8844e+0	-4.1920e+0	-4.5954e+0	-7.2393e+0	-7.9590e+0	-8.2151e+0	-1.2085e+1	-1.2381e+1

Processing VBW = 138.150 mHz RBW = 7.81222 Hz Sample Rate = 44.1000 kHz Updates = 5 T = 0.750544

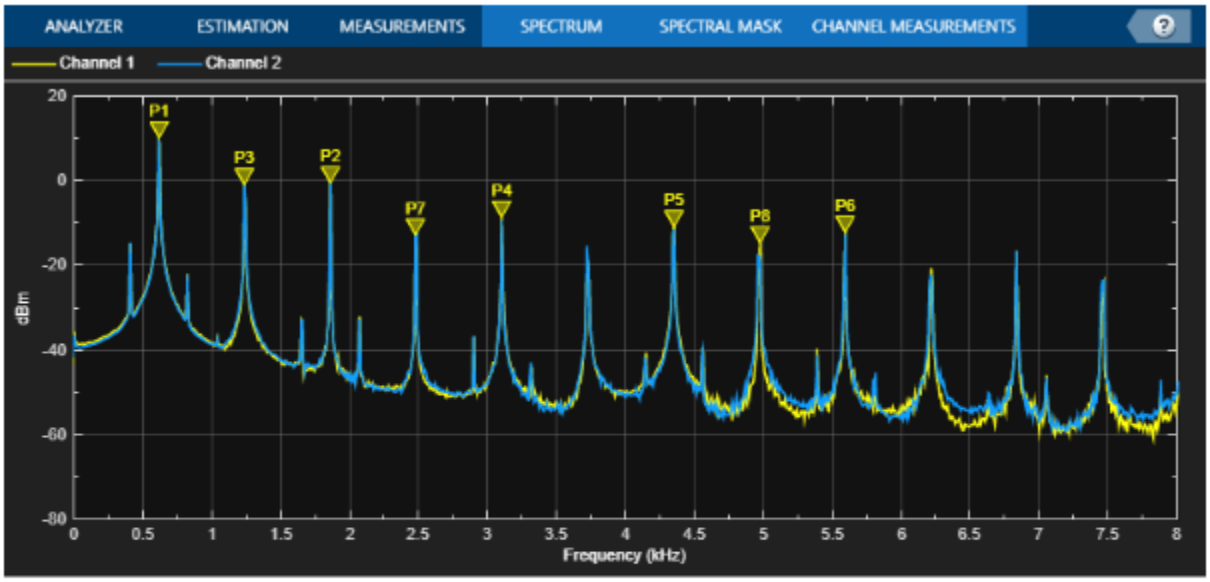


Peaks

Channel 1	P1	P2	P3	P4	P5	P6	P7	P8
Frequency (kHz)	4.1405e-1	8.2810e-1	1.2421e+0	2.0781e+0	1.6640e+0	2.9061e+0	3.7342e+0	4.5702e+0
Power (dBm)	1.0366e+1	4.2277e-1	-1.4124e+0	-6.2519e+0	-1.0655e+1	-1.1235e+1	-1.2087e+1	-1.2373e+1

Processing VBW = 138.150 mHz RBW = 7.81222 Hz Sample Rate = 44.1000 kHz Updates = 3 T = 0.498844

Figure 12: Notes 3 and 4 plotted on the Spectrum Analyzer. Peaks with the frequencies and power computed are labeled in the plots, with their frequency and power displayed below. Note 4 in particular has a lot of peaks that were not computed.



Peaks

Channel 1	P1	P2	P3	P4	P5	P6	P7	P8
Frequency (kHz)	6.2498e-1	1.8671e+0	1.2421e+0	3.1093e+0	4.3592e+0	5.6014e+0	2.4843e+0	4.9842e+0
Power (dBm)	1.0021e+1	-6.1437e-1	-1.0552e+0	-8.7599e+0	-1.0995e+1	-1.2333e+1	-1.2919e+1	-1.4785e+1

Processing VBW = 138.150 mHz RBW = 7.81222 Hz Sample Rate = 44.1000 kHz Updates = 14 T = 1.90927

Figure 13: Note 5 plotted on the Spectrum Analyzer. Similarly to note 4, there are many peaks that were not computed by MATLAB. This can be solved by increasing the number of points to be computed, but significantly increases processing time.

