Alexander Buchholz

CPE555

Audio Processing Project

**Abstract and Introduction**

Machine learning and artificial intelligence have become hot topics both in popular culture and as legitimate research subjects. In more recent times, machine learning algorithms are making their way into consumer products and applications. Speech processing, for example, is a feature present in almost all modern smartphones, carried around by large amounts of the population. Characters like “Siri” and “Cortana”, intelligent assistants made by Apple and Microsoft respectively, use speech recognition algorithms to interact with the real world. Such speech processing algorithms rely on higher level machine learning tactics, as well as lower level algorithms for processing audio concurrently with a machine’s other tasks. The purpose of this project is to focus on the latter aspect of the problem, processing audio concurrently, while also examining how it might fit into the bigger picture.

A Raspberry Pi 3B, USB Microphone, and external speaker with an audio jack, are used as the supporting hardware. A Logitech G930 is used as the USB microphone, but most USB microphones will work. Although not much hardware is required, Figure 1 shows the basic equipment setup. Additionally, TightVNC is used to remote into the Raspberry Pi form a laptop, and an HDMI monitor, keyboard and mouse are used as supplementary material.



Figure 1 - Equipment Setup

For software, Python 2.7.9 is used in conjunction with the PyAudio library. PyAudio is a cross-platform I/O library written for audio processing in Python. Python is a suitable programming language of choice, due to the large number of supporting libraries, such as PyAudio, for applications like audio processing.

**Work Accomplished**

The following Linux commands were used for initial testing and setup:

* alsamixerused to configure sound devices, input audio raised to maximum volume (Figure 2)

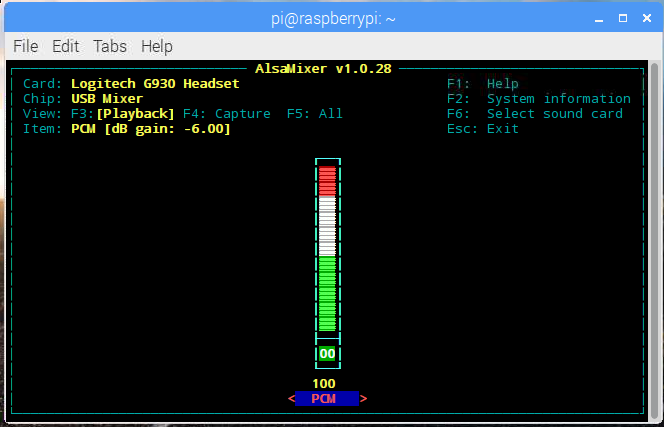


Figure 2 - alsamixer

* **arecord -D plughw:1,0 <filename.wav>** used to record different audio files
* aplay <filename.wav> used for initial playback and sound tests

Using these commands, three different .wav files were recorded for playback and testing: ToneA.wav, ToneCsharp.wav, and ToneE.wav. A function generator was used to generate a sine wave at frequencies 440Hz, 554.365Hz, and 659.255Hz respectively. When played together, these tones form a distinct harmonic on the A major scale. The **arecord command recorded these files at a fairly low quality, to the point where the pauses in the sound buffer are audible, but the harmonic nature of the chord made it easy to hear audio defects.**

Test code taken directly from PyAudio’s website was used as the starting point for code, to make sure that everything was working. PyAudioTest.py demonstrates one of the most basic uses of the PyAudio library─ playing audio. This code is shown in Figure 3, and is the building block for concurrent audio playback. Code screenshots were taken using Sublime Text 3.

