ECE 2312 Project 1 Report

Kara Giordano Feb 12, 2024

Table of Contents:

Abstract:
This project explores recording, writing, and reading audio in MATLAB while plotting those recorded files both in time and with a spectrogram for analysis. Different methods of creating the illusion of sound coming from a particular direction are also explored with stereo delay and attenuation.

Program Required:

- MATLAB R2023b or later
- DSP Toolbox for MATLAB

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

Part I: Recording, Writing, and Reading Audio

The first step in this project was to be able to record audio in MATLAB using a microphone. To do this, we must first know what the internal device ID of the microphone we would like to use so that MATLAB can record sound, and for this project, I first created a function called `AV_Info` to do exactly that:

```
function AV_Info
av = audiodevinfo;
av.input
av.output
end
```

Figure 1: AV_Info Function in MATLAB. Displays both the input and output devices available and their IDs.

When the function `AV_Info` is called, it creates a structure called `av`, with input and output fields. When we call `av.input` and `av.output`, MATLAB displays information about the audio input and output devices respectively.

ans = *1×2 struct*

Fields	Name	DriverVersion	ID
1	'Primary Sound Capture Driver (Windows DirectSound)'	'Windows DirectSound'	0
2	'Microphone (Realtek(R) Audio) (Windows DirectSound)'	'Windows DirectSound'	1

ans = *1×2 struct*

Fields	Name	DriverVersion	ID
1	'Primary Sound Driver (Windows DirectSound)'	'Windows DirectSound'	2
2	'Speakers (Realtek(R) Audio) (Windows DirectSound)'	'Windows DirectSound'	3

Figure 2: Example output from AV_Info. For recording audio, we should use either of the input devices, which are listed in the top structure. If we wanted to play a sound using MATLAB, the output devices listed in the bottom structure can be used.

In this example, we can see that the microphone input for my laptop uses a device ID of 1, so for this program, I will be using that device ID to record audio.

Next, I set up some variables to make writing the rest of the program easier:

```
SampleRate = 44100; %#ok<NASGU> %Sample Rate of Audio Recording in Hertz

NumberBits = 16; %Number of bits per sample

RecordingLength = 5; %Length of recording in seconds

FileName = 'ECE 2312 C24\Do-Re-Mi-Fa-So-La.wav'; %Name of file to be written/loaded

Channels = 1; %Channels to be recorded, 1: mono, 2: stereo

DeviceID = 1; %ID of Device to be used

DelaySide = 1; %Determines which ear will have the stereo delay, 1 for left, 2 for right

DelayLength = 0; %Length of Stereo Delay in ms

%initializes figures to be used for plotting
%without this, calling PlotAudioTime or PlotAudioSpectrum after the other
%would overwrite the previous plot
f1=figure;
f2=figure;
```

Figure 3: Variables set up for later use. Figures f1 and f2 should be used when plotting a time domain plot and spectrogram plot one after another.

In the case of recording audio, we will not need to use all of these variables just yet, however they will become useful later.

To record audio, I made a function called `RecordAudio`, which outputs a matrix called `Audio` that we can set `AudioData` equal to for the program to use later. By calling `RecordAudio(SampleRate, NumberBits, Channel, DeviceID, RecordingLength)`, we can record sound.

```
function Audio = RecordAudio(Fs, NBits, Ch, ID, Len)
recorder=audiorecorder(Fs, NBits, Ch, ID);
disp('Start speaking..')
%Record audio to audiorecorder object,...
...hold control until recording completes
recordblocking(recorder, Len);
disp('End of Recording.');
%Store recorded audio signal in numeric array
Audio = getaudiodata(recorder);
end
```

Figure 4: RecordAudio function in MATLAB. The output of the function is the recorded audio signal and can be set equal to AudioData to use in other functions in MATLAB.

As can be see in figure 4, the function first makes `recorder` an `audiorecorder` object with the desired sample rate, bits per sample, number of channels, and using the desired device. Next, we display "Start Speaking" while `recordblocking` records audio to `recorder` and stops the next line from executing until `Len` seconds have passed. Finally, we display 'End of Recording' and set the output of `RecordAudio` to the data in the `audiorecorder` object.

If we wanted to write this recorded audio to a file to use later, we can use MATLABs built in `audiowrite` command. If we wanted to read an audio file and load it into `AudioData` for us to use, we can use the `audioread` command to get both the data and sample rate from the

file. Below is an example of using the function `RecordAudio`, `audiowrite`, and `audioread` commands:

```
%Records audio into AudioData
AudioData = RecordAudio(SampleRate, NumberBits, Channels, DeviceID, RecordingLength);

%Writes AudioData to path FileName
audiowrite(FileName, AudioData, SampleRate)

%Loads audio from path into AudioData and SampleRate
[AudioData, SampleRate] = audioread(FileName);
```

Figure 5: Example of RecordAudio, audiowrite, and audioread. This code would record a clip of audio, then write it to a file, and load that audio back into AudioData. The audioread command is redundant in this instance as the recorded audio was already loaded into AudioData and is just shown as an example.