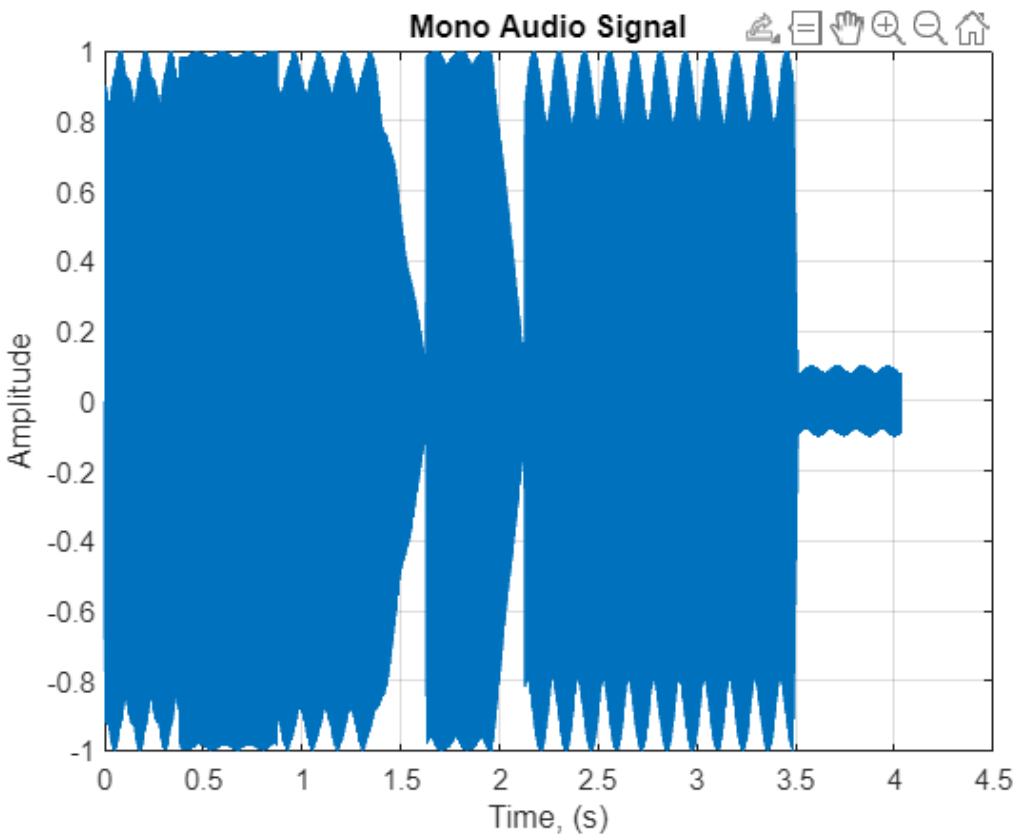


Figure 14: Generated CETK tones plotted in time, the start and end of each note can easily be seen. The peaks from the highest amplitude sine wave in each note are also very clearly visible from the normalization to +/- 1 that was performed.

