Converting Sound to Sheet Music

Name: Kian Mohseni

UID: 505084983

TA: Tristan O’Neill

Lab Partners: Sydney Walsh and Jeremy Lim

Lab Section: Lab 12 Thursday 5:00 PM

Date: December 6, 2019

1. Introduction/Theoretical Background

The objective of this project is to create a computer program that outputs parameters necessary for constructing sheet music (timing, notes, and note duration) from inputted audio data collected by the Arduino microphone.

In order to achieve this, we needed to be able to isolate the different frequencies from our audio and the times when these frequencies are present. Thus, we had to perform a Fourier Transform on the sound waveform in order to obtain the resonant frequencies. Considering these objectives, the main quantities we measured were the data collected from the sound waveform and the Fourier Transform of this waveform.

We chose this project because it allowed us to expand on the Fourier analysis of musical instruments in Lab 4 and optimize our ability to classify frequencies with musical notes. Moreover, we think this project is creative in its ability to apply the knowledge we developed in Lab 4 to create a useful product. Translating audio to sheet music is extremely relevant for music applications.

The main parts of the experiment that I contributed to were the recording of the musical notes as well as the data analysis involved using Python.

2. Methods/Experimental Setup

We performed our experiment by first collecting data from several piano recordings at an optimal BPM to coincide with our sampling rate. Realizing our limits in time, we decided to only record notes in sequence rather than simultaneously.

We began our python analysis by properly truncating the inputted data set to isolate the duration in which the notes were played. We then performed the Fourier Transform to obtain our spectrogram data. Through obtaining more data and attempting different techniques to interpret our collected data, we developed appropriate thresholds for determining notes and approximately what time these notes were played. To increase the accuracy of our note mappings, we separated our data into strategic time chunks and then performing fourier analysis on these chunks individually to determine more accurate frequency values. We continued this process for multiple chunks until we reach an optimal mapping. Moreover, we mitigated issues involving multiple harmonics of notes and overtones appearing in our Fourier Transform by predicting their occurrence and developing thresholds to ignore these values as potential noise.

Using Fourier Transforms, we converted our sound waves, which are a function of time, into graphs that show us the peak frequencies of each sound file and the intensity of each peak, allowing us to determine which notes were played. This then created patterns of equidistant harmonics with decreasing thresholds showing up on the Fourier Transforms for each note. This way determined which frequencies were the resonant frequencies we needed to use to translate into the notes that were played and the times they were played at. We also know that for every octave, the frequency of the note increases by a factor of 2.

We quantified our results by comparing the actual parameter values (frequency, note duration, timing) of our data with our software-predicted values. From these different data parameters, we isolated each variable and determined their individual percent errors by dividing the software-determined estimation by the actual values for our data and dividing this quantity by the actual values.

The materials we needed were:

- Arduino
- Microphone
- Piano/Synthesizer

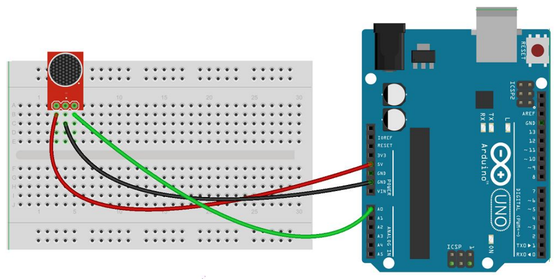


Figure 1. This conceptual drawing shows the wiring of our microphone and Arduino, which is the same as the one done in Lab 4.

3. Results/Discussion

We expected our results to mostly match our software predictions. However, due to the potential errors, we did predict a certain level of error. For example, one source of error was the presence of background noise. In addition, there are several limitations to software thresholding as well as to the hardware we used. The data we got using a professional microphone was much cleaner than the data we got using the Arduino microphone. Moreover, inaccurately determining the sampling rate of our Arduino could have caused inaccuracies in the frequencies the Fourier Transform mapped to.

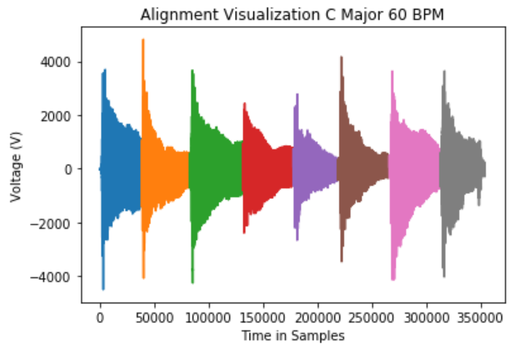


Figure 2. This graph shows the alignment data for both the Arduino Microphone and the Professional Microphone when playing C Major.

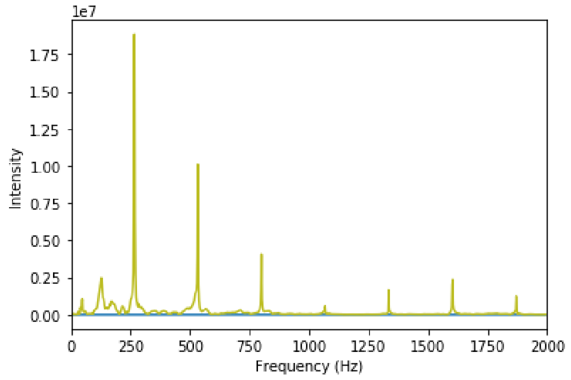


Figure 3. C4 Professional Microphone FFT.

