Matt Catalano

Computational Physics and Engineering Final Report

Analyzing Chords with Fourier Transforms

1. Motivation and Background

Analyzing and manipulating music computationally is a challenging task. Many issues present themselves when attempting to analyze the waveform from even an isolated instrument. For example, when an instrument plays a single note, not only is that frequency present to analyze, but there are cascading harmonics creating additional frequencies within a single note. Analyzing an instrument that can play multiple notes at once has this problem compounded even further.

My initial idea was to create a program that could analyze the waveform of an instrument playing a chord, isolate the notes from the chord, determine if any notes are off-pitch, pitch-correct the off-pitch notes, and re-assemble the chord from the newly manipulated notes. This would all have been done using multiple Fourier Transforms to break down the chord from the time domain to the frequency domain and back again. However, it became quickly clear that this goal was too ambitious to pursue in a project of this length, Therefore, instead of this, I will just be performing the first step in this process. I will be recording melodies and chord progressions and attempt to analyze the notes in the melody or the chords in the progression. The goal is to have a program that is capable of fully identifying note/chord changes in the presence of noise and imperfect pitch.

2. Key Equations and Numerical Method

Although there is a lot of code involved in this project, the only particularly computationally complex aspect is the usage of Fast Fourier Transforms. Without FFT's, this project would be doomed from the start. The idea of any Fourier Transform is that it will take a signal given in the time domain (in our case, this would be the input audio file) and convert it to the frequency domain. This is quite useful as it will allow for analysis of the changing frequencies in an audio file over time. As frequencies change, so do notes and chords; the only way to truly know what note is being played in an audio file is to analyze the frequencies present.

The FFT, specifically, is used here because it is substantially more efficient than its derived Transform, the Discrete Fourier Transform or DFT. The DFT can be written as so:

$$Y_n = \frac{1}{\sqrt{2\pi}} \sum_{k=1}^{N} Z^{nk} y_k \quad where \quad Z^{nk} \equiv [(Z)^n]^k$$

This algorithm has a time efficiency of N^2. The FFT takes advantage of symmetry in the transform process to cut down on calculations and has a reduced time efficiency of $N \log_2 N$ (Landau, Paez and Bordeianu). With the FFT, I can take in a raw audio file in the time domain, convert it to the frequency domain, and analyze the frequencies at any given section of the file. This frequency analysis will then inform the decision-making regarding chord identification.

3. Note Analysis

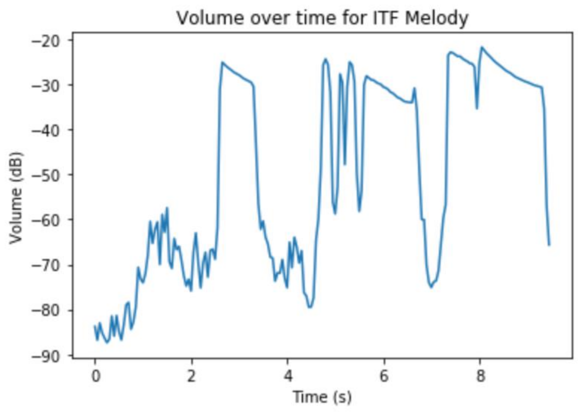
3.A. Crediting sources

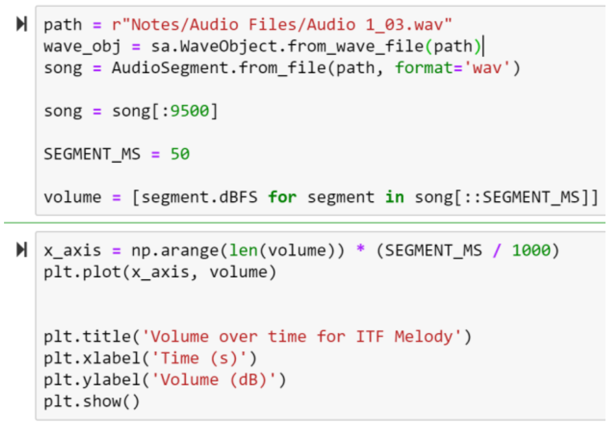
Much of the note analysis code is taken from Ian VonSeggern's blog on Note Recognition in Python (VonSeggern). VonSeggern laid a great foundation for learning how to take in an audio file, utilize Fourier transforms, and extract notes from the input. They did not, however, extend their analysis to chords. I will, of course, be crediting other code and work along the way but thought I should make a special note of VonSeggern's work early as it provided a substantial foundation for the work that I do here.

3.B. Input

The first thing I needed to do was create an audio file. Since my end goal was, originally, to use these advanced techniques to apply harmonic autotune on top of real, polyphonic instruments, I decided the best input I could use was recording myself playing guitar directly onto my computer. Playing the music myself would be better for my analysis than any simulation of a real instrument could be. To record, I used my guitar (a 24-fret PRS whose model name I cannot remember) plugged directly into my interface, a Focusrite Scarlett 2i2. I recorded using the Pro Tools DAW. The melody I recorded is the verse melody for a song I wrote for my band, Sunday Morning Detox, called "It's All Fine". You can find the full song on Apple Music, Spotify, or wherever you listen to music. I know I probably didn't have to say that in this report, but I figured I *had* to promote the music if I'm going to be talking about it for the next several pages here.

I originally recorded the full verse melody of 14 notes to analyze. However, I realized analysis was beginning to take a long time with 14 notes to work on, so I decided to cut it down to the first 7. I used the pydub (Robert) and simpleaudio (Hamilton) Python libraries to take in the .wav file and interpret it. Numpy, scipy, and matplotlib were used for the Fourier transforms, and analysis of results. Figure 1 represents the raw input graphically as volume over time (Figure 2 is the code to get there).





```
path = r"Notes/Audio Files/Audio 1_03.wav"
wave_obj = sa.WaveObject.from_wave_file(path)
song = AudioSegment.from_file(path, format='wav')

song = song[:9500]

SEGMENT_MS = 50

volume = [segment.dBFS for segment in song[::SEGMENT_MS]]
```

```
x_axis = np.arange(len(volume)) * (SEGMENT_MS / 1000)
plt.plot(x_axis, volume)


plt.title('Volume over time for ITF Melody')
plt.xlabel('Time (s)')
plt.ylabel('Volume (dB)')
plt.show()
```

Figure 1: Raw input over time

Figure 2: Code to produce volume over time graph

Once the raw audio file was inputted, I wanted to analytically mark the notes. In this audio file, I obviously knew exactly where the note changes were, so I planned to use analytic markings of where the notes were to check and see how close my code can come to predicting where the notes are. The overall plot with all 7 notes marked analytically is shown in Figure 3.

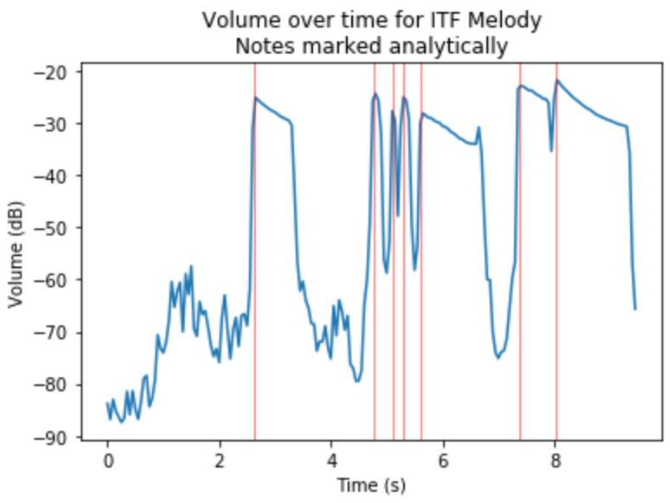
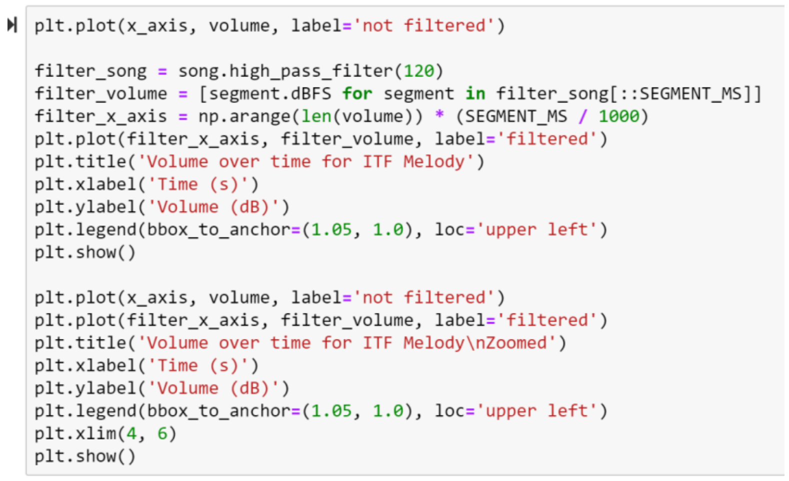


