AUDIO VISUALIZATION & ANALYSIS

PROJECT REPORT

Daniel Hayes
Nate Soule
12/14/20

The purpose of this project is to further our understanding of audio data and visualization, while creating a tool that could analyze incoming sound for irregularities and potentially dangerous audio levels. Our original aim was to create something that would indicate to the user if there were harmful sounds present. We expanded the goal to include any sound outside of an indicated threshold. This would allow the program to support business asset management programs where fluctuating operational audio levels could suggest unusual behavior or be an early indication of faulty equipment. We had hoped to be able to reach a level of precision that would be useful in scientific research or sub-harmonic analysis, but due to limitations in hardware and knowledge we did not reach that goal.

To accomplish our goal, we created a program to receive real-time audio data and analyze it for discrete amplitudes (dB) along a range of frequencies (Hz). The program then matches the incoming data to preset templates, producing a report whenever the incoming data is outside of the template's acceptable range. If a report is made, an indicator will turn red to inform the user there were incidents, and the report will be saved as both a text and a wave file, allowing the user to visually see the minimum and maximum amplitudes of each frequency, or to play back the audio recording and see fluctuations in amplitudes over the duration of the incident. The templates themselves can be either created manually by entering desired thresholds for ranges of frequencies, or created by taking a sample of audio data and using the minimum and maximum amplitudes of the sample to generate a threshold.

We began the project expecting it to be a lot of work. Neither of us had any experience coding graphics or audio systems, meaning we would be learning most of it from scratch. We were warned that the project was ambitious and would take at least a month's worth of learning, which it did. To efficiently tackle the program, we decided to break the work into three parts: digital audio processing; graphic user interface; and computation/visualization of the data. Nate has background as an audio engineer, so he concentrated on audio processing and test signal generation. Daniel is interested in GUI design, so he focused on building the interface. The computation and visualization was dependent on the other sections, so we worked together on those sections.

Audio Processing:

The core of the program is its ability to record, play back, and process digital audio signal data in real-time. There are several Python audio processing modules from which to choose and many were examined for this project, such as Pydub, sounddevice, playsound, simpleaudio, winsound, wave, and PyAudio. The PyAudio and ‘wave’ libraries are used here because PyAudio provides python bindings for PortAudio which allows more base-level access to signal data manipulation and cross-platform functionality. ‘Wave’ is a relatively simple reading and writing module for audio wav files. The aim is to use packages that most GUI formats can accept easily and whose data is accessible.

The data source for our project is binary audio supplied by an input device, such as a microphone, or a wav file. A cursory understanding of classes was necessary to utilize the stream class of PyAudio for reading and writing buffers of bits. We had to learn how streams could be used to read and write binary data, as well as how to save the data to both wav and txt files. One of the first hurdles we hit was in getting usable data for plotting amplitude by frequency. It would have been nice if the stream function turned everything into arrays for us, but that wasn't to be.

It was essential to learn the fundamentals of digital waveform processing to isolate the components of the data. The first topic is the Nyquist Theorem which states that “in order to adequately reproduce a signal it should be periodically sampled at a rate that is 2X the highest frequency you wish to record (Capturing Images).” This gave us our maximum frequency range of ½ our sample rate. As our original idea had been to test for human safety, we went with the audio standard of a 44.1 kHz sample rate. If we wanted to measure ultrasonic waves we would need to increase the sample rate considerably, which is something we would likely do in a more complete form of the program. The other limitation of waveform processing we encountered was the bit rate. The bit rate limited how many frequencies we could plot at once. With a 1024 bit size we could only break the data into 513 frequencies evenly distributed over our detectable range. A higher bit rate would result in higher precision, but also in ever increasing processing power, a problem that we encountered frequently.

An audio spectrum is a frequency domain representation of an input audio’s time domain signal. In order to plot a frequency spectrum, the data must be converted from the time domain to the frequency domain. To make this conversion, we had to learn about the Fast Fourier Transform (fft). Numpy has an .fft method to convert incoming data to a complex np.array of complex coefficients. The array contains two arrays each holding the real component and the imaginary component of the coefficients. The fft method splits the time signal into frequency positions based on sample rate and size and also returns the amplitude (power) for each of those positions. For our purposes we only needed the real values and had to figure out a way to reduce the arrays to simple, real numbers. We had a great deal of difficulty at this stage with finding the data we needed, converting it to arrays, and ensuring the arrays were of the correct format and shape for visual representation.

Now that we were getting audio data into Numpy arrays we could begin to work with it. One of the goals of the project was to allow the user to create templates based on incoming or saved audio data. In order to do that we had to create additional arrays that would save the maximum and minimum amplitudes for each frequency. By creating these arrays during the stream we were able to plot them at the same time as the plot of incoming data. This creates an ongoing display of current audio information, and the minimum and maximum amplitudes for each frequency since the sampling began. At this point it was just a matter of combining these arrays into one 3 by 513 array and saving the data to a text file. Now any audio sample could be turned into a template for later use.