Part II: Plotting Audio in Time

So far, we can record, write, and read audio, and because these are either a vector or matrix, displaying this information isn't very helpful for a human interpreting the signal. To make the signals easier to interpret, we can plot the waveform as a function of time.

To plot a waveform in time, I created a function called `PlotAudioTime`, which takes an input of the audio data, sample rate, and the figure to be used for plotting. First it determines the length of the data, `N`, then creates a vector, `t`, with `N` points evenly spaced between `0` and `N` divided by the sample rate. By plotting `t` on the x-axis and the audio data on the y-axis, we ensure that the x-axis scale is in terms of seconds to be more readable for a human.

Because a mono file will be a vector of size `Nx1`, and a stereo file will be a size of `Nx2`, we can use the `size` command in an if-elseif statement to check the number of columns the audio has and generate the correct number plots.

Plotting a mono file is simple, the `t` vector is plotted on the x-axis and the audio on the y-axis to ensure the x-axis is scaled in seconds. Next, we simply add the labels for the plot and grid. For plotting a stereo file, the tiled layout dimensions first get specified, and the nexttile command increments the tile for each plot and its labels. Within the plot command, `(:, 1)` is added after Audio so that MATLAB only plots 1 vector of the `Audio` matrix for the left and right plots respectively.

```
function PlotAudioTime (Audio, Fs, fig)
N=length(Audio);
t = linspace(0, N/Fs, N);
figure(fig);
    if size(Audio, 2) == 1
        plot(t, Audio)
        xlabel('Time, (s)')
        ylabel('Amplitude')
        title('Mono Audio Signal')
        grid on
    elseif size(Audio, 2) == 2
        tiledlayout(2, 1);
        ax1=nexttile;
        plot(t, Audio(:, 1))
        title(ax1, 'Audio Signal Left')
        ylabel(ax1, 'Amplitude')
        xlabel(ax1, 'Time (s)')
        grid on
        ax2=nexttile;
        plot(t, Audio(:, 2))
        title(ax2, 'Audio Signal Right')
        ylabel(ax2, 'Amplitude')
        xlabel(ax2, 'Time (s)')
        grid on
    end
end
```

Figure 6: PlotAudioTime function in MATLAB. For when the file is stereo, ax1 and ax2 handle both incrementing the tile for the next plot, and making sure that the axis and graph labels are on the correct plot.

To demonstrate recording and plotting audio in time, below is the code and output graphs of me saying “The quick brown fox jumps over the lazy dog”, “We promptly judged antique ivory buckets for the next prize”, and “Crazy Fredrick bought many very exquisite opal jewels”.

Note that in the code, the `clf('reset')` command has been added to reset the figure used for plotting. The `audiowrite` command saves the recorded files for later. Below is the code used to record and generate the following plots:

```
FileName = 'ECE 2312 C24\TQBF Jumps Over Lazy Dog.wav'
AudioData = RecordAudio(SampleRate, NumberBits, Channels, DeviceID, RecordingLength);
audiowrite(FileName, AudioData, SampleRate);
PlotAudioTime(AudioData, SampleRate, f1)
clf('reset')

FileName = 'ECE 2312 C24\Promptly Judged Antique Ivory Buckets.wav'
AudioData = RecordAudio(SampleRate, NumberBits, Channels, DeviceID, RecordingLength);
audiowrite(FileName, AudioData, SampleRate);
PlotAudioTime(AudioData, SampleRate, f1)
clf('reset')

FileName = 'ECE 2312 C24\Crazy Fredrick Bought Opal Jewels.wav'
AudioData = RecordAudio(SampleRate, NumberBits, Channels, DeviceID, RecordingLength);
audiowrite(FileName, AudioData, SampleRate);
PlotAudioTime(AudioData, SampleRate, f1)
```

Figure 7: Code used record and plot audio. Each file gets written to the ECE 2312 C24 folder with their respective file names. For recording these files, a sample rate of 44.1kHz, with 16 bits per sample, 1 channel, a device ID of 1, and a length of 5 seconds.

