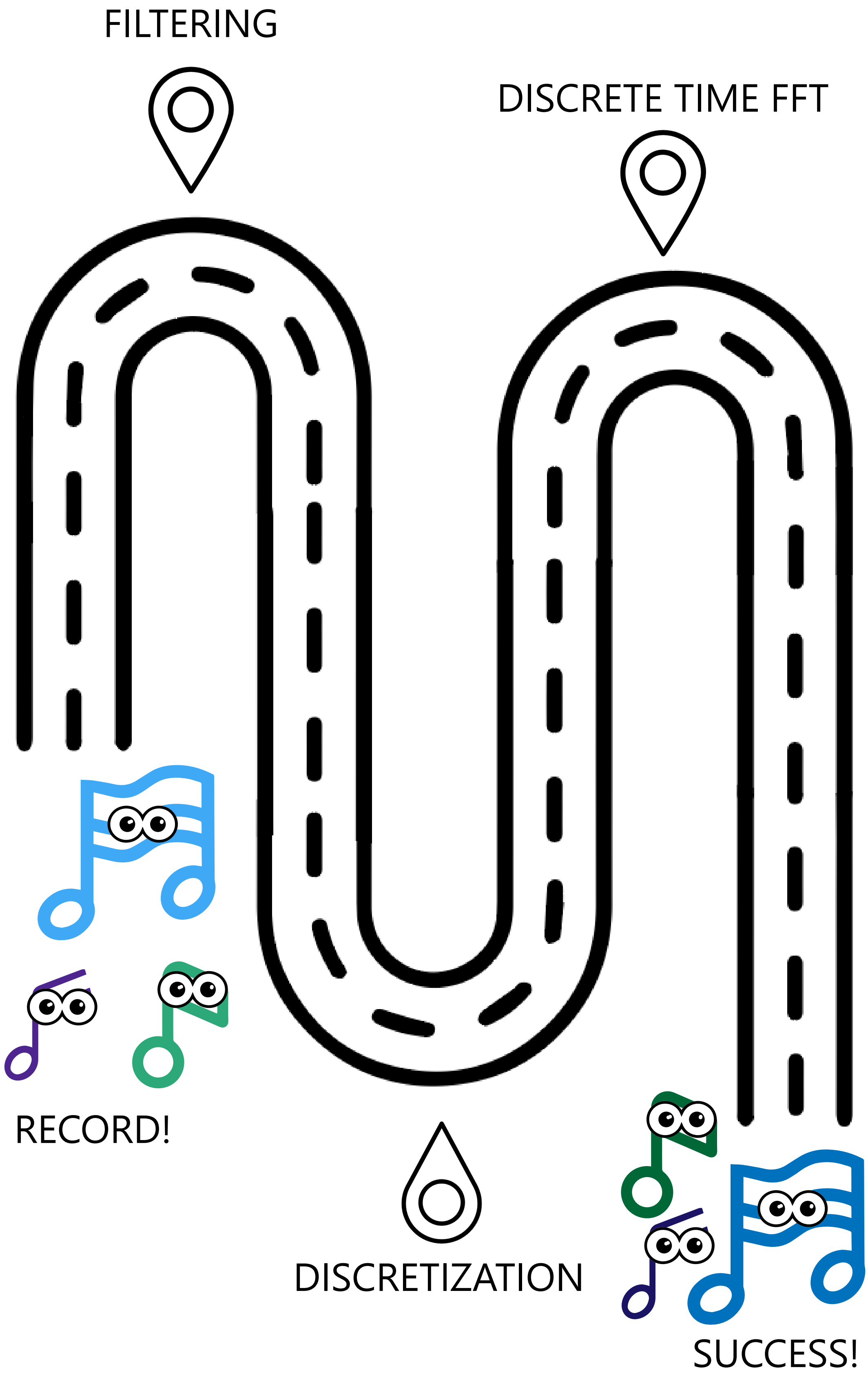
NOTE ROAD

GET STARTED

HERE'S A MAP OF OUR JOURNEY!

Record your unique sound and watch as it transforms!
Click on each step along the way to learn more!



RECORD A UNIQUE SOUND

CLICK THE MICROPHONE TO START

DONE!