Figure 3: Analytically marked notes in melody

3.C. Filtering Input

My plan to identify note changes is going to rely on noticing sharp changes in volume. To help with this recognition, I decided to first use a high-pass filter on the melody to attenuate any extra noise that could make note recognition challenging. Figure 4 holds the code that is used to generate the plots in Figures 5 and 6. In Figure 5, the filter applied across the entire melody can be seen. Figure 6 is a plot of just the notes from the 4 to 6 second mark. In both figures, however, the peaks of the notes remain largely unchanged while the valleys drop lower. This will aid greatly in identifying note changes.



```python
plt.plot(x_axis, volume, label='not filtered')

filter_song = song.high_pass_filter(120)
filter_volume = [segment.dBFS for segment in filter_song[::SEGMENT_MS]]
filter_x_axis = np.arange(len(volume)) * (SEGMENT_MS / 1000)
plt.plot(filter_x_axis, filter_volume, label='filtered')
plt.title('Volume over time for ITF Melody')
plt.xlabel('Time (s)')
plt.ylabel('Volume (dB)')
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.show()

plt.plot(x_axis, volume, label='not filtered')
plt.plot(filter_x_axis, filter_volume, label='filtered')
plt.title('Volume over time for ITF Melody\nZoomed')
plt.xlabel('Time (s)')
plt.ylabel('Volume (dB)')
plt.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.xlim(4, 6)
plt.show()
```

Figure 4: Filtering Code

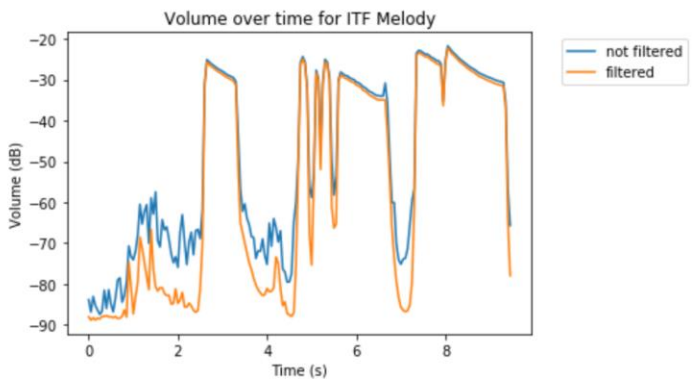


Figure 5: Filtered vs un-filtered melody

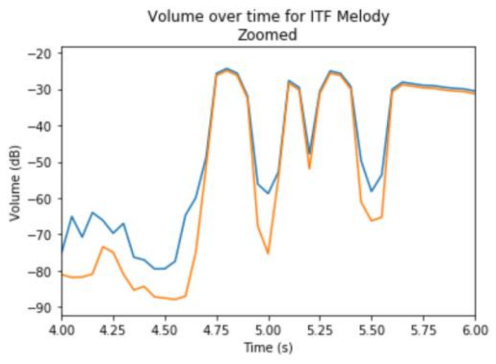
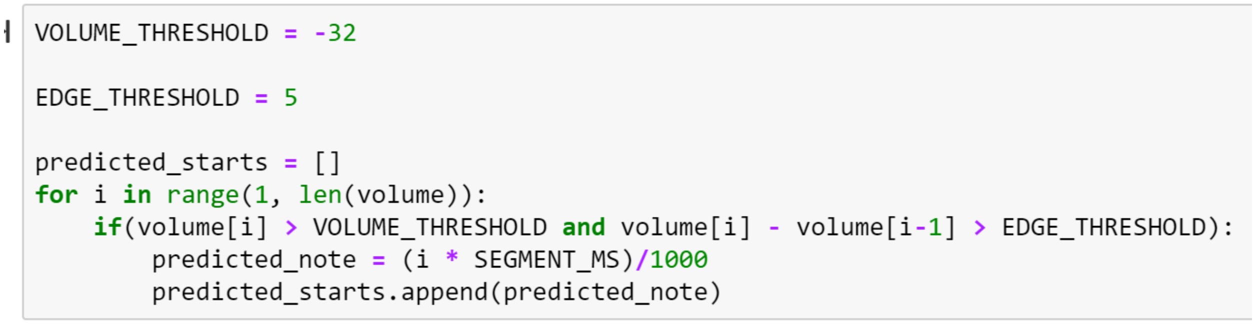


Figure 6: Constraining the x-axis of figure 5

3.D. Identifying Note Changes

Identifying note changes is important for the overall scope of my project, but it is not an inherently challenging computational problem. In the larger scope of my project, a user of the software should be able to input an audio file, without knowing where the note or chord changes occur, and the program should be able to identify where the changes are before analyzing them. After all, if it does not know when different notes/chords appear, it certainly cannot begin to analyze them.

Note changes will be identified by using two thresholds, volume threshold and edge threshold. Volume threshold demands that a sound must be above a certain decibel level to be considered as a note. Edge threshold demands that there must be a sharp increase in volume between the current sample and the previous sample. I had to play around with the values for these thresholds before I landed on something that worked. In the end, I set the volume threshold to -32 dB and the edge threshold to 5 dB. I iterate throughout the entire audio sample, appending predicted notes to an array when they meet the threshold conditions. The code for this can be seen in Figure 7.



```
VOLUME_THRESHOLD = -32

EDGE_THRESHOLD = 5

predicted_starts = []
for i in range(1, len(volume)):
    if(volume[i] > VOLUME_THRESHOLD and volume[i] - volume[i-1] > EDGE_THRESHOLD):
        predicted_note = (i * SEGMENT_MS)/1000
        predicted_starts.append(predicted_note)
```

Figure 7: Threshold code

The results from this initial run are promising, but not perfect. Occasionally, the program appears to be predicting two notes when there is only one. This is likely due to a couple of short, sharp increases in sound at the proper volume threshold as the note is being played. The full first run results can be seen in Figure 8 and some zoomed in plots for increased clarity are provided as Figures 9 and 10.

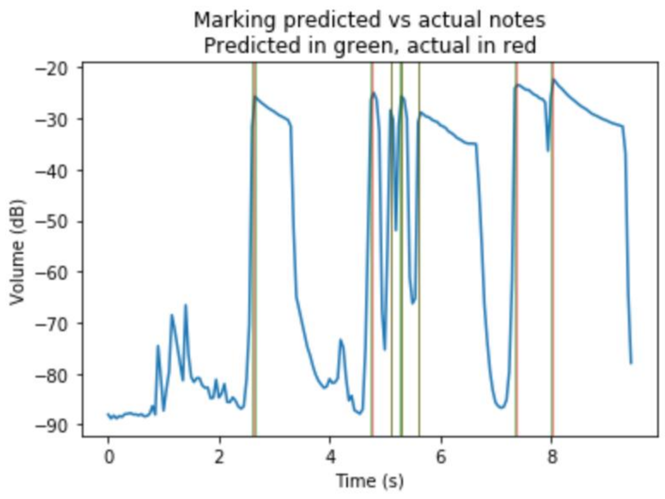


Figure 8: Predicting notes

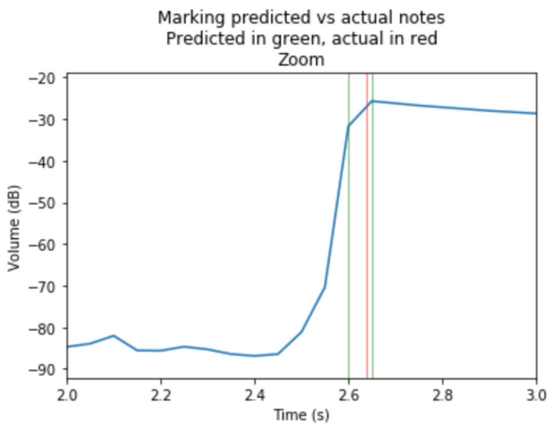


Figure 9: Zooming in on figure 8

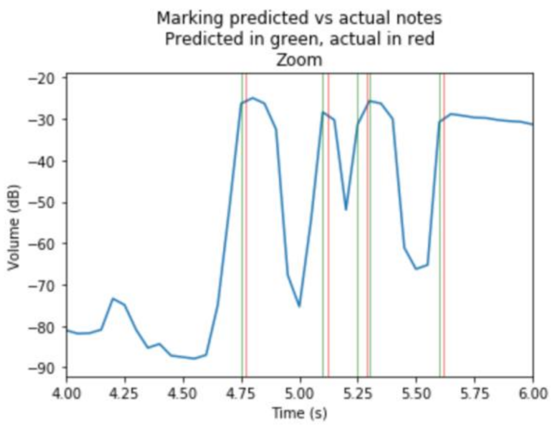


Figure 10: Zooming in on figure 8

To remedy this problem, I wrote a small bit of code that said if any two predicted notes are less than 0.1 seconds apart, take the average of the two notes and use that as a singular note. Three plots (one over the whole melody and two with adjusted x axes) of this second attempt at identifying note changes can be seen in Figures 11, 12, and 13.