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

Start speaking..
End of Recording.
```

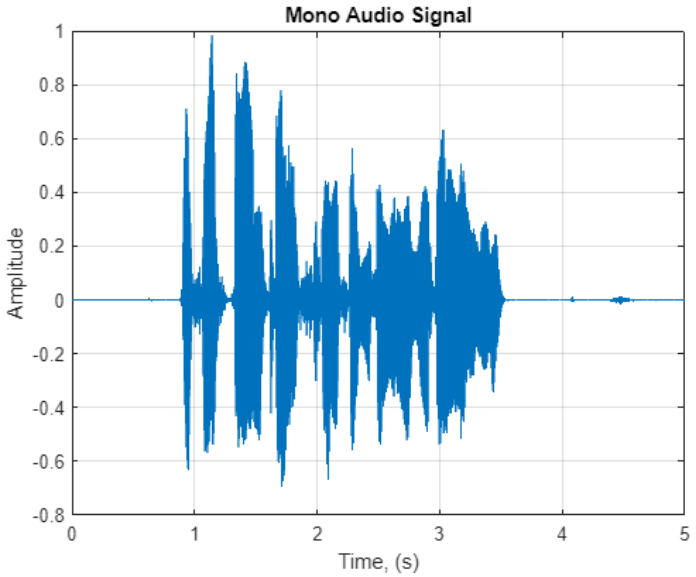

Figure 8: Recording of "The quick brown fox jumps over the lazy dog" plotted in time.

```
FileName = 'ECE 2312 C24\Promptly Judged Antique Ivory Buckets.wav'

Start speaking..
End of Recording.
```

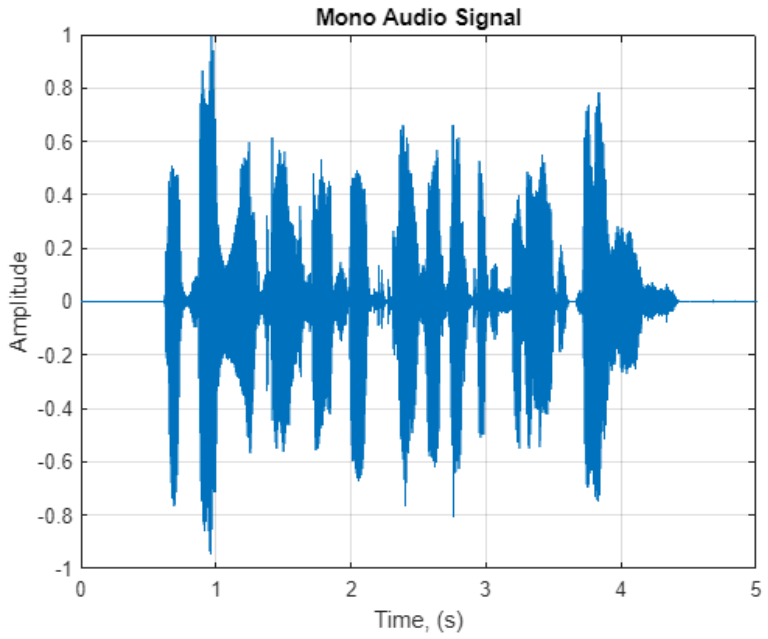

Figure 9: Recording of "We promptly judged antique ivory buckets for the next prize" plotted in time.

```
FileName = 'ECE 2312 C24\Crazy Fredrick Bought Opal Jewels.wav'