CLICK THE MICROPHONE TO CONTINUE

FILTERING

WHAT IS FILTERING AND WHY DO WE DO IT?

Filtering is when we try to take out signals which don't meet certain standards or don't fall within a criteria which we specify. We filter in order to remove noise from signals and only listen to the sounds we most care about hearing.

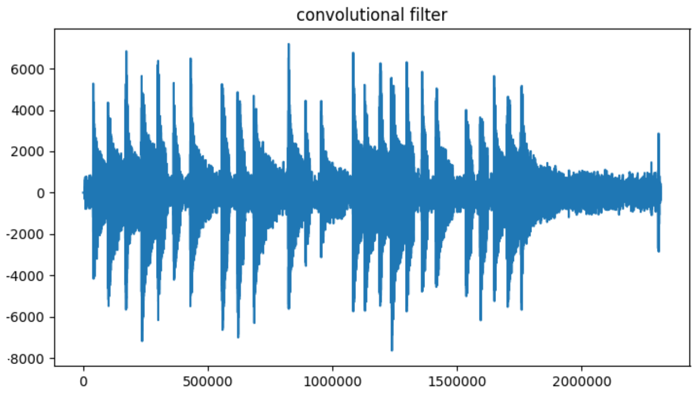
WHAT IS BAND PASS FILTERING?

Band pass filters are filters which only let in signals within a certain range. When filtering this note, we look specifically at only signals that fall in between two frequencies. Every other sound that doesn't fall within those bounds is taken out.

WHAT IS CONVOLUTIONAL FILTERING?

Convolution is a filter that suppresses or enhances components of the signal by taking a running average. Given a sound input, the output is a sound of the same frequency but an amplitude and phase closer to that of the running average. This creates a more normalized sound.

YOUR FILTERED SIGNAL:



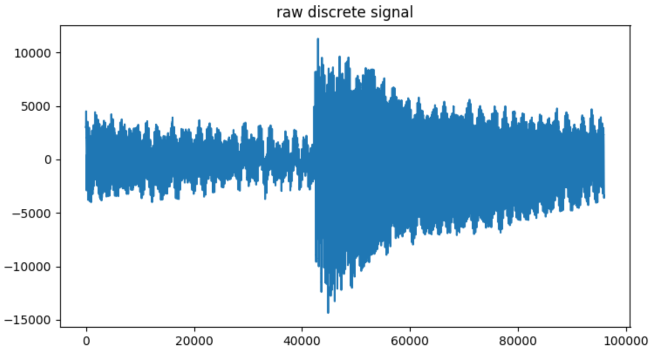
BACK

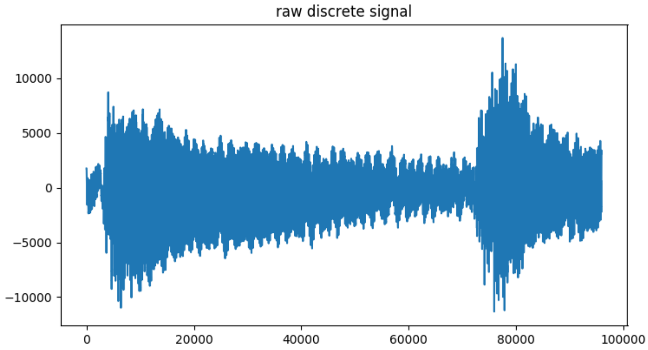
DISCRETIZATION

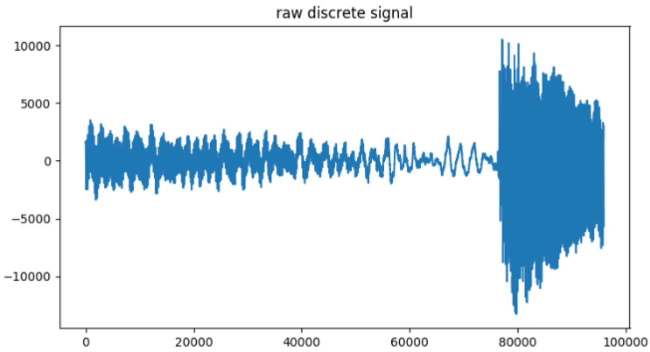
WHAT IS DISCRETIZATION?

