Physics 129L Final: Python Guitar Tuner

Christopher Sellgren

March 2023

1 Materials

This program uses a Python program to analyze a stream of audio data captured by a USB Microphone attachment. This project used a USB microphone manufactured by the Chinese audio hardware company ChanGeek. I used the CGS-M1 model, which received positive reviews on Amazon. The device was purchased from Amazon: here.

The code for this program runs using the Python3 interpreter with several popular libraries. NumPy is used for basic mathematical calculations and a fourier transform. The SciPy package is used to incorporate a signal processing tool called a Hamming Window, and to use a quadratic interpolation. The PyAudio library, installable in the terminal via the command:

```
$ pip install pyaudio
```

is used to extract audio from the microphone and convert into numerical data corresponding to the intensity of the incoming sound wave during the sampling interval. MatPlotLib is used to make simple plots of the raw sound wave sampled, and of the Fourier transformed frequency spectrum. Finally, the package TkInter is used to create a very rudimentary graphical user interface that the program runs in.

2 Project Description and Functionality

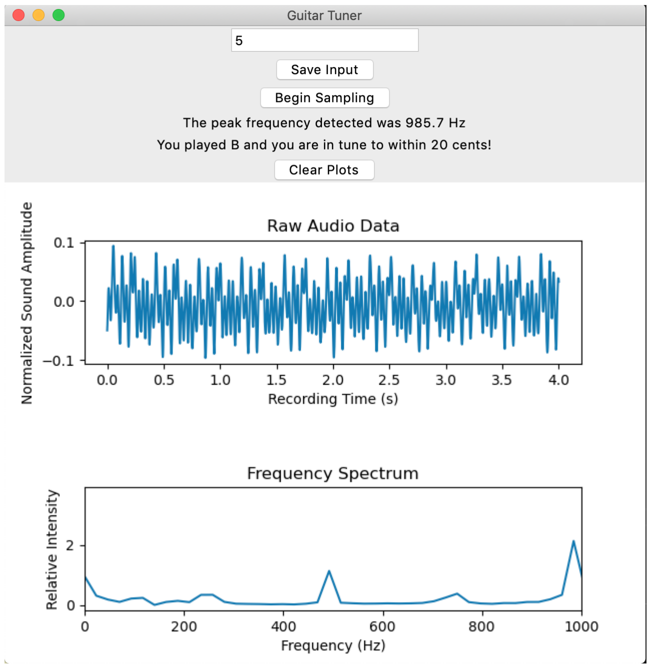
When launched, the program will open a basic graphical interface. The interface prompts the user to enter a length of time that sampling should occur for. A reasonable length is around 3-10 seconds. After inputting the sample time into the box and clicking the "Save Input" button, the user can click the "Begin Sampling" button to initiate the tuner program. The button calls the function tuner(), which will record audio data from the microphone over the specified time interval, run a series of functions to analyze the audio data, determine the primary frequency being detected, and compare it to known frequency values for different notes a guitar could play.

The code creating the graphical interface uses the simple and standard syntax of the TkInter package. The most notable part of this code is the function saveinput(), which is used to ensure that the input the user supplies can be interpreted as a floating-point number. If the user inputs a non-numerical input, the console will print an error message. If no input is supplied, the default sampling time is set to 4 seconds. The program begins with the takeaudio() function, which does the primary audio sampling and analysis for the program. The program begins by opening a PyAudio Audio Stream, where microphone data are converted into 16 bit integers and stored in the audiostream variable. Once the sampling has run for the full time specified, the audio stream is closed and the data is read. The data is converted into an array of 16-bit integers using the numpy.frombuffer() command. The sound data is then converted into normalized floating point numbers by dividing each number by 2^15.

Once the intensity data (as a function of time) is stored as floats, the frequency spectrum is determined through an FFT. To minimize erratic behavior of the spectrum at the tails and highlight the frequencies we are interested in, the spectrum is smoothed on the edges using a Hamming Window function. This is a common choice in signal processing that I learned about here. Our result is a 1-D array of useful intensity data as a function of frequency. To convert the frequency spacing from list index into a value in Hertz, we use a conversion variable which based on the ratio of the sample rate to the sample size (this variable is defined at the top of the code).

The next step in the process is to determine the frequency with the maximum intensity. The search is limited to a range from the minimum frequency of guitar string to the maximum guitar string frequency. Once the appropriate index is determined, a quadratic interpolation is used to fine tune the guess for frequency of maximum intensity. Once the best frequency is determined, in Hz, we convert the frequency measurement to cents, and determine how far it is from our basic tuning frequency by taking its modulo with 12 (there are 12 semitones between each full harmonic). This procedure is explained in more depth in the comments in the code. The final results of this sampling and analysis procedure are plotted using MatPlotLib. The plots are displayed in two canvas widgets at the bottom of the GUI.

Once the program has determined the frequency of maximum intensity with respect to the tuning octave, it is passed through a function of conditional statements to determine which guitar string note it is most similar to, and how far in/out of tune it is from that note. This check is done by the checknote() function, which runs a similar procedure for each of the 5 strings on a guitar. The function checks whether the frequency relative to the octave is closest to the specified note, and if it is, it checks whether the measured frequency is in within 20 cents of that note. If it is not in tune, the function checks whether it is flat or sharp with respect to the established frequency for that note. The function returns a string declaring the status of the note (which note it is most similar to, and how flat/sharp it is). These notes are added as label widgets in the GUI. A button is added that will clear out the prior widgets by running a function clearplots, which clears all canvases on the plot (in case of multiple).


The peak frequency detected was 985.7 Hz

You played B and you are in tune to within 20 cents!

Figure 1: The results of the guitar tuner program for the B string. The results show that the frequency spectrum has two main peaks. Although the peak with the highest intensity is actually the second harmonic of the note we played, the conversion of frequency to cents and the comparison to the octave frequency through the modulo 12 allows us to read the correct note for any upper harmonic.

3 Results

The execution of the program is demonstrated thoroughly in a video attached in the tar file called demonstration.mp4. In the video, I walk through the process of the program being executed. I show that the program launches and shows an initial user-input box, where the user should input the length of the sampling window they want in seconds. Then, they press “Begin Sampling”, and play the guitar string they’re trying to tune. I demonstrate the results of this for various strings. The result is that the interface will print out a message describing what note was played, and whether it was in tune, flat, or sharp. The results for one such case are shown in Figure 1.

Many aspects of the program had to be adjusted as the development progressed. The windowing function and the trimming of the frequency range being inspected were necessary changes that occurred after the tuner was consistently reading frequencies that were caused by ambient noise or other perturbations. The cutoff thresholds for where each note should be considered were also tuned to optimize performance. One of the biggest challenges was reading an accurate value of the peak frequency from the spectrum. Although many samples were taken per second, the spacing between frequencies in the spectrum after Fourier transformation was quite large, around 20Hz. The quadratic interpolation around the peak frequency was a tool used to make the estimate of the peak frequency more precise. However, repeated measurements of the peak frequency produce an estimated uncertainty of around 10Hz. This informed the cutoffs for when a note should be considered in or out of tune.

Other various bugs were encountered, such as a failure to detect values close to A. This was occurring because when the conversion to cents and then to the relative frequency was conducted, the result was around 11, and relative value of A was set to zero, because it is the semi-tone in the octave that all the other notes are compared to. Thus the code had to be modified to check for relative frequencies between 0 and 1 and between 11 and 12.

The final result of this program is likely too course to be much more useful than a well-trained ear with a tuning fork. However, if you were about to step on stage and you needed your instrument to be in the right ballpark, this program could stop you from getting boo'd off stage.