Graphic User Interface:

All we knew about GUIs going into the project is that we did not want to create a new one completely from scratch. While the general idea behind creating an object, waiting for mouse input, and checking if the input was within the range of the object isn't hard, going from that into a full GUI would be daunting, and likely quite ugly. We set to find a module that would provide the tools required for a functional GUI. While there are a number of modules that could have suited our purposes, two stood out as the most used, Tkinter and PyQt5. PyQt5 was

reported as harder to learn, but easier to use once learned. It also provided cleaner images and displays than Tkinter, therefore we chose to use PyQt5.

What we quickly learned while researching how to use PyQt5 was that we knew very little about classes and would need to learn how to use them to make a functional GUI. Therefore, the first few days of GUI development was spent learning about classes and objects in Python. Using sources such as our text, Zelle, educational YouTube videos, and stack exchange posts, we learned the general ideas of classes, enough to begin work on the project.

While learning about classes, we came across a PyQt developer's tool that would allow us to easily make a GUI by dragging the objects we wanted onto an easily customization backdrop. Being gluttons for punishment, we ignored this tool in favor of doing everything by hand, which was probably a good idea. Our understanding of how classes work had to develop significantly, in a way that using the developer's tool could not have matched. By the end of the project classes were making things quite simple for passing around variables and keeping the code relatively orderly.

Unlike the audio analysis section of the project, where there was a lot of research put into the theories and limits of audio data, the GUI was more immediately applicable and produced easy to see results. Once we had a general idea of what we wanted the GUI to look like and what we wanted the user to be able to do, it was just a matter of finding the PyQt5 object to make it work. One day would be making push buttons, the next was radio buttons. It was surprisingly easy to create buttons and assign them functions. Creating tabs took about 20 minutes. There was a decent amount of coding required, 3-4 lines for every object made, not including their functions, but most of that code was repetitive.

One problem in PyQt5 is that while making buttons is easy, getting them positioned where you want can be quite tricky. To create our GUI we had to learn about layering layout objects in PyQt5, where each tab consists of a basic object with 2-3 additional layouts stacked within that object. Each layout containing it's own buttons, graphs, or tables. Space between objects requires its own spacer object, which is actually harder to code than a button is. Even after figuring out how to make and add these spacer objects, there are some parts of the layout that seem to defy placing, which resulted in one of our buttons not lining up properly. Far more time ended up being spent on spaces than buttons, discounting functions.

The GUI functions are also where a lot of our coding occurs. This is particularly true on the second tab for the creation of our template table. The table had to pull data from a text file and convert it to three columns: frequencies; minimum decibels; and maximum decibels. Each space in the table is treated as an object in PyQt5, meaning we had to have code that would take an array, convert it into a series of string objects, then place them into the table as Items, a unique PyQt5 object. Then we had to do the same thing in reverse to allow and edits to be saved. This was surprisingly difficult to do, but actually helped us determine what our audio data would have to look like.

Computation & Visualization

We were confident that the computations for template comparison would not be too hard. Once we had the templates and audio data in Numpy arrays we generated a Boolean column to compare audio data to threshold levels. We even added some code to allow the user to set a tolerance range, if they decided the templates were too strict. If any frequency’s amplitude is higher than the max or lower than the min, then the Boolean returns “False” and triggers an incident report.

The tricky part of the incident reports was making a report that would record for a set period of time then return to normal functioning. We wanted to ensure the incident report would cover a minimum of ten seconds, in case there was a single incident causing the report, but we also didn't want the program to create a million reports if there was a constant incident. To do this we created another test within the incident report that would keep the recording going so long as there was still a sound triggering the incident. There were some limitations to this, which will be discussed later.

The original code that inspired our program utilizes a matplotlib animation method called FuncAnimation or 'function' animation. Its purpose is to continually plot a function result every time that function is called. It is optimized to be very fast and works as a stand-alone program outside a GUI. Our code plots a line graph for every sample taken each second using an update_line function constantly acquiring new point data from an input stream. Unfortunately, due to limitations in PyQt5 and in timing mechanisms we could not make use of the FuncAnimation. Instead we created a QTimer that would constantly run the update function. This may have slowed down our program, though we can't be sure at this time.

Testing

Testing the program is as important as program itself. We had to create conditions by which our program failed or succeeded, and that required generating test files. Learning to generate synthetic wave forms in Python was enjoyable and interesting, but not originally part of the project. An unexpected beneficial side effect was that synthetic conditions actually illuminate program weaknesses we may not have seen otherwise. Specifically, signal noise and 'normalization' considerations were not part of the scope of this project. While the program still 'works,' asymptotes do emerge in our data whenever sound waves cancel or are not picked up properly.