Discretization is the process of turning continuous signals into discrete chunks. Transforming a signal into discrete time spaces is useful because it allows you to independently process and compare the samples.

YOUR DISCRETIZED SIGNALS:







BACK

DISCRETE-TIME FT

WHAT IS A FOURIER TRANSFORM?

A fourier transform is a change in perspective from a complete signal to its parts. It allows you to analyze a signal and learn about it's attributes. Fourier transforms change the axes of a regular sound wave from amplitude vs time to amount of a frequency vs frequency.

WHAT IS DISCRETE-TIME FT?

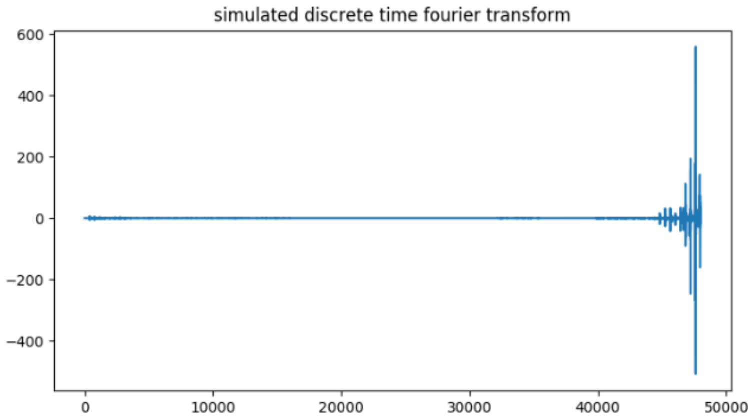
A discrete-time fourier transform is a special type of fourier transform that uses a function of continuous frequency that is broken into discrete time samples.

HOW DOES IT WORK?

Discrete Time Fourier Transforms work similarly to fourier transforms, except the math is a bit different. The way in which signals are added together is done through a sum (as opposed to an integral which adds things continually).

REVEAL MATH

YOUR UNIQUE SIGNAL:



BACK

THE MATH BEHIND DTFT

The following equation shows how a regular continuous-time signal can be represented in the frequency domain.

$$X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\,dt.$$

A discrete-time signal is summable meaning: $\sum_{n=-\infty}^{\infty} |x[n]| < +\infty$

has a DTFT X(Ω), -∞ < Ω < ∞, that can be written as:

$$X(\Omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-jn\Omega}$$

Notice that this is very similar to the continuous-time signal above, the only difference is the integral. This is because the integral is taking the sum of all of the entries continuously, but the discrete time fourier transform adds up all of the entries in discrete chunks, thus the summation symbol.

HOW IS DTFT DIFFERENT THAN DFT?

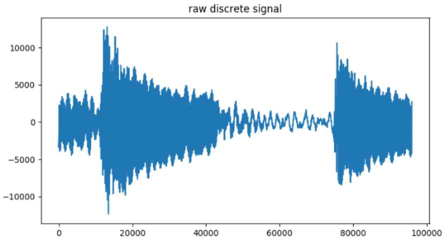
The Discrete Fourier Transform is a type of DTFT. The discrete fourier transform is similar to the DTFT except that it uses discrete frequency points instead of a continuous frequency. It is often used to analyze signals on computers because computers work on a finite number of points.

DTFTs can be simulated on computers by including a large number of samples. As the number of samples gets closer to infinity, the frequency domain becomes a continuous signal. We tried to increase the number of samples in small amounts of time to create things that are similar to DTFTs. It is also important to say that the DFT is just a DTFT which is sampled at equally spaced frequency points.

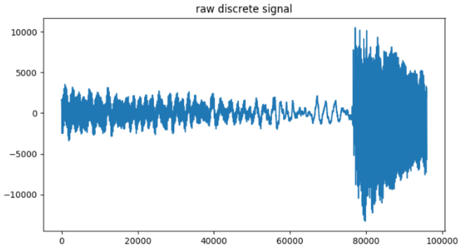
WE FOUND OUR NOTE!

Using the frequency vs. amplitude graphs we generated using discrete-time fourier transforms, we can find note being played in a given sample.