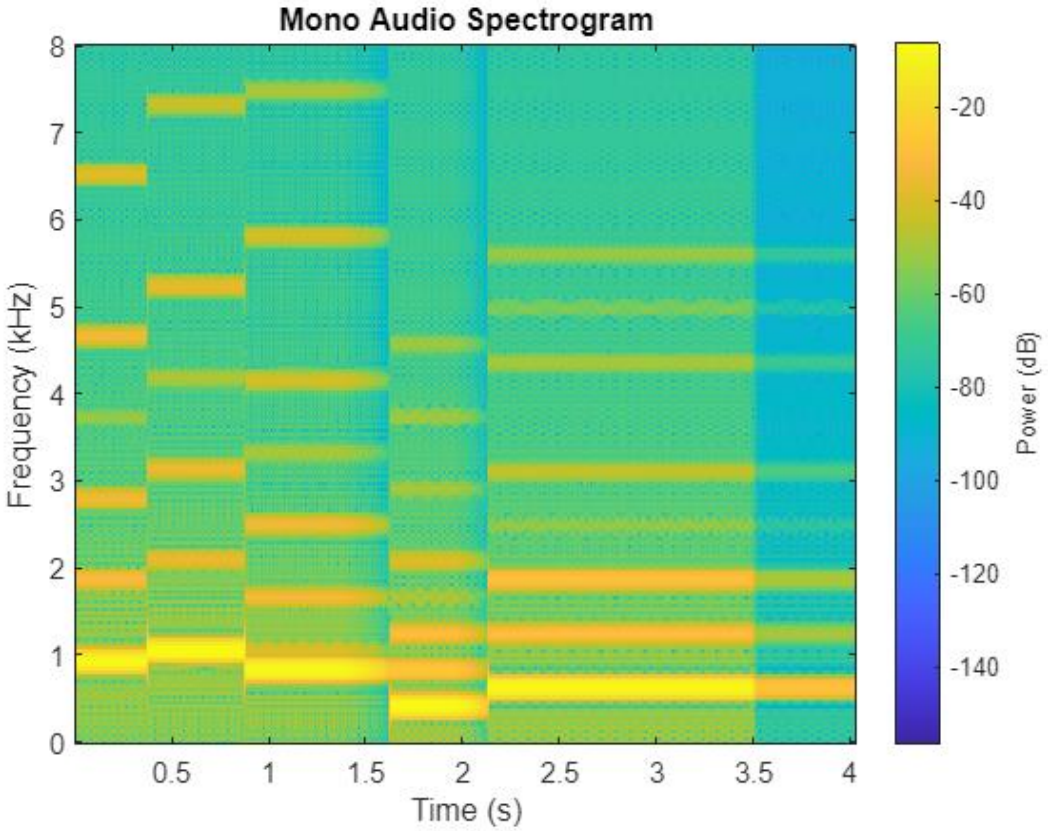


Figure 15: Spectrogram of the generated CETK tones. Similarly to the original file, the lowest pitched note has the highest amplitude, with higher notes generally having lower amplitudes, especially during the 5th note.

Listening to the generated file, it very closely matches the original, with each pitch and note length being almost indistinguishable from each other on their own. However, being only made of 8 sine tones and 1 channel, the generated tones lack the depth in the sound that the original has. Artificially adding noise to the signal, a small amount of stereo delay, different amounts of noise to each stereo channel, and perhaps using a triangle or square wave for a few of the frequencies could help make it sound less 'artificial' and pure, being more like the original. Another option would be to simply increase the number of peaks that get found and generated, however this significantly increases the processing time, and for the lower power noise it just isn't worth it to add the extra processing time, while generating a noise signal would suffice.

Another aspect of the generated signal that is lacking is the attack and decay on each note. Each note has a very slight attack combined with a slight decay before the note gets released. Adding a sawtooth wave and ADSR (attack, decay, sustain, release) parameters to each of the generated notes would help get that stab of sound the original file has.

Part IV: Combining Sound Files

Combining sound files is rather simple to do, loading each signal and adding them will suffice. For using the addition operator to do this, both files must be the same sample rate and length, and for both files, the sample rate is 44.1kHz with a length of 5 seconds. To demonstrate this, here is the code used to combine a speech file recorded during Project 1 with the 5kHz sine wave that was generated in Part I.

```
[AudioData, SampleRate] = audioread("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\TQBF Jumps Over Lazy Dog.wav");
AudioData = AudioData + sin5000Hz;
PlotAudioSpectrogram(AudioData, SampleRate, f1, 8000)
sound(AudioData, SampleRate);
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\speechchirp.wav", AudioData, SampleRate)
pause(5)
```

Figure 16: MATLAB code to load in speech file, combine it with the 5kHz sine generated earlier, plot its spectrogram, play, and write it to the disk.