Start speaking..
End of Recording.
```

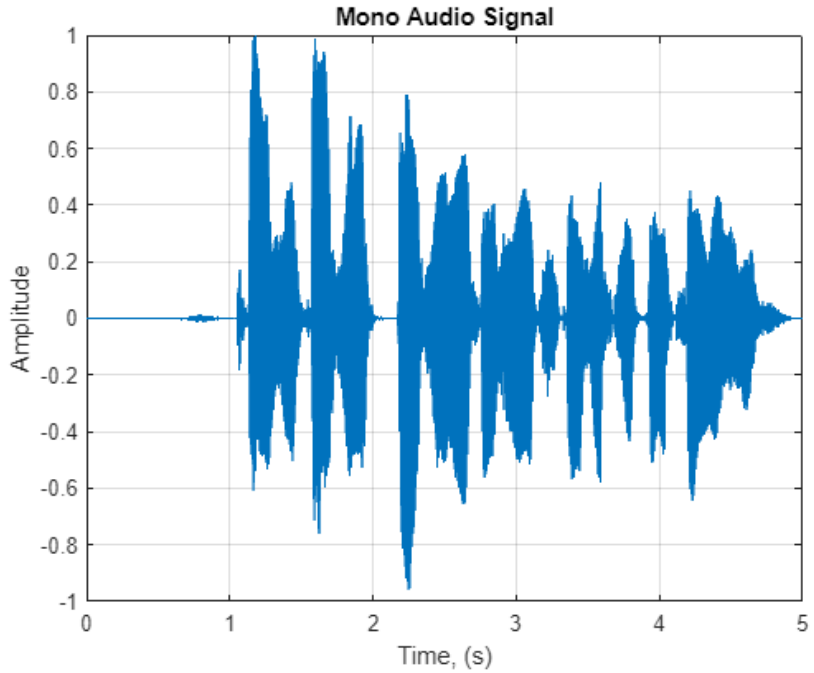

Figure 10: Recording of " Crazy Fredrick bought many very exquisite opal jewels " plotted in time.

When plotted in time, the beginning of words can be clearly seen for each file with a large spike in amplitude after a period of a very small amplitude. The emphasis of a syllables can be discerned too, with “prize” in figure 9 having the emphasis on the p in the beginning, and a large spike there. In that same file, the “promptly” has a large spike at the beginning with the p, then dips in amplitude from the softer r, o, and m in the middle, with another spike from the p and t in the end of the word.

The length of each word can also be seen, particularly with “The quick brown fox jumps over the lazy dog”, the “the” in the beginning and “brown” shortly after have a clear difference in length. From plotting in time, we can tell how long each phrase took to say overall, with “The quick brown fox jumps over the lazy dog” being the shortest at approximately 2.5 seconds from the start of the first word, and the other phrases both taking approximately 4 seconds to say.

Part III: Visualization with a spectrogram

To get a better idea of the audio signal both time and frequency, plotting the signal on a spectrogram can be useful. For this, I created a function called `PlotAudioSpectrogram`, which takes the input of `AudioData`, the sample rate, figure, and maximum frequency.

Similarly to `PlotAudioTime`, an if statement is used with the size command to determine if the signal is mono or stereo. For both mono and stereo cases, the `spectrogram` command is used to plot the spectrogram of audio data using the short-time Fourier transform. For a stereo signal, the program once again sets up the tiled layout dimensions and increments the tile to be plotted to for each channel.

The parameters within the spectrogram command can be changed depending on the level of detail desired for the output. Vector `F` tells the spectrogram command at which frequencies the short time Fourier transform should be performed and plotted. These values can be

adjusted, and in testing I found 200 linearly spaced frequencies gave a good resolution without being too slow when plotting between 0 and 8000Hz.

```
function PlotAudioSpectrogram(Audio, Fs, fig, FMax)
figure(fig);
F=linspace(0, FMax, 200);
    if size(Audio, 2) == 1
        spectrogram(Audio, hamming(512), 128, F, Fs,'yaxis','power')
        title('Mono Audio Spectrogram')
    elseif size(Audio, 2) == 2
        tiledlayout(2, 1);
        nexttile
        spectrogram(Audio(:, 1), hamming(512), 128, F, Fs, 'yaxis', 'power')
        title('Audio Left Spectrogram')
        nexttile
        spectrogram(Audio(:, 2), hamming(512), 128, F, Fs, 'yaxis', 'power')
        title('Audio Right Spectrogram')
    end
end
```

Figure 11: PlotAudioSpectrogram function in MATLAB. The spectrogram command is currently set to use a hamming window and 128 overlap samples between segments. If the overlap was left blank, it would default to ½ the window length, which in this case would be 256 samples because the window length is 512 samples.

For plotting the files that were recorded in section II, the following code was run in MATLAB with the proceeding output spectrograms:

```
FileName = 'ECE 2312 C24\TQBF Jumps Over Lazy Dog.wav'
[AudioData, SampleRate] = audioread(FileName);
PlotAudioSpectrogram(AudioData, SampleRate, f1, 8000)
clf('reset')

FileName = 'ECE 2312 C24\Promptly Judged Antique Ivory Buckets.wav'
[AudioData, SampleRate] = audioread(FileName);
PlotAudioSpectrogram(AudioData, SampleRate, f1, 8000)
clf('reset')