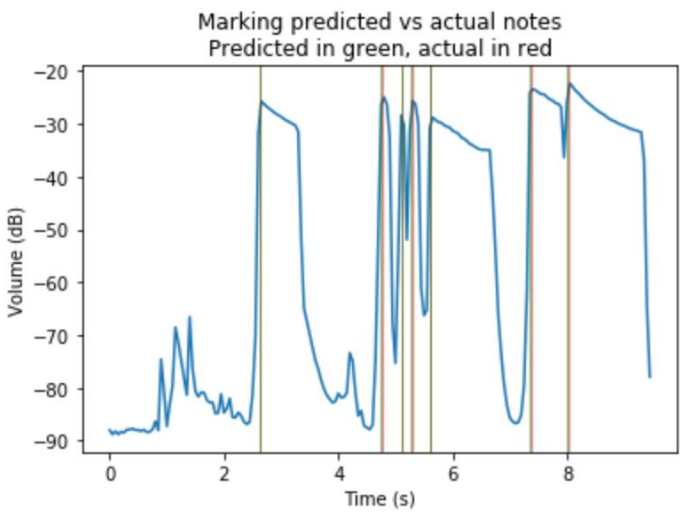


Figure 11: Second attempt at predicting notes

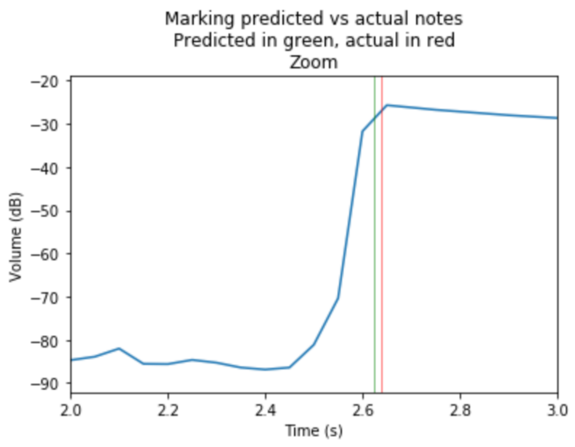


Figure 12: Zooming in on figure 11

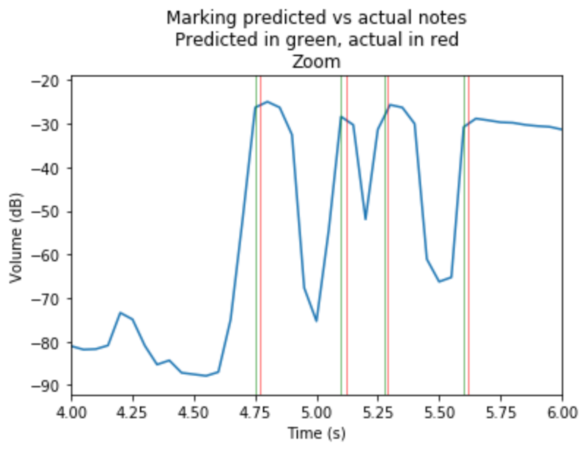
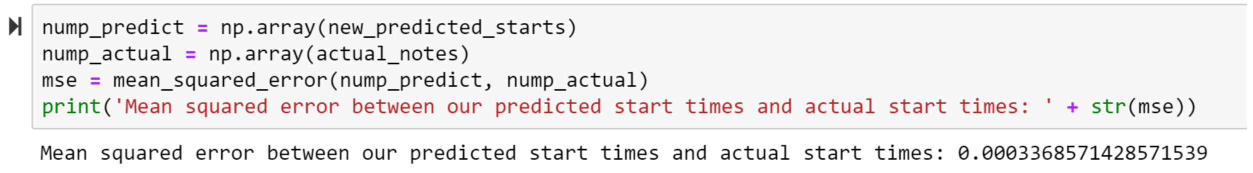


Figure 13: Zooming in on figure 11

As far as the visual test goes, it looks like the predictions are very close to the analytical values. To determine error numerically, I compute the mean squared error between my predicted start times and the actual start times. This can be seen in Figure 14. The result is extremely small even with the small scale of frequencies used. After this step, I was confident in my ability to predict note changes.



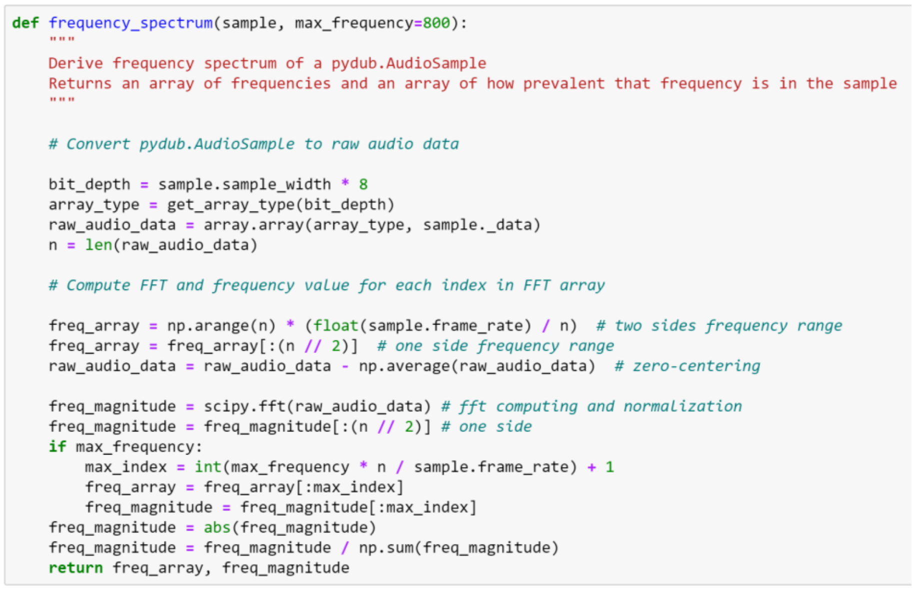
```
nump_predict = np.array(new_predicted_starts)
nump_actual = np.array(actual_notes)
mse = mean_squared_error(nump_predict, nump_actual)
print('Mean squared error between our predicted start times and actual start times: ' + str(mse))
```

```
Mean squared error between our predicted start times and actual start times: 0.0003368571428571539
```

Figure 14: MSE on prediction

3.E. Performing Fourier Transform on Individual Notes

Now comes the computationally challenging part. Now that I know where each note occurs, I need to convert the audio data from existing in the time domain to existing in the frequency domain. Code for this section was put together using multiple sources and can be seen in Figure 15 (Pydub raw audio data) (Audio Frequencies in Python) (VonSeggern). The function I have created, called frequency_spectrum, will first take in a pydub AudioSample and convert it to raw audio data. After I got the raw audio data, I was able to perform a Fast Fourier Transform on the data to convert the time domain to the frequency domain. The function returns two arrays. The first array is the data in the frequency domain and the second array can be used for easier analysis on the prevalence of specific frequencies in the sample. I used this code (Figure 15) on the first predicted note in the melody (Figure 16). As can be seen in the plot, the result is quite clean.



```python
def frequency_spectrum(sample, max_frequency=800):
    """
    Derive frequency spectrum of a pydub.AudioSample
    Returns an array of frequencies and an array of how prevalent that frequency is in the sample
    """

    # Convert pydub.AudioSample to raw audio data

    bit_depth = sample.sample_width * 8
    array_type = get_array_type(bit_depth)
    raw_audio_data = array.array(array_type, sample._data)
    n = len(raw_audio_data)

    # Compute FFT and frequency value for each index in FFT array

    freq_array = np.arange(n) * (float(sample.frame_rate) / n)  # two sides frequency range
    freq_array = freq_array[:(n // 2)]  # one side frequency range
    raw_audio_data = raw_audio_data - np.average(raw_audio_data)  # zero-centering

    freq_magnitude = scipy.fft(raw_audio_data) # fft computing and normalization
    freq_magnitude = freq_magnitude[:(n // 2)] # one side
    if max_frequency:
        max_index = int(max_frequency * n / sample.frame_rate) + 1
        freq_array = freq_array[:max_index]
        freq_magnitude = freq_magnitude[:max_index]
    freq_magnitude = abs(freq_magnitude)
    freq_magnitude = freq_magnitude / np.sum(freq_magnitude)
    return freq_array, freq_magnitude
```