Below is the spectrogram of the unmodified recorded speech file, and the spectrogram of the signal with the 5kHz sine added:

```
FileName = 'ECE 2312 C24\TQBF Jumps Over Lazy Dog.wav'
```

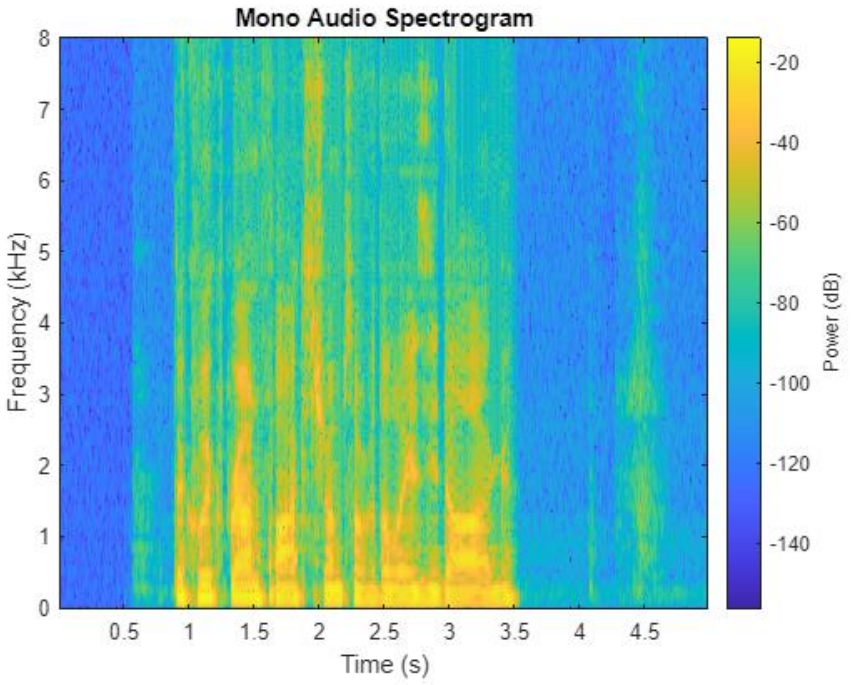


Figure 17: Spectrogram of the original speech file.

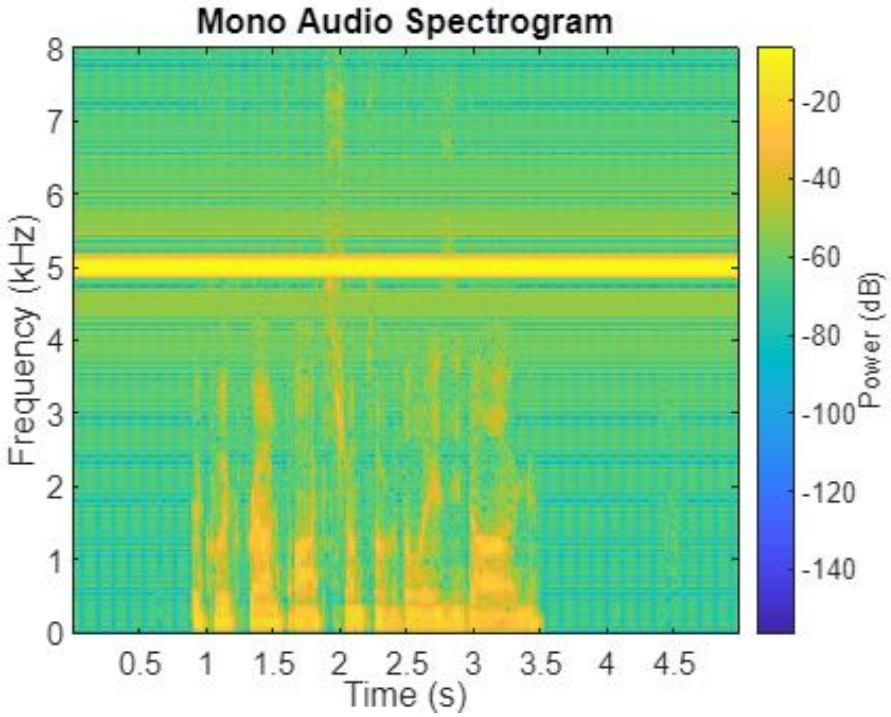


Figure 18: plotted spectrogram of the combined sound files.

When listening to the file, the 5kHz sine wave is by far the loudest part of the file. This makes sense because the amplitude of the original speech file and the sine wave has a maximum and minimum of +/-1, and the generated sine wave will always modulate between that maximum and minimum, while the recorded speech will not always be moving between those two extremes. This combined with the observed effect of higher pitches sounding louder as with the linear chirp tone that was generated in Part II compounds.