FileName = 'ECE 2312 C24\Crazy Fredrick Bought Opal Jewels.wav'
[AudioData, SampleRate] = audioread(FileName);
PlotAudioSpectrogram(AudioData, SampleRate, f1, 8000)
```

Figure 12: Code used to plot spectrograms in MATLAB. The maximum frequency to be plotted is 8kHz. Like when plotting in time, the clf('reset') command is added to make sure MATLAB clears the figures before plotting the next one.

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

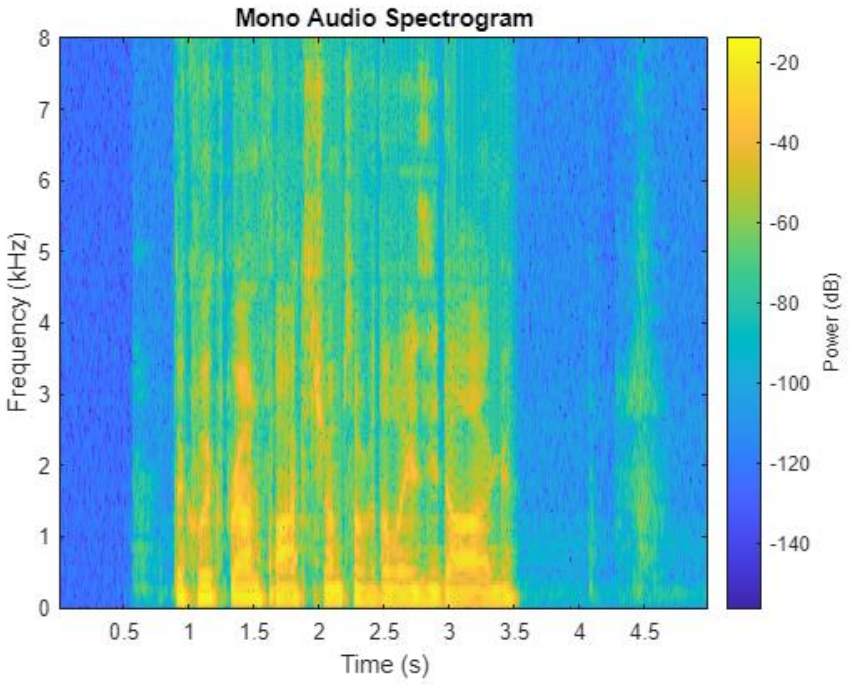

Figure 13: Spectrogram of "The quick brown fox jumps over the lazy dog."