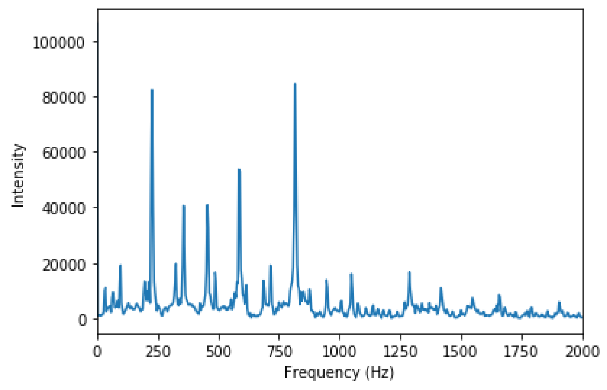


Figure 4. C4 Arduino FFT.

The theoretical frequency of C4 is about 261 Hz so the Professional Microphone data translated perfectly as seen in Figure 3, while the Arduino Microphone data was extremely inaccurate despite accurate alignment. We suspect this to be due to limitations of the Arduino Microphone to handle the high amplitude strikes of the piano which can cause saturation and distortion.

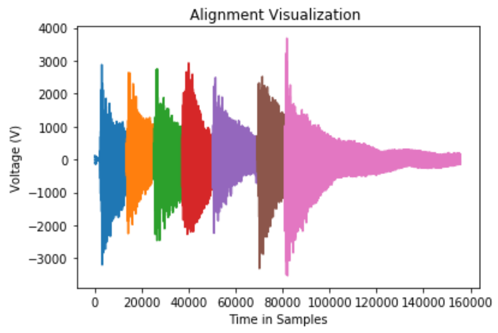


Figure 5. This graph shows the alignment data for both the Arduino Microphone and the Professional Microphone when playing “The Lick” at 120 BPM.

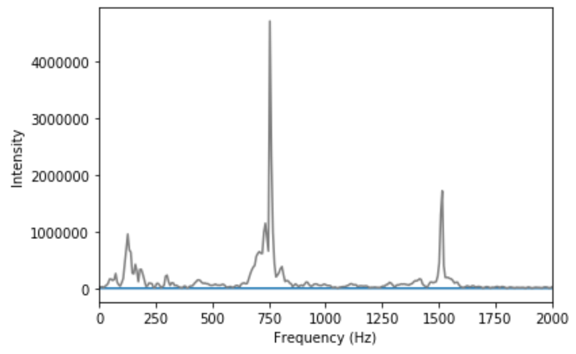


Figure 6. F#4 Professional Microphone FFT.

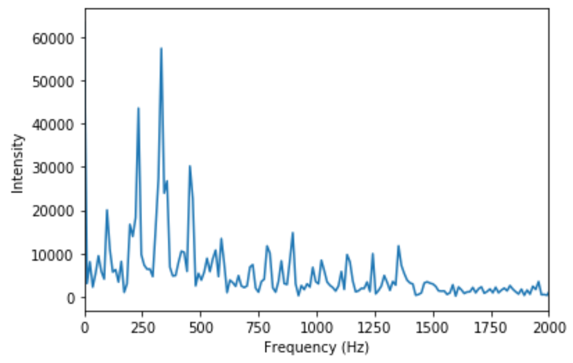


Figure 7. F#4 Arduino FFT.

The theoretical frequency of F#4 is about 784 Hz so the Professional Microphone data maps the notes and note durations perfectly again for “The Lick”, while the Arduino Microphone doesn’t as shown in Figures 5, 6, and 7.

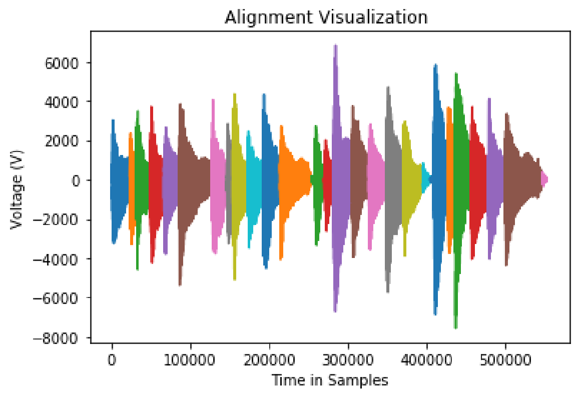


Figure 8. This graph shows the alignment data for both the Arduino Microphone and the Professional Microphone when playing "Happy Birthday" at 120 BPM.

We can see that accurate alignment becomes very difficult with increasing BPM, varied rhythm, and increased length of the dataset. The Professional Microphone was completely accurate except for three instances of incorrect partitioning, where a note not present was identified for A4, A#2, and A2. The Arduino Microphone data was again very inaccurate.