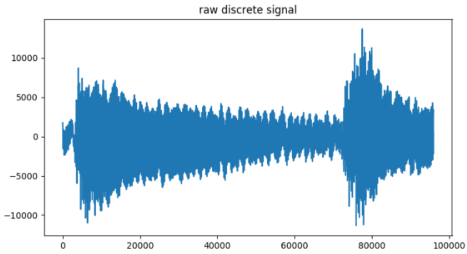
CORRESPONDING NOTE



→ A5



→ C5



→ G4

Map Your Adventure Reflection

Dhara and Maia

10, December 2019

For this assignment, we decided to create an experience for a child which covered content like Discrete Time Fourier Transforms and Convolutional filtering in an engaging and informative way. To do that, we thought about music and how it could relate to these topics in a way that would be entertaining for a child. From there, we decided to document a music note in the midst of an identity crisis and show how using these techniques could lead us to identify music notes.

We did this by analyzing a sample song (Mary had a Little Lamb) and visualizing the data in matlab and python at each step in the process as well as creating an app experience which has more information about each of these steps. The app outlines why the steps are important and how they work.

The steps of the app and analysis are: taking in a sound, filtering that sound, separating that sound into discrete time sections, running discrete time fourier transforms on each discrete time section, and identifying the note from the resulting discrete time fourier transform.

We hit two main areas of technical depth in this project (and reviewed much of the content we were a little less sure of ourselves in!) The first was convolutional filtering as a means of digital signal processing. To be honest, we learned tons about digital signal processing in an effort to create a filter which we liked and did what we needed it to. This is most evident in our code, both in the matlab where we are currently doing our filtering, and in the python where we have left many of the filtering methods we tried in the comments. The second major point of learning was a detailed understanding of discrete time fourier transforms and how they differ from the discrete fourier transform and the continuous fourier transform. This is especially evident in the way in which discrete time fourier transforms are explained in our app and the way in which we outline the math. One of the biggest challenges we faced was trying to relay this extremely complex technical content to a non technical audience. We hope that we did a decent job in that endeavour.

The analysis was mostly done in python, with the filtering completed in MATLAB. This allowed us many visualization options for our final graphs. At the moment, the name of the note is the title for plots which depict each step in the filtering process, making it intuitive to see how the signal changes over time to reach a solution.

FILTERING AND DISCRETE-TIME FFT

```
[x, Fs] = audioread("lamb.wav");
sound(x, Fs);

N = length(x);
f = linspace(-Fs/2, Fs/2 - Fs/N, N) + Fs/(2*N)*mod(N, 2);

X = fft(x);
```

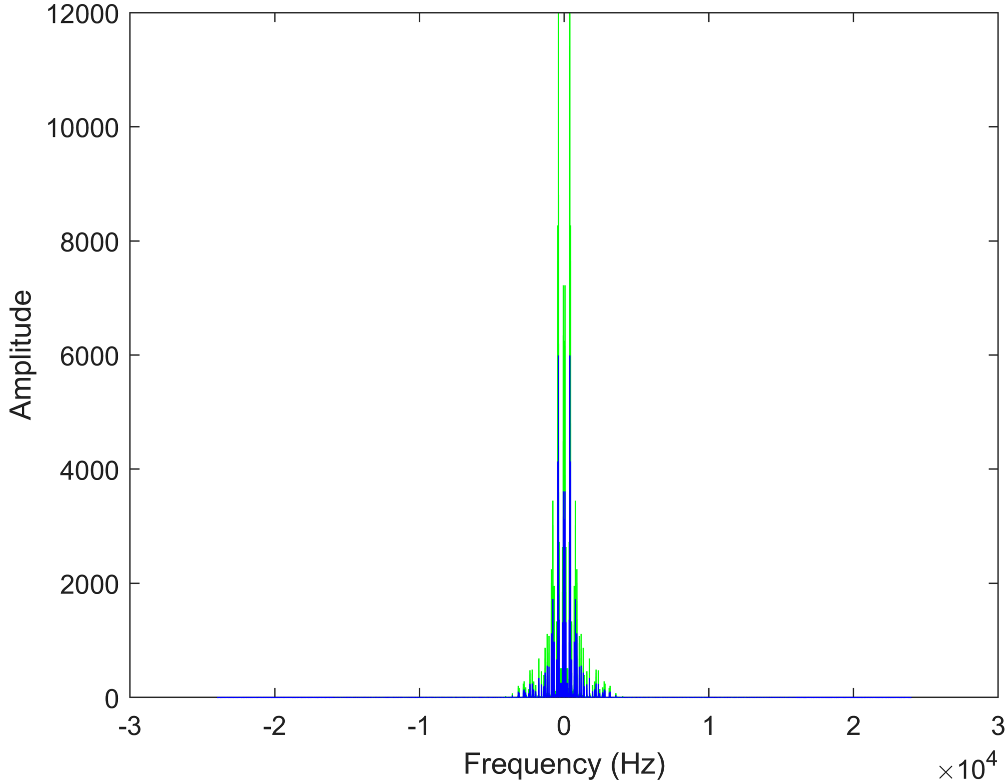
CONVOLUTION FILTER:

```
xSmooth = conv2(x, 0.5, "same")
```

```
xSmooth = 1159158×2
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     ⋮
```

```
sound(xSmooth, Fs);
XSmooth = fft(xSmooth);

%frequency vs. amplitude of filtered and unfiltered
plot(f,fftshift(abs(X)),'g',f,fftshift(abs(XSmooth)),'b');
xlabel('Frequency (Hz)');
ylabel('Amplitude')
```