```
FileName = 'ECE 2312 C24\Promptly Judged Antique Ivory Buckets.wav'
```

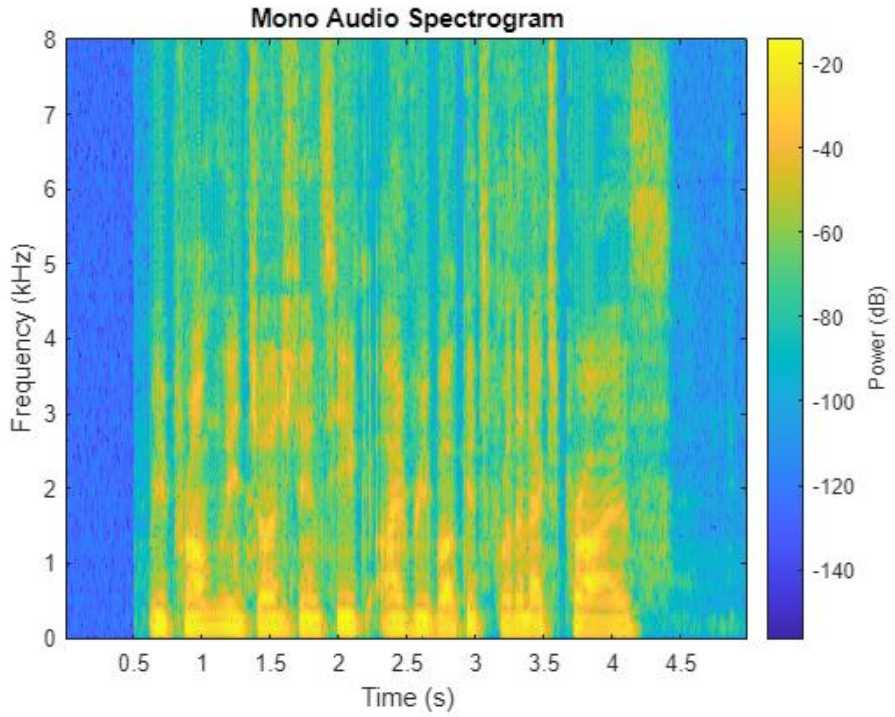

Figure 14: Spectrogram of " We promptly judged antique ivory buckets for the next prize."

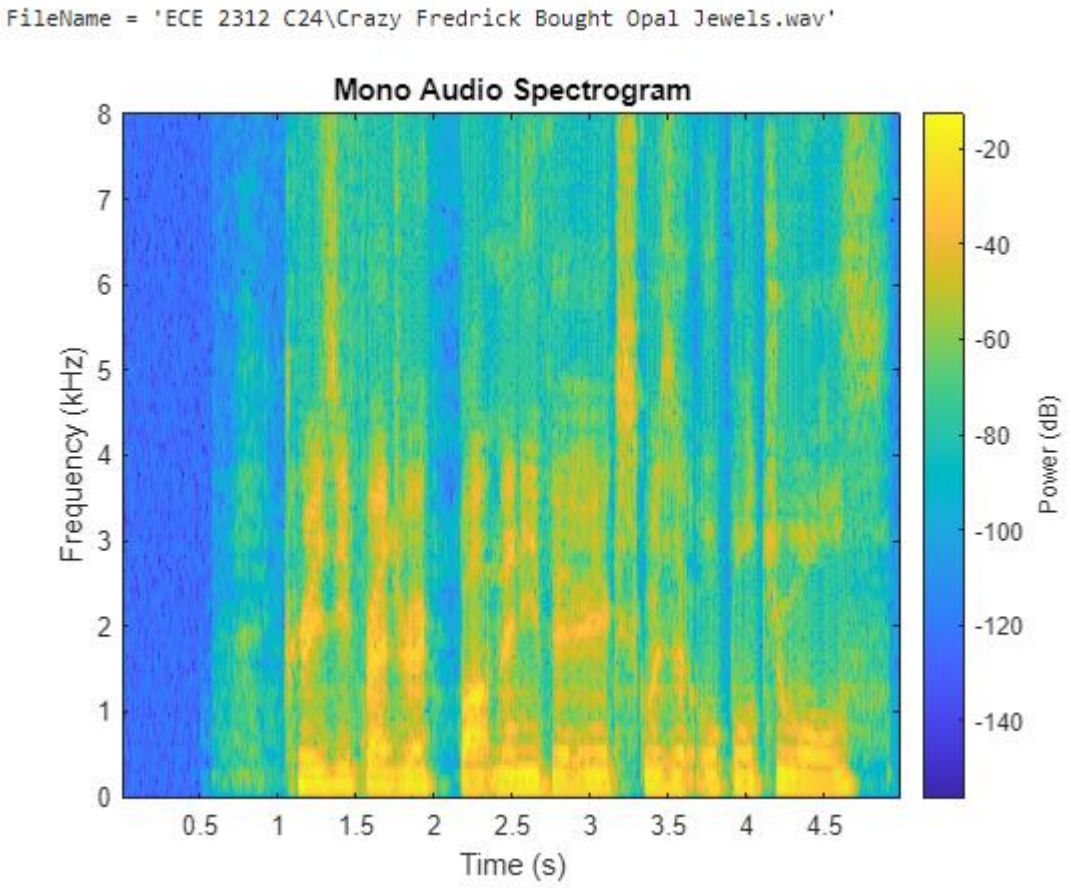

Figure 15: Spectrogram of "Crazy Fredrick bought many very exquisite opal jewels." Clear spikes in frequency can be observed on syllables like t and k, like the amplitude spikes when plotted in time.

Some clear differences can be seen between the frequency of the signal and the words spoken, in figure 15, there is a large spike of frequencies between 5-8kHz at around 1.25 seconds in, this aligns with when the ck at the end of 'Fredrick' is spoken. A similar spike occurs at the end of 'exquisite' when the t syllable is pronounced. The soft syllables of 'jewels' also seem to mostly contain low frequencies, with the j at the beginning of the word only having a small spike in frequency at around 3kHz.

Like when plotting the signal in time, in figure 14 the beginning and end of 'promptly' can be clearly seen with the spike in frequency for the p and t syllables. The word 'judged' also

exhibits this behavior but is slightly less pronounced. A similar pattern is in figure 13, with the j in the beginning of 'jumps' having a pronounced spike in frequency as well.

Overall, it seems like harder syllables, like p, t, j, and c/k tend to have a higher frequency than softer ones like o, u, m, and e when I am speaking. The beginnings and ends of words tend to have a spike in frequency too, though that may be because harder vowels are often placed in the beginning or end of a word.

In figure 2 from the project assignment PDF, there is a clear difference between frequencies of each part of 'do-re-mi-fa-so-la-ti-do'. Assuming the notes are not in the correct order (ie: not lowest to highest pitch), the first do likely occurs at ~1.5s due to the low frequencies and energy. The second note, re, being slightly higher in pitch than do, likely occurs at ~3.25s. The next note, mi, probably occurs at ~0.75s, with fa likely occurring at 2.5s because of the large spike above 4kHz. The 6th note of the scale, so, could occur at 1s, but this is difficult to be sure about, with the 2nd to last note being at 2s because of the large amount of frequencies above 1kHz, and the highest note, the 2nd do occurring around 3.75s.

Part IV: Converting a Mono Signal to Stereo & Stereo Delay

So far, we have been recording and working with mono audio files, however having a stereo file may be useful, like in the case of wanting a delay between what one ear hears and what the other hears. To implement this, I created a function called `StereoDelay`.

The function `StereoDelay` takes an input of `AudioData`, the sample rate, the delay length in ms, and the channel to be delayed (`1` for left, `2` for right). First, the function calculates the number of samples per millisecond, `FsPerms`, by dividing the sample rate in hertz by 1000. Next, the function multiplies `FsPerms` by the delay in milliseconds, and rounding that value to the nearest whole number to get `NumZeros`, which is the number of zeros we will add to the signal to get the desired delay length.

Like `PlotAudioTime` and `PlotAudioSpectrogram`, an if statement is used along with the `size` command to determine if the data is mono or stereo. Within that statement, another if statement is used for each side of the stereo delay.

Since MATLAB R2023b, there is a built-in command called `paddata`, which makes adding values to the beginning or end of a vector or matrix simple to implement. For this use case, we are using 3 inputs of `paddata`. First, the input audio signal we want to pad, next the final desired length of the output, and finally, if we want to pad either the beginning or end of the column.