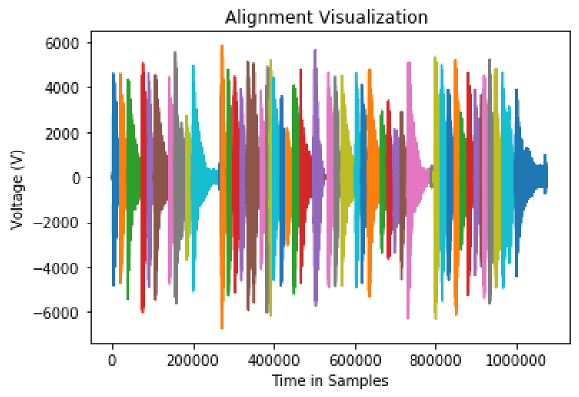


Figure 9. This graph shows the alignment data for both the Arduino Microphone and the Professional Microphone when playing "Jingle Bells" at 160 BPM.

The Professional Microphone data was accurate except for the omission of C4 and D4 and the addition of B1. Meanwhile the beginning frequencies for the Arduino Microphone were somewhat close but very different from those of the Professional Microphone, despite similar alignment.

We identified several possible sources of error in our results. The first of which had to do with determining intensity thresholds through experimental testing. This can be a source of error if they are too sensitive or not sensitive enough and was clearly noticeable in the Arduino Microphone when playing "Jingle Bells". Also, with faster rhythms and higher BPM's there is a conflict between partitioning for short and long duration notes. If partitions are too short, this can cause the recognition of some noise as a note unintentionally as seen in the alignment in Figure 9.

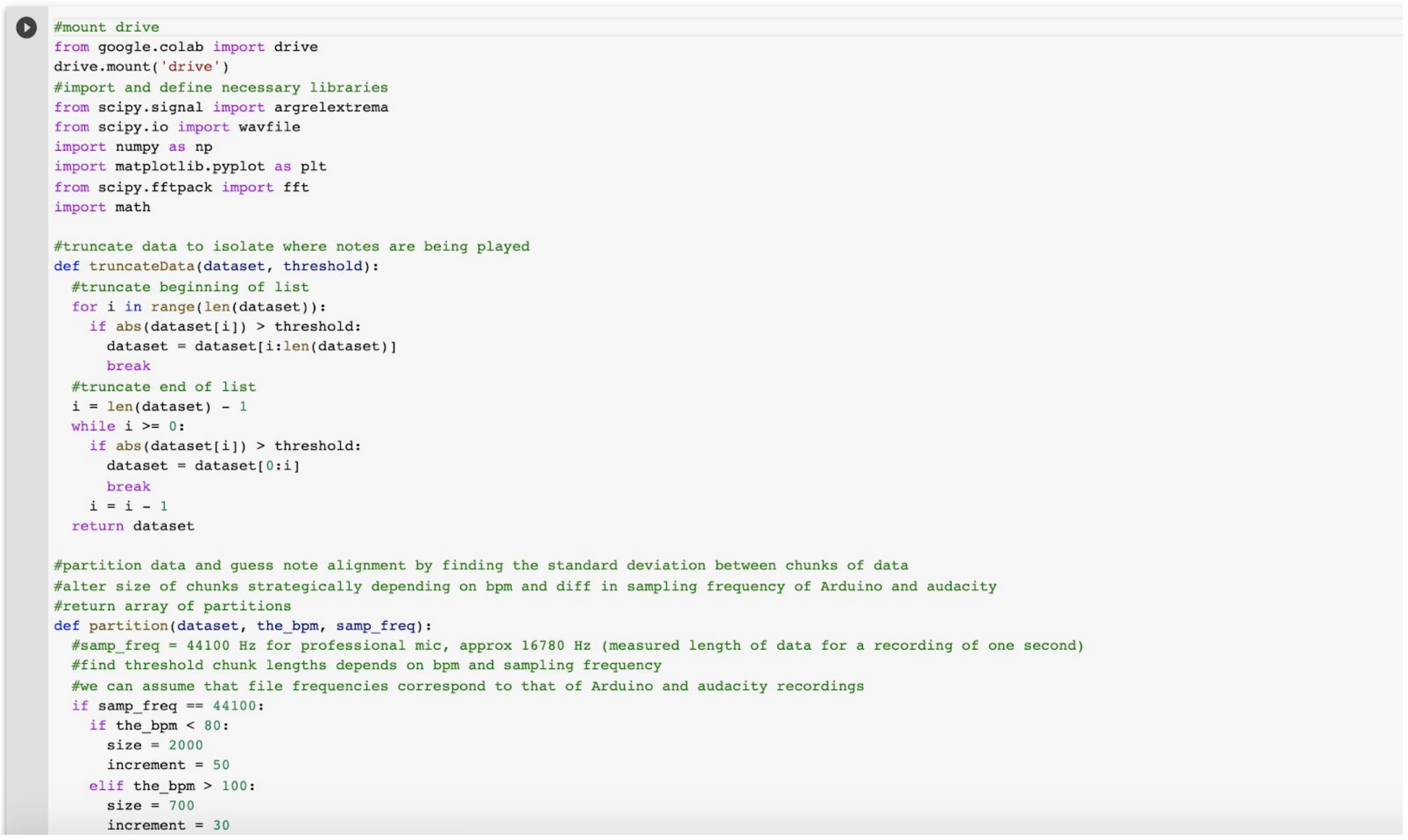
Overall, we just needed better equipment like a Professional Microphone for cleaner and more accurate results than we saw with the Arduino Microphone.