Figure 15: FFT Code

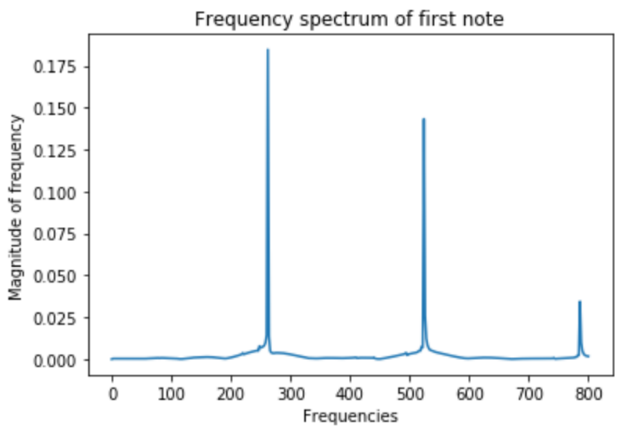
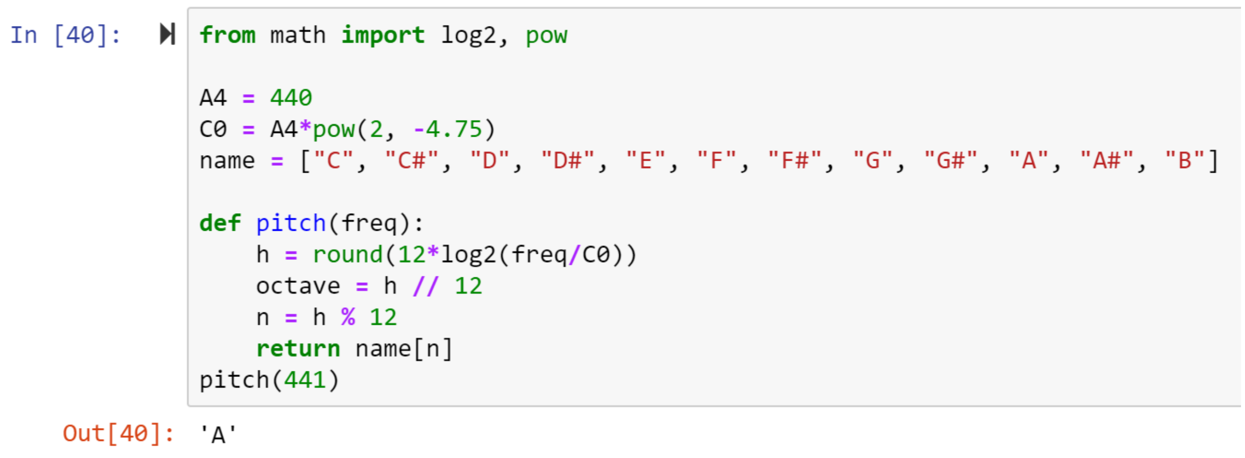


Figure 16: FFT produces frequency spectrum

3.F. Converting Frequencies to Notes

The next step in the process is to convert these frequencies to note names. Every note in Western music can be calculated as a distance from C0. The first letter indicates the name of the note and the second indicates its octave. C0 is four octaves lower than middle C. With the code here, I calculate C0's frequency numerically from A4 (as I already know that A4's frequency is 440 Hz), then calculate how many half-steps my experimental frequency is from C0. Once this is determined, I will know the note name and its octave. The octave, for my purposes, is not very important, so I will only focus on grabbing the note name. Code for this can be seen in Figure 17. The logic for the code used here is taken from a blog written by John D. Cook (Cook).

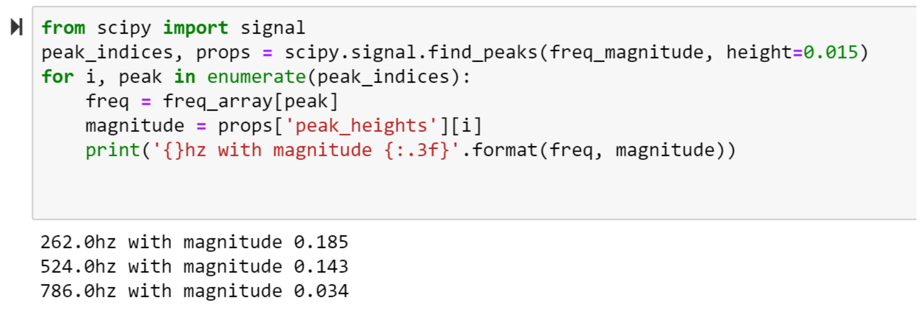


```
In [40]:  from math import log2, pow

          A4 = 440
          C0 = A4*pow(2, -4.75)
          name = ["C", "C#", "D", "D#", "E", "F", "F#", "G", "G#", "A", "A#", "B"]

          def pitch(freq):
              h = round(12*log2(freq/C0))
              octave = h // 12
              n = h % 12
              return name[n]
          pitch(441)

Out[40]:  'A'
```

Figure 17: Converting frequencies to note names

3.G. Predicting Notes

Now that I can predict note changes, Fast Fourier Transform all the notes in an audio file from the time domain to the frequency domain, and convert these frequencies to note names, I should be able to predict the note names of all of the notes in my audio file. I use the scipy signal library to do this. I grab the peaks of the previous frequency graph (Figure 16) using the find_peaks function in the signal library and print out the magnitude of each peak. This is first done on the first note in the melody (Figure 18).



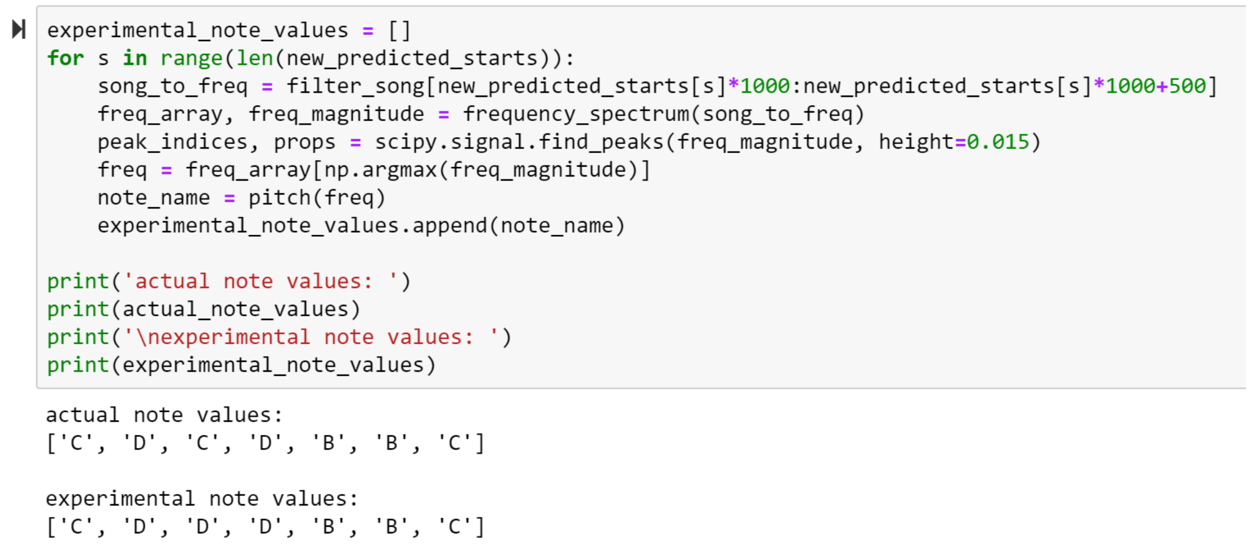
```
from scipy import signal
peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
for i, peak in enumerate(peak_indices):
    freq = freq_array[peak]
    magnitude = props['peak_heights'][i]
    print('{}hz with magnitude {:.3f}'.format(freq, magnitude))
```

```
262.0hz with magnitude 0.185
524.0hz with magnitude 0.143
786.0hz with magnitude 0.034
```