```
function Audio = StereoDelay(Audio, Fs, Delay, Side)
FsPerms = Fs/1000;
NumZeros = round(FsPerms*Delay);
if size(Audio, 2) == 1
    if Side == 1
        Audio = [paddata(Audio, NumZeros+length(Audio), Side='leading'),...
            paddata(Audio, NumZeros+length(Audio), Side='trailing')];
    elseif Side == 2
        Audio = [paddata(Audio, NumZeros+length(Audio), Side='trailing'),...
            paddata(Audio, NumZeros+length(Audio), Side='leading')];
    end
elseif size(Audio, 2) == 2
    if Side == 1
        Audio = [paddata(Audio(:, 1), NumZeros+length(Audio),Side='leading'),...
            paddata(Audio(:, 2), NumZeros+length(Audio), Side='trailing')];
    elseif Side == 2
        Audio = [paddata(Audio(:, 1), NumZeros+length(Audio),Side='trailing'),...
            paddata(Audio(:, 2), NumZeros+length(Audio), Side='leading')];
    end
end
end
```

Figure 16: StereoDelay function in MATLAB. Checking the size of the 2nd dimension of the audio makes sure that the correct operations are performed for either a stereo or mono signal. The cases for each side lets us input which channel the delay should be added onto for the desired effect.

For a mono signal with the `DelaySide` set to 1:

Using `paddata`, column 1 (left channel) is set equal to the input signal, the original signal's length plus the number of zeros needed to get the desired delay amount, with those zeroes added to the leading side of that column to delay the left channel. For the right channel (column 2), the same process is used, except the trailing side of the original data has zeroes added to make the lengths of each column the same.

For a mono signal with the desired `DelaySide` set to the right channel, all that is changed is the output column 1 is now set to trailing, and column 2 is set to leading to delay the right channel.

If our input is a stereo signal, the code for is the same as if it was a mono signal, except with `(:, 1)` added after `Audio` for the output of column 1, which tells MATLAB that for `paddata`, we want use the data of column 1 of the input. Similarly, for column 2 of the output `(:, 2)` is added to ensure only the data of column 2 of the input is used. Without this, the `paddata` command would pad both columns on the input matrix, causing the output to have 4 columns.

A side effect of using `paddata` is if the input signal had the left channel delayed by 1000ms, and the `DelayLength` was also set to 1000ms, the output signal would have the left channel delayed by a total of 2000ms.

Because sound travels at a speed of approximately 343m/s, if there is a sound coming from someone's right, there will be a small delay between the right ear receiving the sound and the left ear. Our brains use that delay, along with the volume of the sound received by each ear to determine the direction the sound is coming from.

To approximate this effect, we can add a delay equal to the amount of time it would take for a sound wave to travel from the right ear to the left ear. Using a string wrapped around my head, I measured the circumference to be equal to 21 inches, or 0.5334m. By dividing the

circumference by pi, the diameter of a circle can be calculated, and in this case, is approximately the distance between my ears. By dividing the resulting value by 343m/s gives the time it takes for a sound to travel from one ear to another in seconds. Multiplying this value by the sample rate (in this case, 44.1kHz) divided by 1000 and rounding to the nearest whole number gives the number of samples it would take for sound to travel from one side of someone's head, which at 44.1kHz is 22 samples, or ~0.5ms.

Below is the code implemented to convert one of the sound files that was recorded earlier to stereo, and add stereo delay to the right channel of the audio:

```
FileName = 'ECE 2312 C24\Crazy Fredrick Bought Opal Jewels.wav';
[AudioData, SampleRate] = audioread(FileName);
DelayLength = 0; %Delay = 0ms
AudioData = StereoDelay(AudioData, SampleRate, DelayLength, DelaySide);
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-0ms-0dB.wav', AudioData, SampleRate);

[AudioData, SampleRate] = audioread(FileName);
DelayLength = ((0.5334/pi)/343)*1000; %Delay = 0.5ms
AudioData = StereoDelay(AudioData, SampleRate, DelayLength, DelaySide);
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-avghead-0dB.wav', AudioData, SampleRate);