4. Conclusion

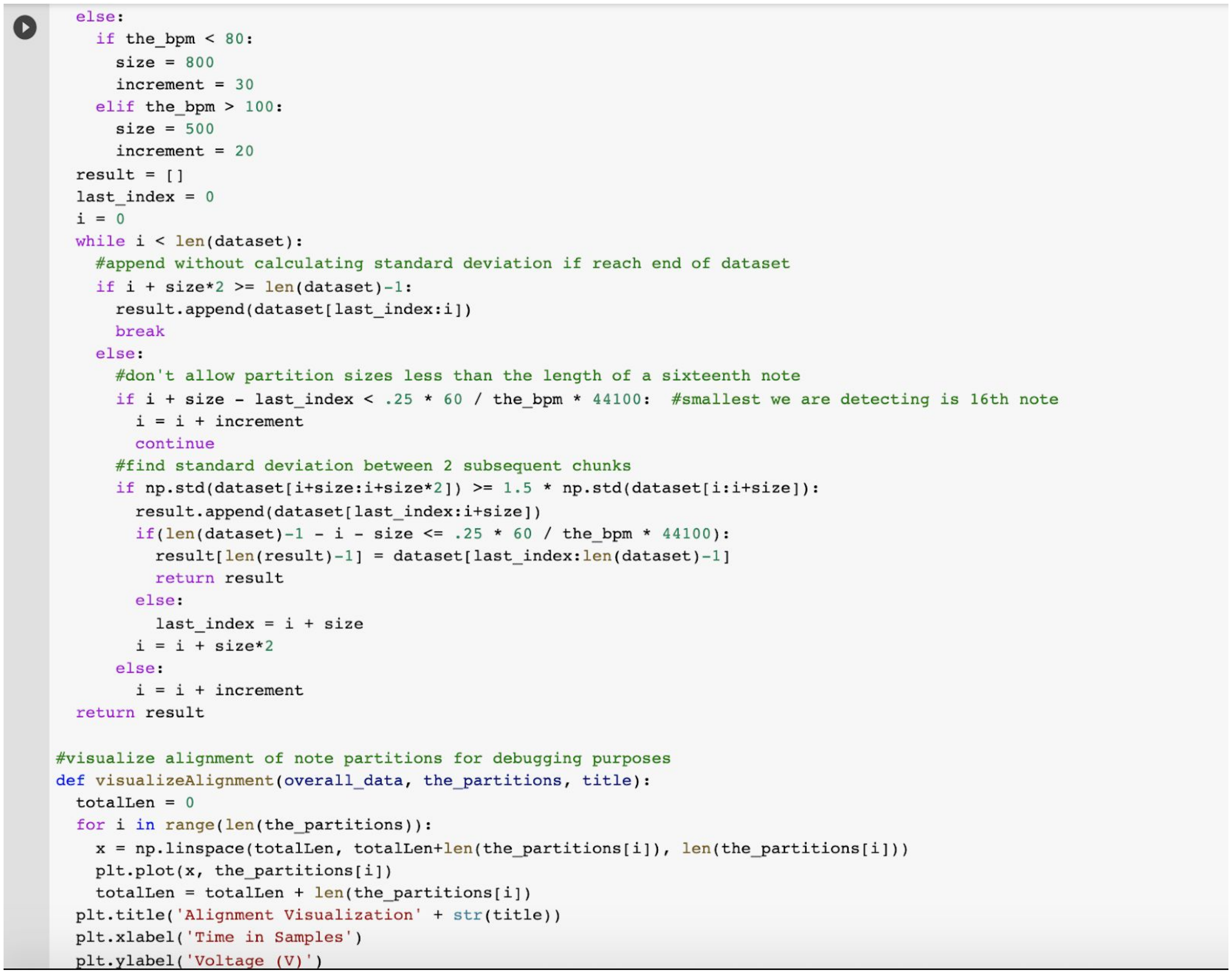
In conclusion, we obtained successful results with our professional microphone for the most part. However our results indicate that future optimization of our note alignment algorithm should be made to account for short and long notes. Meanwhile the results we obtained with our Arduino microphone were very poor most likely due to the saturation of audio data. This can be observed by comparing the Fourier Transform plots for each microphone while playing the same notes, as one showed a much cleaner signal.

Overall, the physics portion of our experiment was successfully accomplished because we collected all the necessary data to write sheet music, however the coding could still use some polishing up. In the future we could expand on what we already have in order to recognize multiple simultaneous notes with varied rhythm and rests. We could figure out how to optimize our thresholding techniques using machine learning to test our future program on other instruments like the guitar.