Figure 18: Finding peaks and their magnitudes

These results show that, in the first note, the prominent frequencies, in decreasing order of magnitude, are 262, 524, and 786 Hz. The actual first note played is a C4. C4 has a frequency of 261.63 Hz. This is extremely close to the largest magnitude frequency in the frequency graph (262 Hz). The other two frequencies that pop up correspond to the overtones that the note produces when it is played. It is important to recognize that when even only a single note is played on a guitar, multiple overtones will ring out. The overtones that can be seen here are a C5 (which makes sense as it is another C note) and a G5. The G5 makes sense as G is the perfect fifth of C. This means it is the fifth note in the key of C major and, for lack of a better description, extremely harmonically compatible with the note C. At the moment, it looks like we will be able to simply use the pitch with the largest magnitude peak in our frequency data as the note (Tuning Frequencies for equal-tempered scale, A4 = 440 Hz).

Finally, I want to iterate over all of my notes and see how this method works on predicting all of their names (Figure 19). Every note is predicted correctly, except for one. The only note predicted incorrectly was the third note. The note that is played is a C, but the program is predicting that it should be a D.



```
experimental_note_values = []
for s in range(len(new_predicted_starts)):
    song_to_freq = filter_song[new_predicted_starts[s]*1000:new_predicted_starts[s]*1000+500]
    freq_array, freq_magnitude = frequency_spectrum(song_to_freq)
    peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
    freq = freq_array[np.argmax(freq_magnitude)]
    note_name = pitch(freq)
    experimental_note_values.append(note_name)

print('actual note values: ')
print(actual_note_values)
print('\nexperimental note values: ')
print(experimental_note_values)
```

```
actual note values:
['C', 'D', 'C', 'D', 'B', 'B', 'C']

experimental note values:
['C', 'D', 'D', 'D', 'B', 'B', 'C']
```

Figure 19: Prediction on all notes in the melody

To get a closer look, I plotted the frequency spectrum of just the third note. Figure 20 shows the frequency spectrum plot and Figure 21 shows the magnitudes of each peak. The 260.0 and 264.0 Hz frequencies correspond to C (the correct note), but unfortunately the 294 Hz and 588 Hz frequencies correspond to a D. I believe that the problem here is because the third note is sandwiched in between two notes that are D's. The second and fourth note are D's, and I believe that noise from the second and fourth notes has corrupted the frequency spectrum of the third note. To solve this problem, I attempted to reduce the time period within which I analyze the frequency spectrum. I thought that by tightening the time period, the noise might not corrupt as strongly, but this did not work. I was not able to solve this problem, but I felt confident with my method so far and believed it was time to start analyzing chords. If I ran into this same problem with chord analysis, I would know, then, that something needed to change.

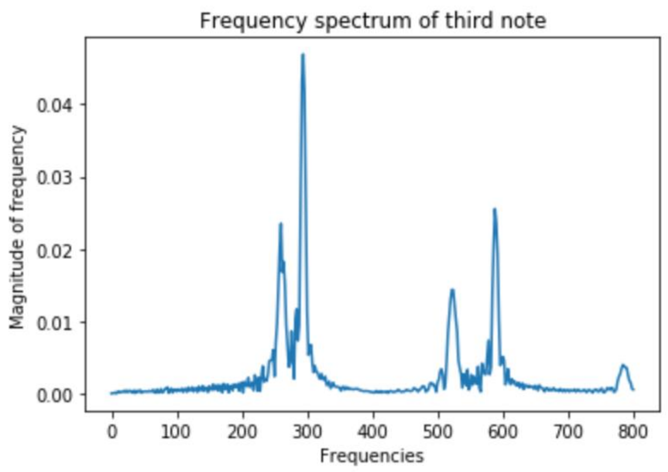
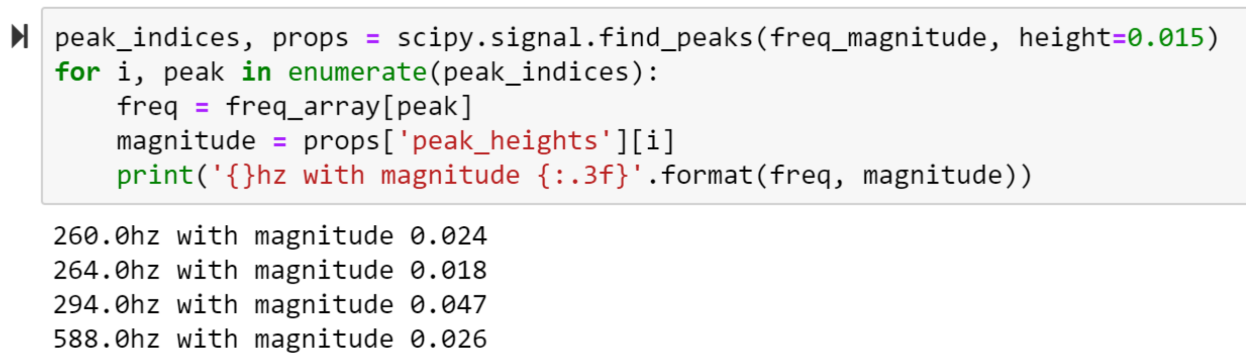


Figure 20: FFT produces frequency spectrum



```
peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
for i, peak in enumerate(peak_indices):
    freq = freq_array[peak]
    magnitude = props['peak_heights'][i]
    print('{}hz with magnitude {:.3f}'.format(freq, magnitude))
```

```
260.0hz with magnitude 0.024
264.0hz with magnitude 0.018
294.0hz with magnitude 0.047
588.0hz with magnitude 0.026
```

Figure 21: Peak analysis

3.H. Conclusions

Despite the presence of overtones that occur in any singularly played note in a melody, the primary frequency that the Fourier Fast Transform produces on an audio file is, almost always, the base note being played. To determine if this simplistic method works on chords, I will need to record a new audio file of chordal data and see how my method works on that.

4. Chord Analysis

4.A. Input

To analyze chords, the process is nearly the exact same as it was for notes. Instead of a melody of single notes, I record myself playing a chord progression. I, again, pull in a plot for volume over time for the chord progression as can be seen in Figure 22. To remove extra noise, the same high-pass-filter from the note analysis is applied (Figure 23). I am omitting much of the code from this section as it is the exact same code as was used for note analysis. However, I did need to modify the threshold functions for determining when chord changes were occurring. The volume of the chord recording, overall, was much louder than the volume of the melody. Due to this, I set the volume threshold for a new chord at -20 dB and the edge threshold at 3 dB (Figure 24). When setting these thresholds and including my code that does not let chord changes occur too closely together, we can see, visually, that the chord changes are predicted correctly (Figure 25).