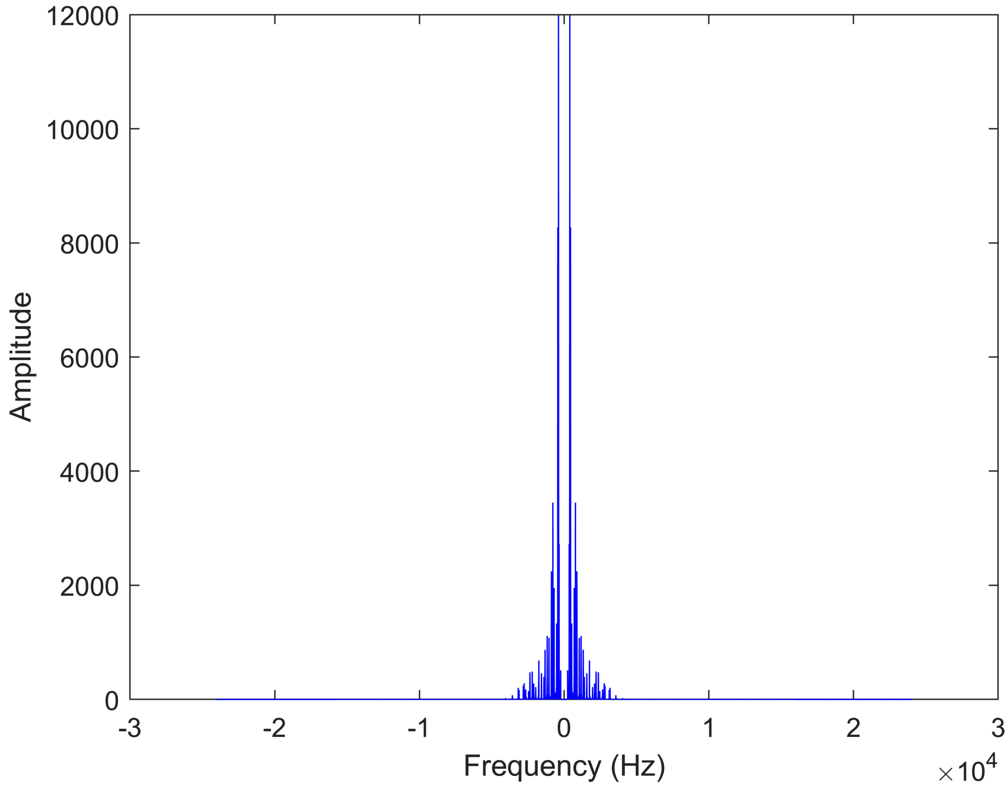
BAND PASS FILTER:

```
xBand = bandpass(x,[.01, .999])
```

```
xBand = 1159158×2
   -0.1254   -0.1254
   -0.1248   -0.1248
   -0.1240   -0.1240
   -0.1232   -0.1232
   -0.1222   -0.1222
   -0.1212   -0.1212
   -0.1200   -0.1200
   -0.1187   -0.1187
   -0.1173   -0.1173
   -0.1157   -0.1157
      ⋮
```

```
sound(xBand, Fs);
XBand = fft(xBand);

%frequency vs. amplitude of filtered and unfiltered
plot(f,fftshift(abs(XBand)),'b');
xlabel('Frequency (Hz)');
ylabel('Amplitude')
```




```
%DTFT
omega = linspace(-pi, (pi*(1-2/N)),N)
```