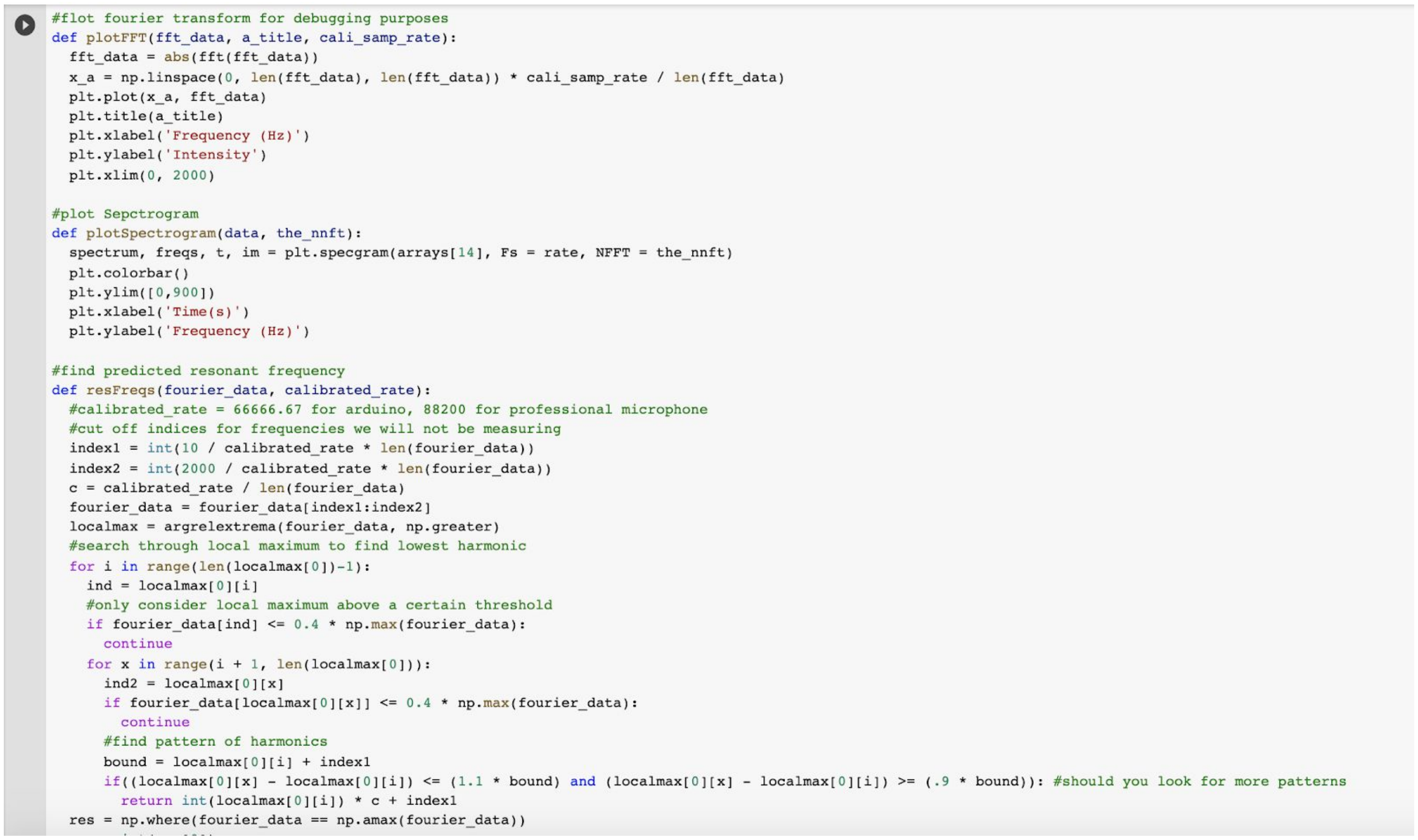
Appendix (Code)