The actual speech also has a lower quality to it, sounding close to the crunchy voice lines in old video games. I think this could be because that for each bit, there is a limited bit depth per sample, which was set to 16 bits for the recording during Project 1. Because of the limited amount of data available, the addition of the sine wave takes up more data, essentially leaving less room for each bit of the speech file to take up, leading to the crunchy, low-bit sound.

Part V: Audio and Speech Filtering

There are multiple ways to create a filter in MATLAB, for this project I used the `designfilt` function, which can be used to create a multitude of filters. A lowpass IIR filter was chosen with an order of 8. The passband frequency was specified so that the cutoff frequency would be 4kHz. Ripple was also enabled, with the stopband attenuation at a maximum of -60dB, and a sample rate of 44.1kHz like the rest of this project. All of this information is neatly contained in `LPfilter`.

```
%Make lowpass iir filter
LPfilter = designfilt('lowpassiir','FilterOrder',8, ...
    'PassbandFrequency',4000,'PassbandRipple',1, ...
    'StopbandAttenuation',60,'SampleRate',44100);
fvtool(LPfilter) %Plot frequency response of filter
AudioData = filter(LPfilter, AudioData); %Put AudioData through filter

PlotAudioSpectrogram(AudioData, SampleRate, f3, 8000) %Plot filtered data
sound(AudioData, SampleRate);
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\filteredspeechsine.wav", AudioData, SampleRate)
pause(5)
```

Figure 19: MATLAB code for filter generation and application to the edited speech file. After the filter is made, the signal is filtered, played, and written to the disk. For a lowpass IIR filter, the designfilt object creates the lowpass using a Butterworth filter design method /

Running the `fvtool` command on the filter, a display of the filter's frequency response for easy visualization:

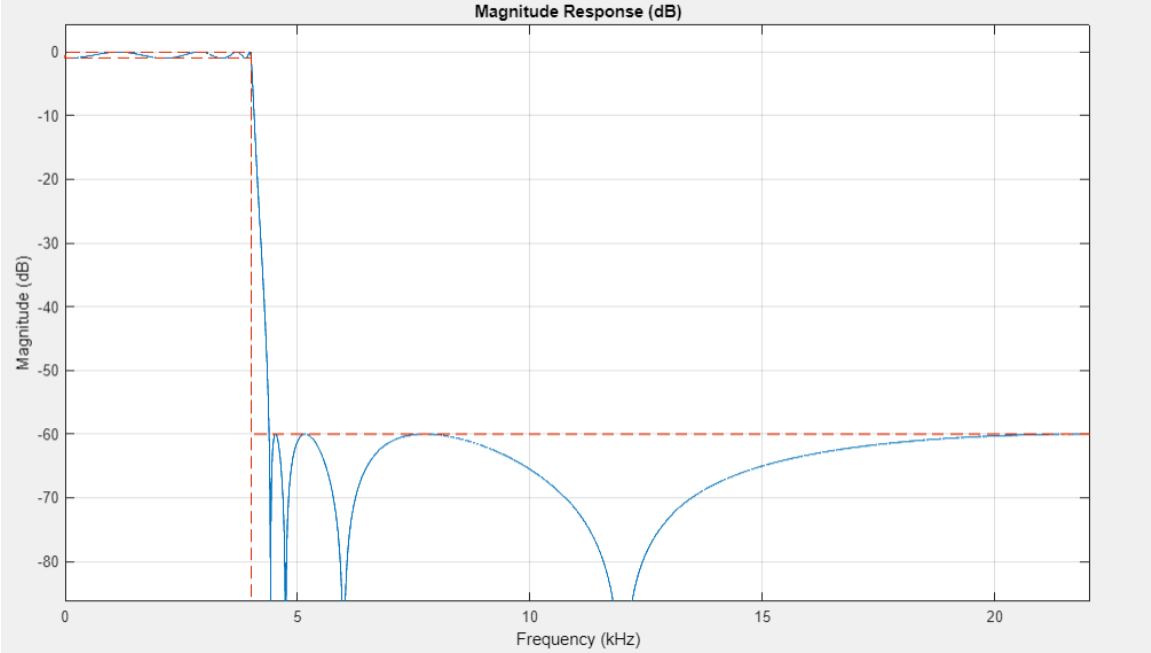


Figure 20: Frequency response of the lowpass IIR filter made using the designfilt command. This is a 8th order filter with a stop band attenuation at -60dB, with rippling in both the stop and pass band. MATLAB generates this response using a Butterworth filter, hence the flat pass and stop band other than the ripples.