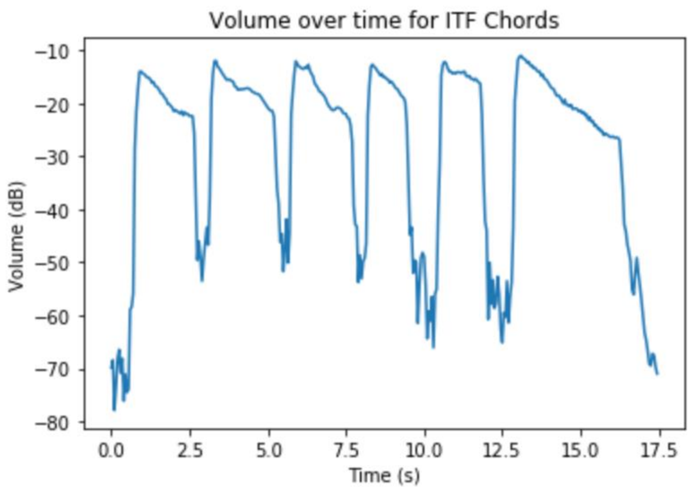


Figure 22: Pulling in chord audio file

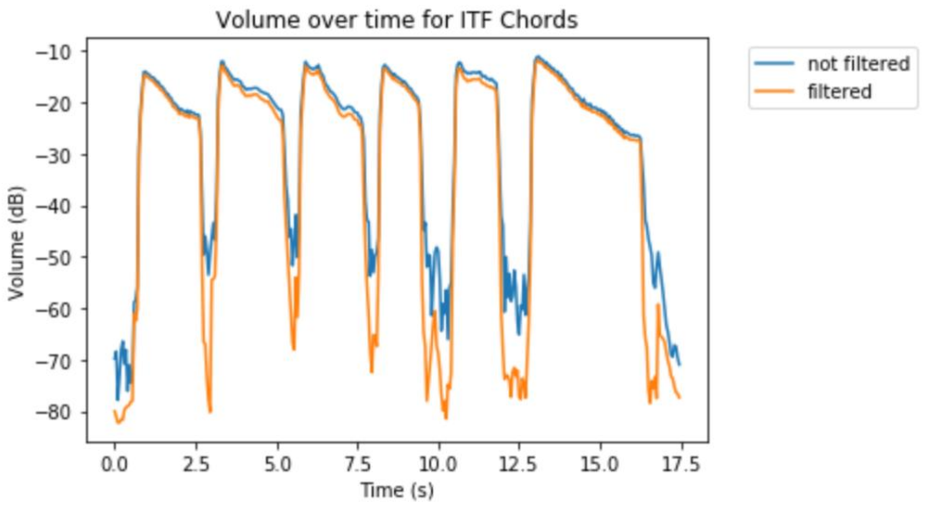
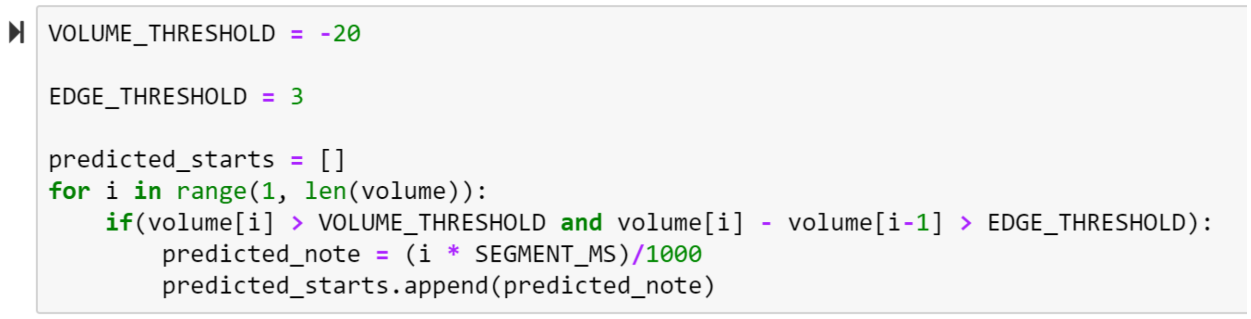


Figure 23: Filtering chord audio file



```
VOLUME_THRESHOLD = -20

EDGE_THRESHOLD = 3

predicted_starts = []
for i in range(1, len(volume)):
    if(volume[i] > VOLUME_THRESHOLD and volume[i] - volume[i-1] > EDGE_THRESHOLD):
        predicted_note = (i * SEGMENT_MS)/1000
        predicted_starts.append(predicted_note)
```

Figure 24: Adjusting thresholds

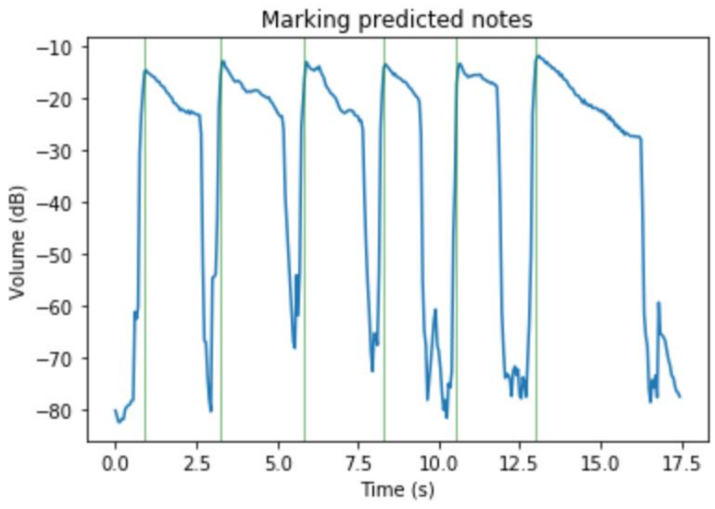


Figure 25: Predicting chord changes

4.B. Performing Fourier Transform on Individual Chords

The same Fast Fourier Transform algorithm as was applied to the note analysis is used for chord analysis here, so I will omit the code in this section to avoid redundancy. Figure 26 represents a frequency spectrum of the first chord in the progression, a C chord.

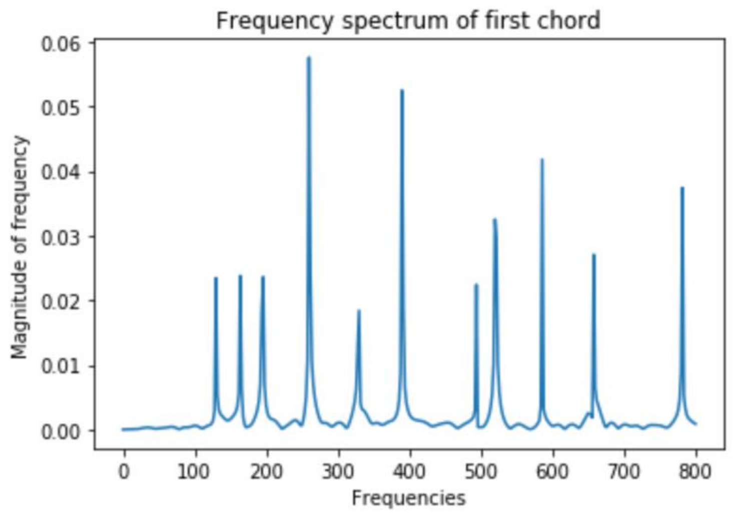
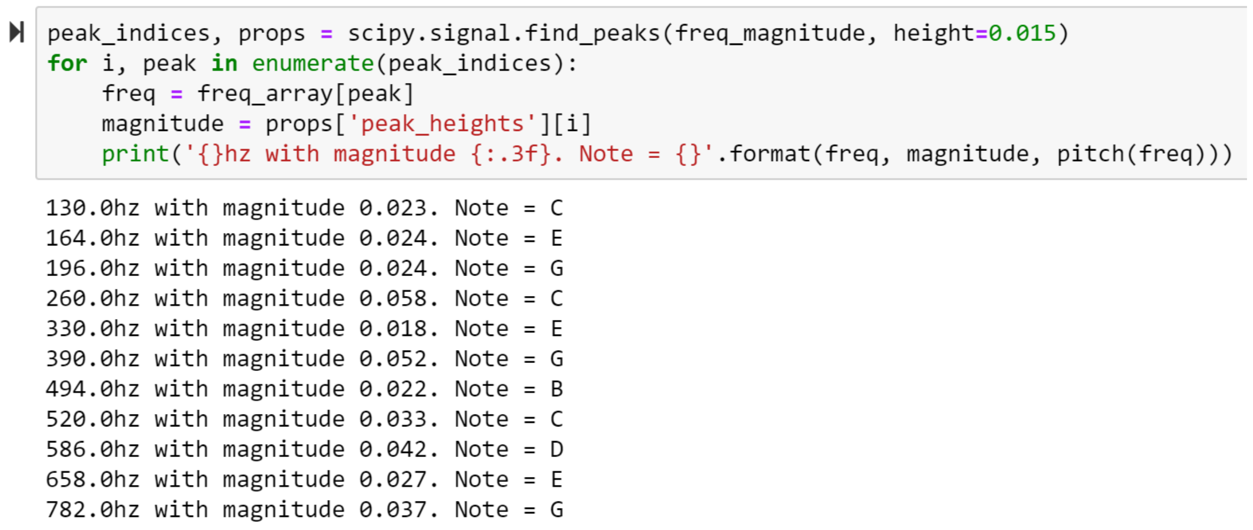


Figure 26: FFT frequency spectrum on a C chord

The overall spectrum retains resemblance to the note spectrum in the sense that the peaks are quite well defined, but there are clearly a lot more peaks to analyze than in the note spectrum. Code and output of peak magnitude can be seen in Figure 27.