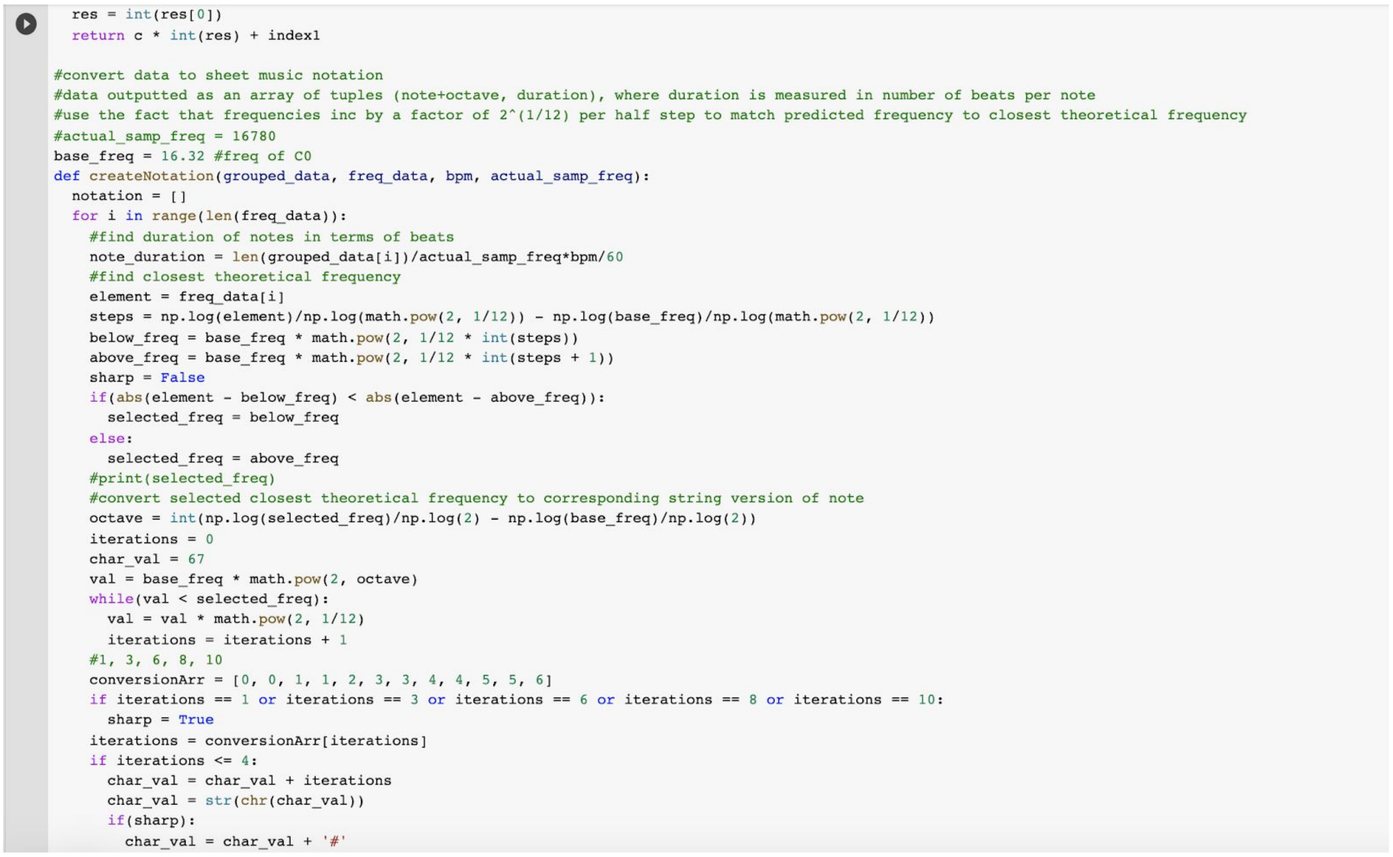
```
#mount drive
from google.colab import drive
drive.mount('drive')
#import and define necessary libraries
from scipy.signal import argrelextrema
from scipy.io import wavfile
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft
import math

#truncate data to isolate where notes are being played
def truncateData(dataset, threshold):
  #truncate beginning of list
  for i in range(len(dataset)):
    if abs(dataset[i]) > threshold:
      dataset = dataset[i:len(dataset)]
      break
  #truncate end of list
  i = len(dataset) - 1
  while i >= 0:
    if abs(dataset[i]) > threshold:
      dataset = dataset[0:i]
      break
    i = i - 1
  return dataset

#partition data and guess note alignment by finding the standard deviation between chunks of data
#alter size of chunks strategically depending on bpm and diff in sampling frequency of Arduino and audacity
#return array of partitions
def partition(dataset, the_bpm, samp_freq):
  #samp_freq = 44100 Hz for professional mic, approx 16780 Hz (measured length of data for a recording of one second)
  #find threshold chunk lengths depends on bpm and sampling frequency
  #we can assume that file frequencies correspond to that of Arduino and audacity recordings
  if samp_freq == 44100:
    if the_bpm < 80:
      size = 2000
      increment = 50
    elif the_bpm > 100:
      size = 700
      increment = 30
```