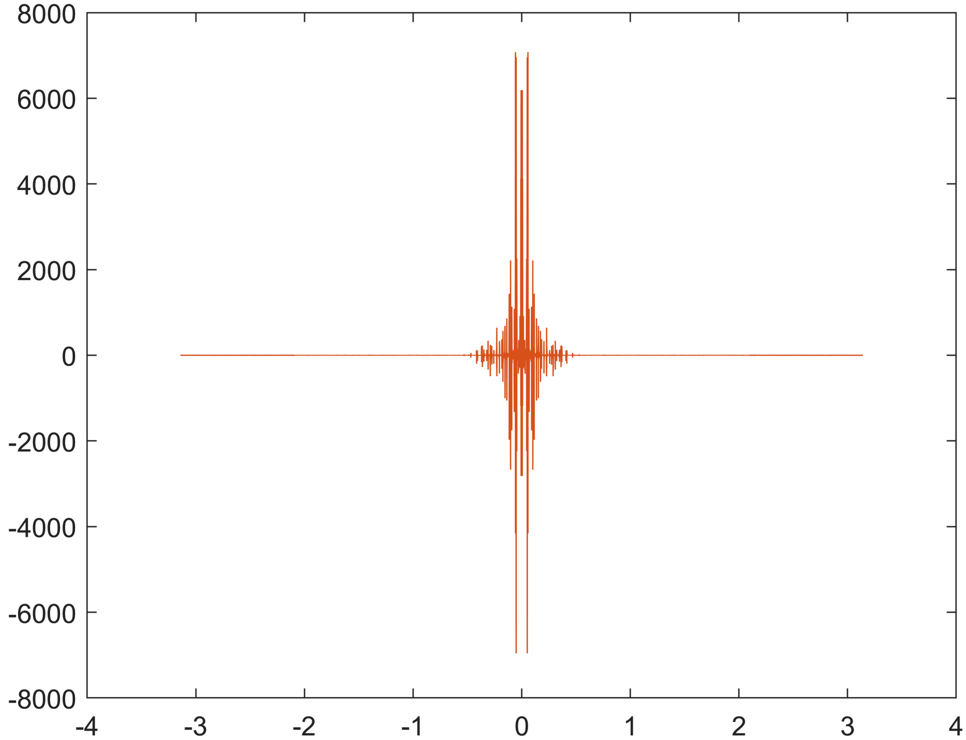
```
omega = 1×1159158
   -3.1416   -3.1416   -3.1416   -3.1416   -3.1416   -3.1416   -3.1416   -3.1416 ···
```

```
c = fftshift(fft(fftshift(x)))
```

```
c = 1159158×2 complex
10^4 ×
   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 - 0.0000i   0.0000 - 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i
   0.0000 - 0.0000i   0.0000 - 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i
  -0.0000 + 0.0000i  -0.0000 + 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i
  -0.0000 - 0.0000i  -0.0000 - 0.0000i
   0.0000 - 0.0000i   0.0000 - 0.0000i
      ⋮
```

```
plot(omega, c)
```

```
Warning: Imaginary parts of complex X and/or Y arguments ignored
```




```
audiowrite("bandpass.wav",xBand,Fs);
audiowrite("convolution.wav",xSmooth,Fs);
```