DelayLength = 10; %Delay = 10ms
[AudioData, SampleRate] = audioread(FileName);
AudioData = StereoDelay(AudioData, SampleRate, DelayLength, DelaySide);
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-10ms.wav', AudioData, SampleRate);

DelayLength = 100; %Delay = 100ms
[AudioData, SampleRate] = audioread(FileName);
AudioData = StereoDelay(AudioData, SampleRate, DelayLength, DelaySide);
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-100ms.wav', AudioData, SampleRate);
```

Figure 17: Code in MATLAB used to generate and write files with the desired amount of delay. Because AudioData gets overwritten when the delay is added, the original file must be read back into MATLAB for each delay amount.

Attenuation to the right channel can also be added to potentially make the effect of something sounding like it comes from the left be more convincing. By multiplying the right channel by 0.5, the signal is attenuated by approximately -3dB, 0.707 for approximately -1.5dB, and 0.25 for approximately -6dB. Below is the code used to attenuate the file with a 0ms delay, and file with the delay from the distance from one ear to another.

```
[AudioData, SampleRate] = audioread('ECE 2312 C24\Kara Giordano-stereosoundfile-0ms-0dB.wav');
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-0ms-1.5dB.wav', ...
    [AudioData(:,1), 0.707*AudioData(:,2)], SampleRate);

audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-0ms-3dB.wav', ...
    [AudioData(:,1), 0.5*AudioData(:,2)], SampleRate);

audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-0ms-6dB.wav', ...
    [AudioData(:,1), 0.25*AudioData(:,2)], SampleRate);


[AudioData, SampleRate]  = audioread('ECE 2312 C24\Kara Giordano-stereosoundfile-avghead-0dB.wav');
audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-avghead-1.5dB.wav', ...
    [AudioData(:,1), 0.707*AudioData(:,2)], SampleRate);

audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-avghead-3dB.wav', ...
    [AudioData(:,1), 0.5*AudioData(:,2)], SampleRate);

audiowrite('ECE 2312 C24\Kara Giordano-stereosoundfile-avghead-6dB.wav', ...
    [AudioData(:,1), 0.25*AudioData(:,2)], SampleRate);
```

Figure 18: Code in MATLAB for generating the attenuated files. Attenuation is done simply by writing AudioData with the 2nd column multiplied by a scalar to the disk. Doing this directly within the audiowrite command removes the need to load the original file back into AudioData.

After listening to each of the files that was created from the code in figure 17 and 18, there are some interesting distinctions between them and the stereo effect they provide. First off, the file with the 100ms delay has a clear time miss match between the ears, and it just sounds like the sound coming from both sides with an echo. The file with the 10ms delay has a very 'wide' sound, like taking a synthesizer and detuning the left and right channel from each other to give it a full and wide sound. While the effect is cool, it only sounds like the source of the sound is slightly to my left, if the direction I was facing was 12 o'clock on a clock, it sounds like it's coming from 11 or 10:30.

The file with the delay of ~0.5ms to match the time it takes for sound to go the distance of my head and with a 0dB attenuation sounding like it was from the 10 o'clock position. The -1.5dB, -3dB, and -6dB files progressively shifted more towards the left, with the -6dB file sounding like something was coming directly from my left. The 0ms file was clearly centered, with each subsequent reduction in the amplitude of the right channel causing the sound to shift more to the left.

The slight stereo delay of the ~0.5ms file made it overall sound more spacious than the one without any delay, providing a slightly wider sound that felt more realistic and less artificial, especially when using headphones. The 'artificial' sound of the 0ms file just being attenuated had the same effect as just adjusting the panning slider on a file within an audio editing program, only moving where the sound was coming from without providing any extra space or width that sound in the real world has.

Overall, the file with the ~0.5ms delay with -3dB of attenuation sounded the most like it was coming from the left to me. The -6dB version of the file had a similar problem to the one with no delay and just the attenuation, sounding like there wasn't much spaciousness to the sound and feeling artificial. I think this is because the -6dB attenuation is a little too much to the point where it's difficult to hear the right channel, causing that artificial and less spacious sound.

For an improvement on the effect, maybe a very slight reverb and echo can be added to the file as a whole. When experiencing sound in the real world, different rooms, and spaces each have slightly different acoustics to each other, causing reverb and echo. Mimicking those effects by modifying a sound file could make the sound more realistic, as if someone is talking to you in one of those spaces.

Adding more speakers than just the left and right ones would be helpful for creating a more surrounding environment for sound. The addition of front and back speakers along with the traditional left and right speakers would allow for a whole new set of options and effects for creating an experience like sound in the real world. While this would add more complexity, processing, and file size to hold the extra data, interesting and diverse soundscapes could be created to give the illusion of being somewhere the listener is not.