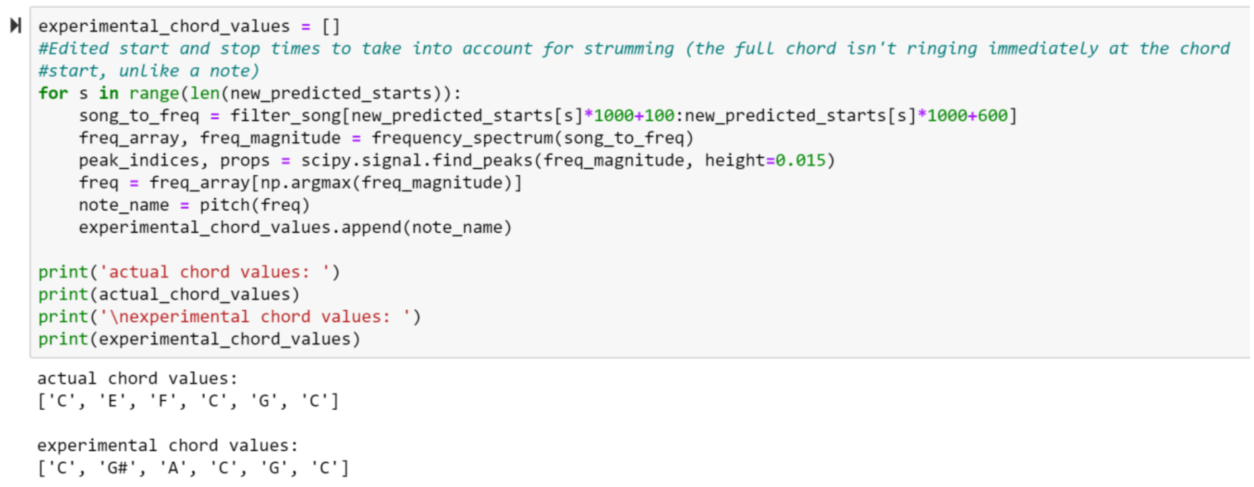


```
peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
for i, peak in enumerate(peak_indices):
    freq = freq_array[peak]
    magnitude = props['peak_heights'][i]
    print('{}hz with magnitude {:.3f}. Note = {}'.format(freq, magnitude, pitch(freq)))
```

```
130.0hz with magnitude 0.023. Note = C
164.0hz with magnitude 0.024. Note = E
196.0hz with magnitude 0.024. Note = G
260.0hz with magnitude 0.058. Note = C
330.0hz with magnitude 0.018. Note = E
390.0hz with magnitude 0.052. Note = G
494.0hz with magnitude 0.022. Note = B
520.0hz with magnitude 0.033. Note = C
586.0hz with magnitude 0.042. Note = D
658.0hz with magnitude 0.027. Note = E
782.0hz with magnitude 0.037. Note = G
```

Figure 27: Peak analysis of the first C chord

The most prominent note occurs at frequency of 260 Hz with magnitude of 0.058. This is a C. So far, it looks like the same method used previously will work. However, when running on all of the chords in the progression, we incorrectly predict two chords (Figure 28). Something interesting to note is that the incorrectly predicted chords are G# and A when they should be E and F. G# is the major third of E and A is the major third of F. In a harmony, the note a third up from the tonic is what determines whether the chord is major or minor. Although I am not delving into predictions of tonality (I am only concerned with the tonal center of any chord in my analysis), this could be useful in the future to predict the tonality of a chord. In the next section, I analyze the incorrect chord predictions to see where the program went wrong and how to fix the problem.



```
experimental_chord_values = []
#Edited start and stop times to take into account for strumming (the full chord isn't ringing immediately at the chord
#start, unlike a note)
for s in range(len(new_predicted_starts)):
    song_to_freq = filter_song[new_predicted_starts[s]*1000+100:new_predicted_starts[s]*1000+600]
    freq_array, freq_magnitude = frequency_spectrum(song_to_freq)
    peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
    freq = freq_array[np.argmax(freq_magnitude)]
    note_name = pitch(freq)
    experimental_chord_values.append(note_name)

print('actual chord values: ')
print(actual_chord_values)
print('\nexperimental chord values: ')
print(experimental_chord_values)
```

```
actual chord values:
['C', 'E', 'F', 'C', 'G', 'C']

experimental chord values:
['C', 'G#', 'A', 'C', 'G', 'C']
```

Figure 28: Predicting all chords

4.C. Error Analysis and Problem Solving

To see where everything went wrong, I first analyze the frequency spectrum of the second chord in the progression. The chord played is an E, but the program predicts that it should be a G# (the major third of the E chord). The plot of this frequency spectrum can be seen in Figure 29 and analysis of the peak frequencies and magnitudes in Figure 30.

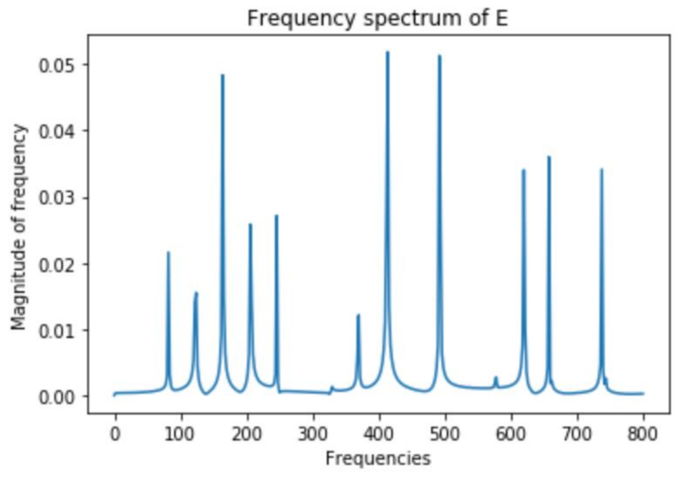
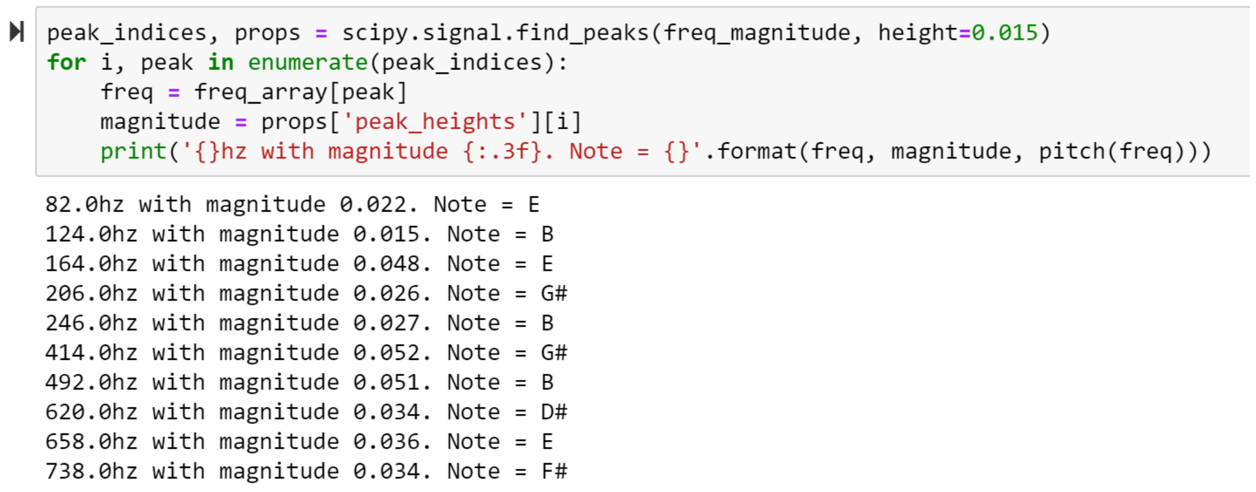