Visualizer Code

```
import numpy as np
import sounddevice as sd
from matplotlib import pyplot as plt
import pylab as pl
from pydub import AudioSegment
from pydub.playback import play
from scipy import signal, fft, fftpack
from scipy.io.wavfile import write, read

class SoundProcessing:

    def __init__(self):
        pass

    def record_sound(self):
        '''
        Not taking in our own sound, but if we were this method would process the file to make it usable for us
        '''
        fs = 44100  # Sample rate
        seconds = 90  # Duration of recording

        myrecording = sd.rec(int(seconds * fs), samplerate=fs, channels=2)
        sd.wait()  # Wait until recording is finished
        write('output.wav', fs, myrecording)  # Save as WAV file


    def break_up_sound(self):
        audio_chunk_list = []
        audio_file = "lamb.wav"
        audio = AudioSegment.from_wav(audio_file)
        print(type(audio))
        list_of_timestamps = [1, 2, 3, 4, 5 ,6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24] #and so on in *seconds*

        start = 0
        for  idx, t in enumerate(list_of_timestamps):
            #break loop if at last element of list
            if idx == len(list_of_timestamps):
                break

            end = t * 1000 #pydub works in millisec
            print("split at [{}:{}] ms".format((start/1000), (end/1000)))
            audio_chunk = audio[start:end]
            audio_chunk.export("audio_chunk_{}.wav".format(end), format="wav")

            start = end #pydub works in millisec
            print(type(audio_chunk))
            print(audio_chunk.max_dBFS)
            print(audio_chunk.max_possible_amplitude)
            audio_chunk_list.append(audio_chunk)
        return audio_chunk_list

class MusicClip:

    def __init__(self, clip):
        self.music_clip = clip

    def filter(self):
        '''
        We ended up doing our filtering in matlab, but this method shows some of the many filters and ways in which we tried to filter in python. Ultimately, we decided that MATLAB would allow us more flexibilty
        '''
        num_array = AudioSegment.get_array_of_samples(self.music_clip)
        # print(num_array)
        print(np.shape(num_array))

        #hilbert transforms
        # plt.plot((np.array(num_array)))
        # # plt.plot(np.real(signal.hilbert(np.array(num_array))))
        # plt.show()

        # print(np.real(signal.hilbert(np.array(num_array))))
        # filtered = signal.hilbert(np.array(num_array))
        # plt.plot(filtered, label = 'hilbert filtering ')
        # plt.show(block = False)
        num_array = np.array(num_array)
        num_array = num_array/1000
        # act_num_array =[]
        # for row in np.real(signal.hilbert(np.array(num_array[1]))):
        #     print(row)
            # act_num_array.append(row[1])

        max_sound = np.asarray(num_array).max()
        print(np.asarray(num_array).max())
        print(max_sound)
        print((np.asarray(num_array).max() == max_sound))
        print(type(num_array))

        # hann window
        # print(num_array.si)
        # win = signal.hann(np.size(num_array/2), False)
        # # plt.plot(win)
        # # plt.show()
        # win = win*(max_sound)
        # # plt.plot(win)
        # # plt.show()
        # print(np.shape(num_array))
        # # win = signal.hann(20)
        # print(type(win))
        # # plt.plot(num_array)
        # # # # plt.show()
        # # plt.plot(win)
        # plt.show(block = False)

        # time = np.linspace(0,1,np.size(num_array))
```

```
        # high pass filters with fourier transforms
        # W = fftpack.fftfreq(np.size(num_array))
        # print(W)
        # f_signal = fftpack.rfft(num_array)

        # cut_f_signal = f_signal.copy()
        # cut_f_signal[(W>200000000000000000000)] = 0
        # cut_signal = fftpack.irfft(cut_f_signal)

        # plt.subplot(221)
        # plt.plot(time,num_array)
        # plt.subplot(222)
        # plt.plot(W,f_signal)
        # plt.xlim(0,1)
        # plt.subplot(223)
        # plt.plot(W,cut_f_signal)
        # plt.xlim(0,1)
        # plt.subplot(224)
        # plt.plot(time,cut_signal)
        # plt.show()

        # plt.plot(fftpack.rfft(num_array))