Next, the filter command is run with `LPfilter` and the audio data that we want to filter as an input. Below is the spectrogram of the filtered signal:

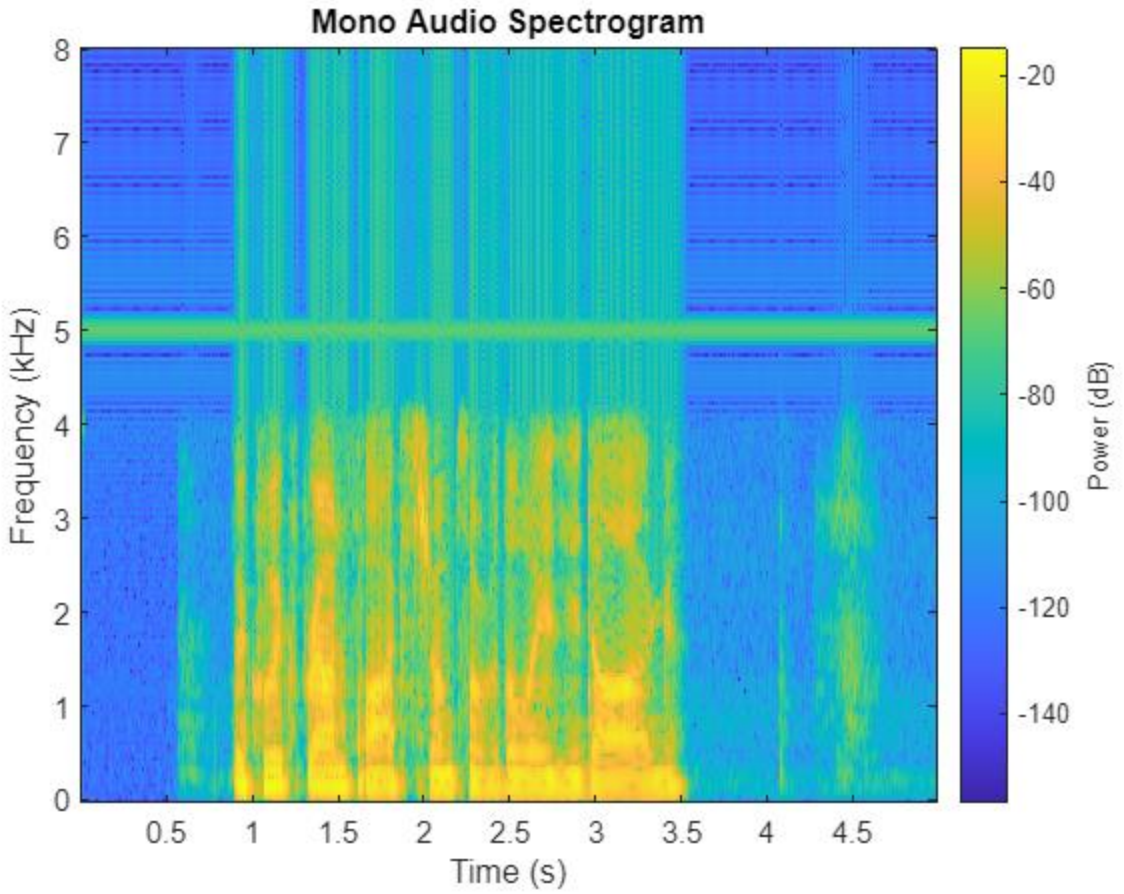


Figure 21: Speech file after the lowpass IIR filter was applied. There is a very clear difference between the amplitude of the signal above and below 4kHz, with the highest power above 4kHz being -60dB, equal to the stop band attenuation.

Compared to the signal with the 5kHz wave added, the overall noise in the spectrogram that was present at around -60dB is greatly reduced, and the original speech signal is much clearer. The transition from the passband to the stopband is very obvious, with the cutoff frequency being at 4kHz and quite clearly defined. The maximum amplitude of each frequency above that 4kHz cutoff is around -60dB, which is equivalent to an amplitude of 0.00001, with 0dB being an amplitude of 1. When listening to the file, this attenuation means that frequencies in this range are essentially imperceptible.