Figure 29: FFT on the third chord, an E

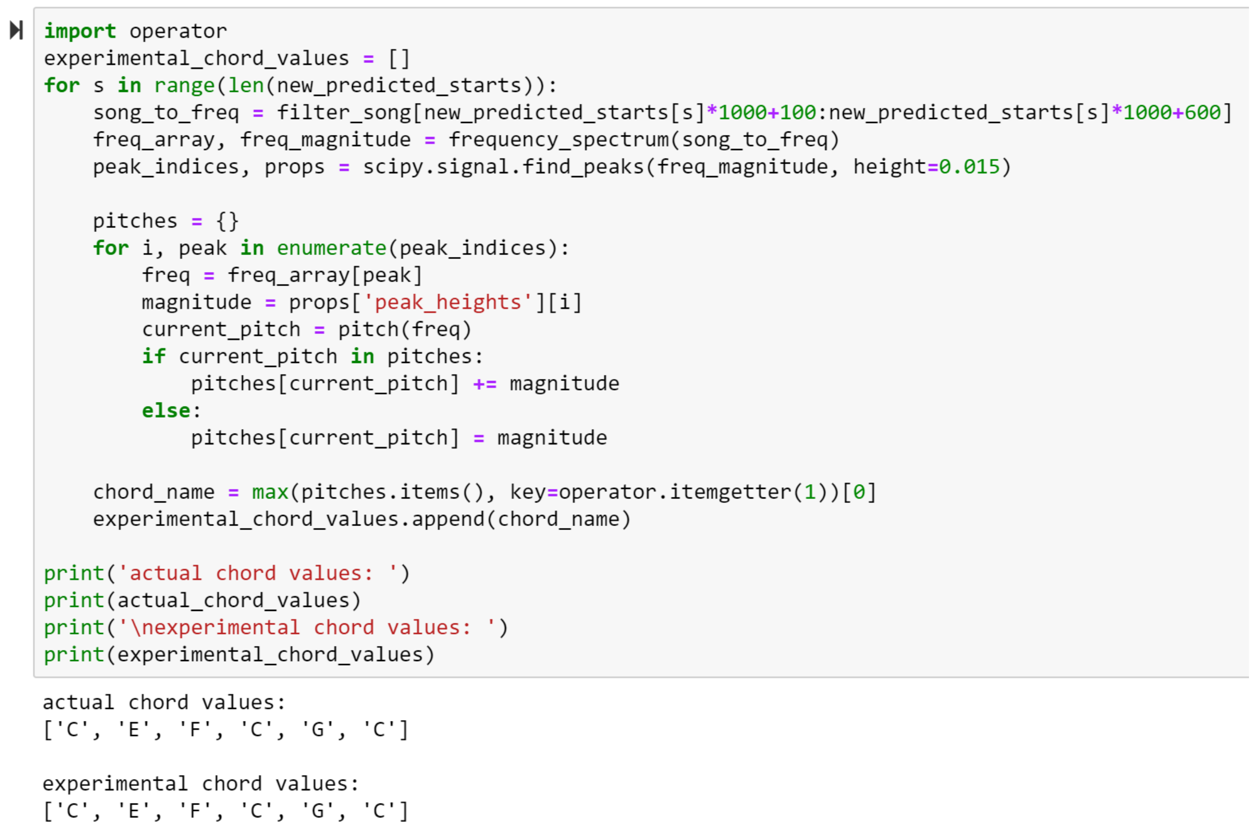


```python
peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)
for i, peak in enumerate(peak_indices):
    freq = freq_array[peak]
    magnitude = props['peak_heights'][i]
    print('{}hz with magnitude {:.3f}. Note = {}'.format(freq, magnitude, pitch(freq)))
```

```
82.0hz with magnitude 0.022. Note = E
124.0hz with magnitude 0.015. Note = B
164.0hz with magnitude 0.048. Note = E
206.0hz with magnitude 0.026. Note = G#
246.0hz with magnitude 0.027. Note = B
414.0hz with magnitude 0.052. Note = G#
492.0hz with magnitude 0.051. Note = B
620.0hz with magnitude 0.034. Note = D#
658.0hz with magnitude 0.036. Note = E
738.0hz with magnitude 0.034. Note = F#
```

Figure 30: Peak analysis of the E chord

The frequency with the largest magnitude, 414 Hz, is a G# which is why the program thinks that this chord is a G#. Concerningly, the second highest magnitude is a B (the perfect fifth in an E chord), which is also not an E. The third highest magnitude frequency is an E, but it is clear now that simply grabbing the peak with the largest magnitude is not an effective way to identify chords, even though this method worked moderately well with notes. Inspired again by VonSeggern, instead of taking the frequency with the highest magnitude, I implemented code that would get the total magnitude of each pitch in the frequency spectrum (VonSeggern). This can be seen in Figure 31. For example, in the above analysis of peak frequencies and magnitudes, Figure 30, frequencies indicating an E note occurred with magnitudes 0.022, 0.048, and 0.036. Summing all of these will give us our new, total magnitude. When using this instead of my previous method, all chords are now predicted correctly (Figure 31)!



```python
import operator
experimental_chord_values = []
for s in range(len(new_predicted_starts)):
    song_to_freq = filter_song[new_predicted_starts[s]*1000+100:new_predicted_starts[s]*1000+600]
    freq_array, freq_magnitude = frequency_spectrum(song_to_freq)
    peak_indices, props = scipy.signal.find_peaks(freq_magnitude, height=0.015)

    pitches = {}
    for i, peak in enumerate(peak_indices):
        freq = freq_array[peak]
        magnitude = props['peak_heights'][i]
        current_pitch = pitch(freq)
        if current_pitch in pitches:
            pitches[current_pitch] += magnitude
        else:
            pitches[current_pitch] = magnitude

    chord_name = max(pitches.items(), key=operator.itemgetter(1))[0]
    experimental_chord_values.append(chord_name)

print('actual chord values: ')
print(actual_chord_values)
print('\nexperimental chord values: ')
print(experimental_chord_values)
```

```
actual chord values:
['C', 'E', 'F', 'C', 'G', 'C']

experimental chord values:
['C', 'E', 'F', 'C', 'G', 'C']
```

Figure 31: Correctly predicting chords with new summation method

5. Conclusion

Before beginning this project, I had done quite a bit of research on the topic at hand. Countless sources were telling me how complex chord identification was and that no simple Fourier Transform would be able to convey easily interpretable information from a raw audio file. My results prove that this is not entirely accurate. It would be interesting to see how effective my program is at predicting more challenging chords. The chord progression I chose only used major chords. Therefore, I am not sure how well it would be able to predict on minor chords, 7th chords, 9th chords, suspended chords, and other, less common, chord types. However, it felt quite promising that I was able to create a program that could read in an audio file, determine where chord changes occurred, and spit out an accurate (at least as far as my testing went) prediction of what the chord name.

As I mentioned in the introduction section, my original goal was to create polyphonic autotune. This is just one step towards that goal. If I find time in the future to continue working on this project, the next step is to isolate the individual notes being played from polyphonic sound. Standard autotune libraries can then be used on the individual notes. Once the individual notes are pitch-corrected, the note frequencies can be combined and the Fast Fourier Transform can be used, once again, to convert the frequency domain back to the time domain. This will give the user the proper wave function for the audio. I am sure there will be numerous issues that crop up in that process, but it is encouraging that this first foundational step is working effectively. The entire code can be found at https://github.com/mcatalano26/FourierChordAnalysis.

References

"Audio Frequencies in Python." Forum Post. 2019. <https://stackoverflow.com/questions/53308674/audio-frequencies-in-python>.

Cook, John D. "Musical pitch notation." Blog Post. 2016. <https://www.johndcook.com/blog/2016/02/10/musical-pitch-notation/>.

"Guitar Chord Recognition Algorithm?" Forum Post. 2010. <https://stackoverflow.com/questions/4033083/guitar-chord-recognition-algorithm>.

Hamilton, Joe. "simpleaudio." Python Library. 2019. <https://pypi.org/project/simpleaudio/>.

Mihira, Yuma. "PyChord." Python Library. 2021. <https://pypi.org/project/pychord/>.

"Note and Chord Analysis in Python?" Reddit Post. 2015. <https://www.reddit.com/r/learnpython/comments/2gvp8l/note_and_chord_analysis_in_python/>.

"Pydub raw audio data." Forum Post. 2015. <https://stackoverflow.com/questions/53308674/audio-frequencies-in-python>.

Robert, James. "Pydub." Github Repository. 2021. <https://github.com/jiaaro/pydub>.

"Using Fast Fourier Transform to determine musical notes." Forum Post. 2019. <https://dsp.stackexchange.com/questions/54072/using-fast-fourier-transform-to-determine-musical-notes>.

VonSeggern, Ian. "Note Recognition In Python." Blog Post. 2020. <https://medium.com/@ianvonseggern/note-recognition-in-python-c2020d0dae24>.