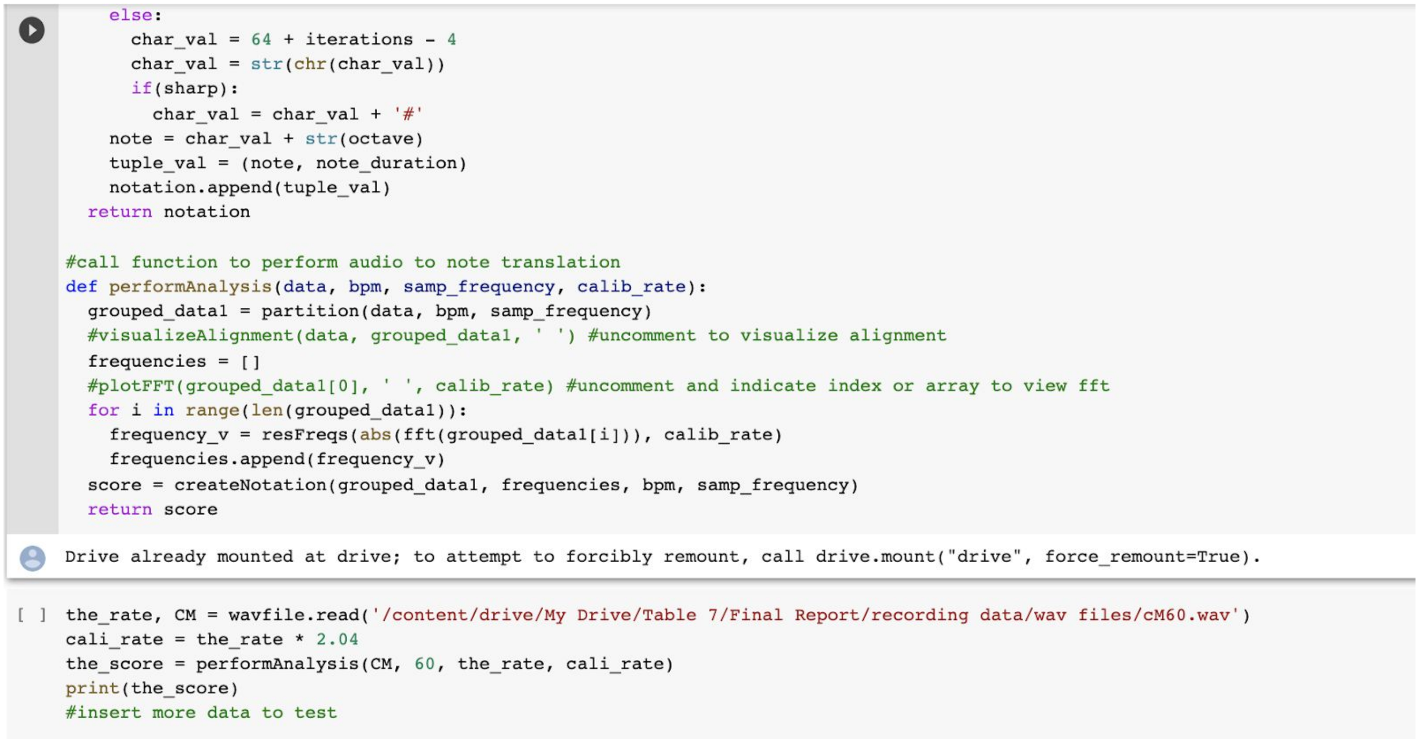
```
  else:
    if the_bpm < 80:
      size = 800
      increment = 30
    elif the_bpm > 100:
      size = 500
      increment = 20
  result = []
  last_index = 0
  i = 0
  while i < len(dataset):
    #append without calculating standard deviation if reach end of dataset
    if i + size*2 >= len(dataset)-1:
      result.append(dataset[last_index:i])
      break
    else:
      #don't allow partition sizes less than the length of a sixteenth note
      if i + size - last_index < .25 * 60 / the_bpm * 44100:  #smallest we are detecting is 16th note
        i = i + increment
        continue
      #find standard deviation between 2 subsequent chunks
      if np.std(dataset[i+size:i+size*2]) >= 1.5 * np.std(dataset[i:i+size]):
        result.append(dataset[last_index:i+size])
        if(len(dataset)-1 - i - size <= .25 * 60 / the_bpm * 44100):
          result[len(result)-1] = dataset[last_index:len(dataset)-1]
          return result
        else:
          last_index = i + size
        i = i + size*2
      else:
        i = i + increment
  return result

#visualize alignment of note partitions for debugging purposes
def visualizeAlignment(overall_data, the_partitions, title):
  totalLen = 0
  for i in range(len(the_partitions)):
    x = np.linspace(totalLen, totalLen+len(the_partitions[i]), len(the_partitions[i]))
    plt.plot(x, the_partitions[i])
    totalLen = totalLen + len(the_partitions[i])
  plt.title('Alignment Visualization' + str(title))
  plt.xlabel('Time in Samples')
  plt.ylabel('Voltage (V)')
```



```
#flot fourier transform for debugging purposes
def plotFFT(fft_data, a_title, cali_samp_rate):
  fft_data = abs(fft(fft_data))
  x_a = np.linspace(0, len(fft_data), len(fft_data)) * cali_samp_rate / len(fft_data)
  plt.plot(x_a, fft_data)
  plt.title(a_title)
  plt.xlabel('Frequency (Hz)')
  plt.ylabel('Intensity')
  plt.xlim(0, 2000)

#plot Sepctrogram
def plotSpectrogram(data, the_nnft):
  spectrum, freqs, t, im = plt.specgram(arrays[14], Fs = rate, NFFT = the_nnft)
  plt.colorbar()
  plt.ylim([0,900])
  plt.xlabel('Time(s)')
  plt.ylabel('Frequency (Hz)')

#find predicted resonant frequency
def resFreqs(fourier_data, calibrated_rate):
  #calibrated_rate = 66666.67 for arduino, 88200 for professional microphone
  #cut off indices for frequencies we will not be measuring
  index1 = int(10 / calibrated_rate * len(fourier_data))
  index2 = int(2000 / calibrated_rate * len(fourier_data))
  c = calibrated_rate / len(fourier_data)
  fourier_data = fourier_data[index1:index2]
  localmax = argrelextrema(fourier_data, np.greater)
  #search through local maximum to find lowest harmonic
  for i in range(len(localmax[0])-1):
    ind = localmax[0][i]
    #only consider local maximum above a certain threshold
    if fourier_data[ind] <= 0.4 * np.max(fourier_data):
      continue
    for x in range(i + 1, len(localmax[0])):
      ind2 = localmax[0][x]
      if fourier_data[localmax[0][x]] <= 0.4 * np.max(fourier_data):
        continue
      #find pattern of harmonics
      bound = localmax[0][i] + index1
      if((localmax[0][x] - localmax[0][i]) <= (1.1 * bound) and (localmax[0][x] - localmax[0][i]) >= (.9 * bound)): #should you look for more patterns
        return int(localmax[0][i]) * c + index1
  res = np.where(fourier_data == np.amax(fourier_data))
```



```
    res = int(res[0])
    return c * int(res) + index1

#convert data to sheet music notation
#data outputted as an array of tuples (note+octave, duration), where duration is measured in number of beats per note
#use the fact that frequencies inc by a factor of 2^(1/12) per half step to match predicted frequency to closest theoretical frequency
#actual_samp_freq = 16780
base_freq = 16.32 #freq of C0
def createNotation(grouped_data, freq_data, bpm, actual_samp_freq):
  notation = []
  for i in range(len(freq_data)):
    #find duration of notes in terms of beats
    note_duration = len(grouped_data[i])/actual_samp_freq*bpm/60
    #find closest theoretical frequency
    element = freq_data[i]
    steps = np.log(element)/np.log(math.pow(2, 1/12)) - np.log(base_freq)/np.log(math.pow(2, 1/12))
    below_freq = base_freq * math.pow(2, 1/12 * int(steps))
    above_freq = base_freq * math.pow(2, 1/12 * int(steps + 1))
    sharp = False
    if(abs(element - below_freq) < abs(element - above_freq)):
      selected_freq = below_freq
    else:
      selected_freq = above_freq
    #print(selected_freq)
    #convert selected closest theoretical frequency to corresponding string version of note
    octave = int(np.log(selected_freq)/np.log(2) - np.log(base_freq)/np.log(2))
    iterations = 0
    char_val = 67
    val = base_freq * math.pow(2, octave)
    while(val < selected_freq):
      val = val * math.pow(2, 1/12)
      iterations = iterations + 1
    #1, 3, 6, 8, 10
    conversionArr = [0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 5, 6]
    if iterations == 1 or iterations == 3 or iterations == 6 or iterations == 8 or iterations == 10:
      sharp = True
    iterations = conversionArr[iterations]
    if iterations <= 4:
      char_val = char_val + iterations
      char_val = str(chr(char_val))
      if(sharp):
        char_val = char_val + '#'
```