The actual sound of the file sounds very similar to the original file, however the recording has some of the low-bit depth sound as before the signal was filtered, but not to the same extent. This is probably because higher frequency sounds from the original speech file were cut out in addition to the sine wave, losing out on a lot of those sharper, more piercing tones from speech. The filter is also not perfect, with some slight ripple in the pass band and not having a slight band after the cutoff where the attenuation is not quite -60dB. Using a higher

order filter and disabling ripple can help with this, however using a higher order filter in particular increases the amount of computation needed, and the effect of the ripple is likely minimal at most.

Part VI: Stereo Fun

Now, let's take the speech file with the added sine wave and convert it to stereo, with the left channel being just the speech file, and the right channel being the speech file with the added sine tone. To do this, I implemented the following code in MATLAB. It simply loads each of the written files from earlier into their respective channels in `AudioData`, then plots a spectrogram, plays the sound, and writes it to the disk.

```
AudioData = zeros(size(sin5000Hz,1), 2);
[AudioData(:,1), SampleRate] = audioread("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\TQBF Jumps Over
[AudioData(:,2), SampleRate] = audioread("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\speechsine.wav"
PlotAudioSpectrogram(AudioData, SampleRate, f2, 8000)
sound(AudioData, SampleRate)
audiowrite("C:\Users\ngior\Documents\Class Assignments & Files\ECE 2312 C24\Project 2\stereospeechsine.wav", AudioData, SampleRate)
pause(5)
```

Figure 22: Code for generating the stereo file in MATLAB.

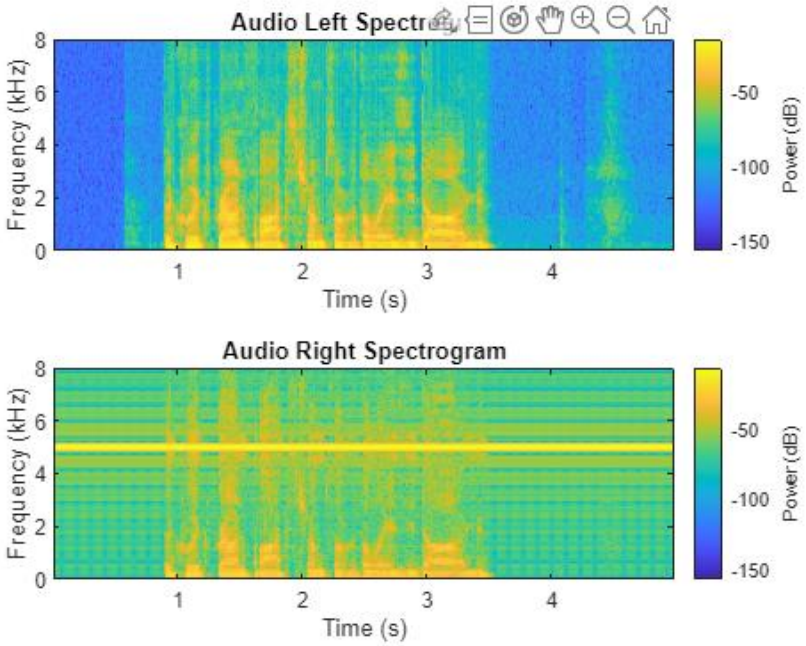


Figure 23: Spectrogram of the stereo signal. The left channel contains the original file, and the right one contains the signal with the added sine wave.

Listening to the file, it sounds similar to the speech file with the sine wave before the filtering was done. The speech has the same crushed effect on both channels; however the right channel of the speech is very quiet, and the majority of the perceived speech is coming from the left side. The file is also just uncomfortable to listen to with headphones because of the sheer difference in the signals from each channel, presumably because our brains aren't sure how to processes such drastically different sounds. The sine tone is also much louder than before, to the point that even on very low volumes it hurts to listen to.

Otherwise, the file is unremarkable, though I think it would be interesting to see the effect of using a very low sine tone or a sine tone that's so high, it wasn't perceptible by humans. For the latter, we would want to increase the sample rate to represent sounds above the Nyquist frequency, because the current sample rate of 44.1kHz can represent frequencies up to 22.05kHz, which is just barely above what humans can hear.