        # plt.show()

        fs = 44100
        t = np.arange(np.size(num_array))
        fc = 30  # Cut-off frequency of the filter
        w = fc / (fs / 2) # Normalize the frequency
        b, a = signal.butter(7, w, 'high')
        output = signal.filtfilt(b, a, num_array)
        plt.plot(t, output, label='filtered')
        plt.legend()
        plt.show()

        # convoltional ffts to filter
        # filtered = np.fft.fftshift(signal.fftconvolve(num_array, win, mode = 'same'))
        # filtered = filtered/1000000
        # fw = signal.firwin(17,.11, pass_zero='highpass')
        # f3 = signal.sosfilt(fw, num_array)
        # plt.plot(f3)
        # plt.show()

        # butterworth filter
        # butter1, butter2 = signal.butter(15, .99, 'hp', output='ba', analog = True)
        # filtered2 = signal.filtfilt(butter1, butter2, num_array)
        # # plt.plot(num_array)
        # plt.plot(filtered2)

        # plt.show()

        # convolutional filter
        # conv = signal.convolve(num_array, win, mode = 'same')
        # plt.plot((np.array(num_array)), label = 'Signal')
        # plt.show(block = False)
        # plt.plot(win, label = 'hann window')
        # plt.show(block = False)
        # plt.plot(butter, label = 'fft convolve with hann filter')
        # # plt.show(block = False)
        # # plt.plot(conv/1000000, label = 'normal convolution')
        # plt.show()


        filt = write('output_fr.wav', 44100, filtered2)

        # filt = AudioSegment.from_numpy_array(filtered)
        print(filt)


    def run_fft(self):
        data = AudioSegment.get_array_of_samples(self.music_clip)
        play(self.music_clip)
        # signal.freqz()
        fft_d = np.fft.fftshift(fft(np.fft.fftshift(data)))
        fft_norm = fft_d/len(data)
        fft_norm = fft_norm[range(int(len(data)/2))]
        real_d = np.real(fft_d)

        tp = len(data)/1000
        vals = np.arange(int(len(data)/2))
        freq = vals/tp
        # hist_bins, hist_vals = self.music_clip.fft()
        # hist_vals_real_normed = np.abs(hist_vals) / len(hist_vals)
        # plt.plot(hist_bins / 1000, hist_vals_real_normed)
        # plt.xlabel("kHz")
        # plt.ylabel("dB")
        # plt.plot(hist_vals_real_normed)
        # plt.plot(freq, abs(fft_norm))
        print(max(np.real(fft_d))/100000)
        note = self.identify_note(max(np.real(fft_d))/100000)
        print(type(note))

        plt.title(note)
        plt.subplot(221)
        plt.plot(np.real(fft_norm))
        plt.title('simulated discrete time fourier transform')
        plt.subplot(223)
        plt.plot(AudioSegment.get_array_of_samples(AudioSegment.from_wav('lamb.wav')))
        plt.title('raw entire signal')
        plt.subplot(224)
        plt.plot(AudioSegment.get_array_of_samples(AudioSegment.from_wav('convolution.wav')))
        plt.title('convolutional filter')
        plt.subplot(222)
        plt.plot(data)
        plt.title('raw discrete signal')
        plt.suptitle('Note: ' + note)

        # plt.legend()
```

```
        # plt.legend()
        plt.show()
        pass

    def identify_note(self, num):
        notes = [16.35, 17.32, 18.35, 19.45, 20.60, 21.83, 23.12, 24.50, 25.96, 27.50, 29.14, 30.87, 32.70, 34.65, 36.71, 38.89, 41.20, 43.65, 46.25, 49.00, 51.91, 55.00, 58.27, 61.74, 65.41, 69.30, 73.42, 77.78

        # Take in the max of the fft, then run through this
        fr_in = notes.index(min(notes,key=lambda x:abs(x-num)))
        print(min(notes,key=lambda x:abs(x-num)))

        notes_list = ['C0', 'C#0/Db0', 'D0', 'D#0/Eb0', 'E0', 'F0', 'F#0/Gb0', 'G0', 'G#0/Ab0', 'A0', 'A#0/Bb0', 'B0', 'C1', 'C#1/Db1', 'D1', 'D#1/Eb1', 'E1', 'F1', 'F#1/Gb1', 'G1', 'G#1/Ab1','A1', 'A#1/Bb1', 'B1', 'C2', 'C#
        print(str(notes_list[fr_in]))
        #ret val of notes thats closest, make another list that's note names, then find index of notes that right entry is and then find corresponding note
        return notes_list[fr_in]

if __name__ == "__main__":
    sp = SoundProcessing()
    # sp.filter()
    clips = sp.break_up_sound()

    for clip in clips:
        print(clip)
        clip_object = MusicClip(clip)
        # clip_object.filter()
        clip_object.run_fft()
        # break
```