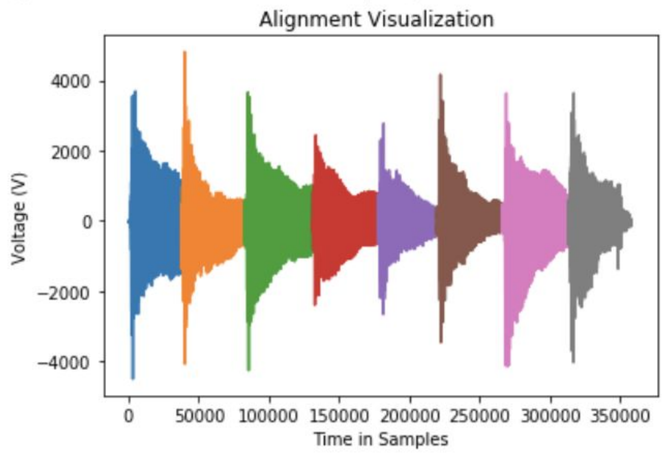
```
    else:
      char_val = 64 + iterations - 4
      char_val = str(chr(char_val))
      if(sharp):
        char_val = char_val + '#'
    note = char_val + str(octave)
    tuple_val = (note, note_duration)
    notation.append(tuple_val)
  return notation

#call function to perform audio to note translation
def performAnalysis(data, bpm, samp_frequency, calib_rate):
  grouped_data1 = partition(data, bpm, samp_frequency)
  #visualizeAlignment(data, grouped_data1, ' ') #uncomment to visualize alignment
  frequencies = []
  #plotFFT(grouped_data1[0], ' ', calib_rate) #uncomment and indicate index or array to view fft
  for i in range(len(grouped_data1)):
    frequency_v = resFreqs(abs(fft(grouped_data1[i])), calib_rate)
    frequencies.append(frequency_v)
  score = createNotation(grouped_data1, frequencies, bpm, samp_frequency)
  return score
```

```
Drive already mounted at drive; to attempt to forcibly remount, call drive.mount("drive", force_remount=True).
```

```
[ ] the_rate, CM = wavfile.read('/content/drive/My Drive/Table 7/Final Report/recording data/wav files/cM60.wav')
    cali_rate = the_rate * 2.04
    the_score = performAnalysis(CM, 60, the_rate, cali_rate)
    print(the_score)
    #insert more data to test
```

```
[ ]
```

```
[('C4', 0.8639455782312925), ('D4', 1.0124716553287982), ('E4', 1.1020408163265307), ('F4'
```



```
[ ]
```

References

Khondaker, Mahamudul Karim. “Arduino Sound Sensor Module / Sound Sensor with Arduino Code.” *Arduino Project Hub*, 15 Mar. 2018, create.arduino.cc/projecthub/karimmufte/arduino-sound-sensor-module-sound-sensor-with-arduino-code-868d55.