We also had to test our program to make sure it was evaluating what we thought it was. To do this we compared dB levels shown by our program with online descriptions of dB ratings for different sounds. While there are some limitations, our data matches closely with what we expected. There was also extensive testing of user input, mainly to ensure the user could not overwrite important files, such as the Blank_Template file, from within the program. Similarly when improper data is entered, such as writing a word in a number column, the program should not crash. That said, there may be areas where this is not thoroughly tested.

Limitations & Troubles

There are a few functions we would like to add but could not, due to time and knowledge limitations. We would have liked the user to adjust the frequencies being analyzed, and to choose a precision level for analysis. Right now the code always covers 0-22050 Hz frequencies separated by 43 Hz increments. For scientific research, or anything dealing with infrasonic or ultrasonic pitches, we would need the precision to be far lower, ideally taking samples at fraction of a Hz separation. This can be accomplished by using the scipy module, but the level of detail takes too much processing power to graph, as it results in thousands of points. There is some question if we could even do an analysis in real time.

There is also a question of what exactly our amplitude is measuring. There are some factors beyond our control, such as built in microphone dampening, and some factors we just don't know how to address. One problem is that dB is a relative value to an arbitrary starting point. When talking about human perception, 0 dB is the threshold of human hearing.

Microphones will have their own unit of measurement as well. This means our dB measurement is using a base of 1.0, where that base is actually set by the microphone or wav file as being a 'unit' of amplitude. Exactly what that means is unclear and may vary by source. In our tests dB levels seemed to match up closely to what we would expect for a human hearing comparison, but there is a degree of uncertainty.

Our original hope was to be able to tell if a machine was having issues using sound. Our program will do that, but only if the sound signaling something is off is louder than normal operations. We would have to add a function to the program that creates templates over a time period and looks for unusual sounds over time. That way a rattling sound that might not be louder than normal could be picked up as being outside a time-based pattern of activity.

Perhaps our greatest problem and limitation has to do with processing speed. Our playback of data runs slower than real-time, and when we record in real-time frames are being cut out, resulting in shorter than actual recordings. The most egregious example of this limitation is during our incident report, where the graphing freezes, and the GUI stops responding as the report is made. The most telling part of this is that after the report is made, the data saved is complete as far as time and complexity, nothing is being dropped. This raises our confidence that the graphing portion of the code is slowing down the rest of the code. In order to solve this problem we would have to learn how to utilize threading, which we decided was too much to work on given our time constraints.

Conclusions

Dividing the work between us worked out quite well. There were times that we didn't always understand what the other person was working on, but for the most part we were able to share the highlights of our endeavors without both going through all the trial and error involved at each step. It also helped to have someone be able to look at a problem with fresh eyes. Understanding how the audio needed to be presented in the GUI helped with the analysis of audio data, and similarly understanding how the audio was being recorded helped with designing parts of the GUI.

One thing that could have been better about the division of labor was the lack of tangible success when working on the audio portions of the code. There was a lot of trial and error, a lot of research, but often little to show for it after a day's work. While the research lead to everything pulling together in the end, having smaller objectives that produced concrete results would have helped make things less frustrating.

While the project was quite challenging, it was very engaging and educational. We had reasonable expectations going into the project and budgeted our time accordingly. We were a little unprepared for the complexity of audio processing but had given ourselves enough time to work through it. There are certainly improvements to be made, but we accomplished our primary objectives, and learned a lot in the process. Overall, the project was quite enjoyable, and a good chance to explore the real-world uses of Python.

References

"Capturing images." September 9, 1999. Retrieved December 11, 2020, from http://microscopy.berkeley.edu/courses/dib/sections/02Images/sampling.html

Coders Legacy. *"PyQt5 tutorial: Python GUI with Qt."* October 22, 2020. Retrieved December 12, 2020, from https://coderslegacy.com/python/pyqt5-tutorial/

De Langen, J. *"Playing and Recording Sound in Python."* April 23, 2019. Retrieved December 11, 2020, from https://realpython.com/playing-and-recording-sound-python/

László, F. *"Simple spectrum analyzer in python using pyaudio and matplotlib."* January 5, 2017. Retrieved December 11, 2020, from https://gist.github.com/netom/8221b3588158021704d5891a4f9c0edd

Pham, H. *"PyAudio Documentation."* 2006. Retrieved December 11, 2020, from https://people.csail.mit.edu/hubert/pyaudio/docs/

Pham, H. *"Pyaudio.py · fossasia/PyAudio."* 2006. Retrieved December 11, 2020, from https://gemfury.com/fossasia/python:PyAudio/-/content/pyaudio.py

Pythonspot. *"PyQT5."* July 26, 2016. Retrieved December 12, 2020, from https://pythonspot.com/pyqt5/

Zelle, J. M. *"Python Programming: An Introduction To Computer Science, Third Edition."* 2017. Portland, Oregon: Franklin, Beedle & Associates.