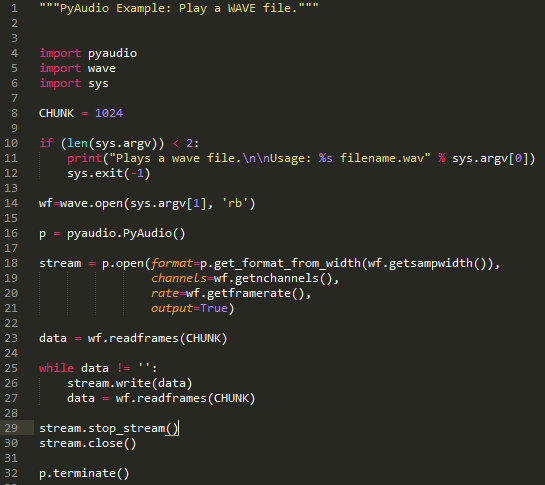


Figure 3 - PyAudioTest.py

Using the concurrent programming techniques taught in the class, this code was developed into PyAudioThreading.py. PyAudioThreading.py moves the audio playback test into the play\_audio function, and which is called in a thread created for each .wav file provided by the audioThread class. ConcurrentAudio.mp4 is a video file demonstrating the code in action. Since the three Tones all have different file lengths, a listener can listen to all three tones simultaneously, then listen to each tone drop out individually. PyAudioThreading.py is detailed in Figure 4.

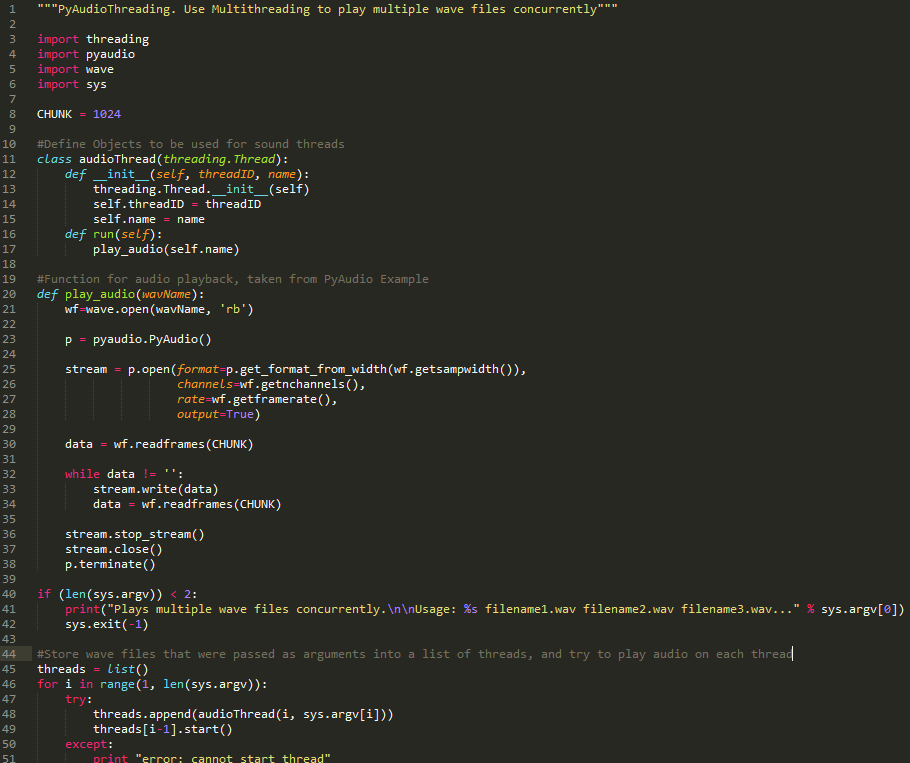


Figure 4 - PyAudioThreading.py

Although PyAudioThreading.py is successful at playing multiple audio files concurrently, efforts were made to try and improve performance, which was ultimately limited by using Python and a Raspberry Pi, as opposed to better real-time options. Figure 5 details PyAudioThreadingLock.py, which is an attempt to add locks to regions that may be critical to PyAudio’s performance. Unfortunately, getting the code to function with locks was unsuccessful, due to the design of the play\_audio function; The play\_audio function lasts for the entire length of the .wav files, rather than iterated over repeatedly. Passing a lock to each thread’s play\_audio results in the .wav files playing sequentially instead of simultaneously. This can be resolved by splitting the play\_audio function into pieces, and locking the while loop instead of the entire function. Even then, quality issues were already present when recording the Tone .wav files, so a different recording method would also be needed for higher quality.

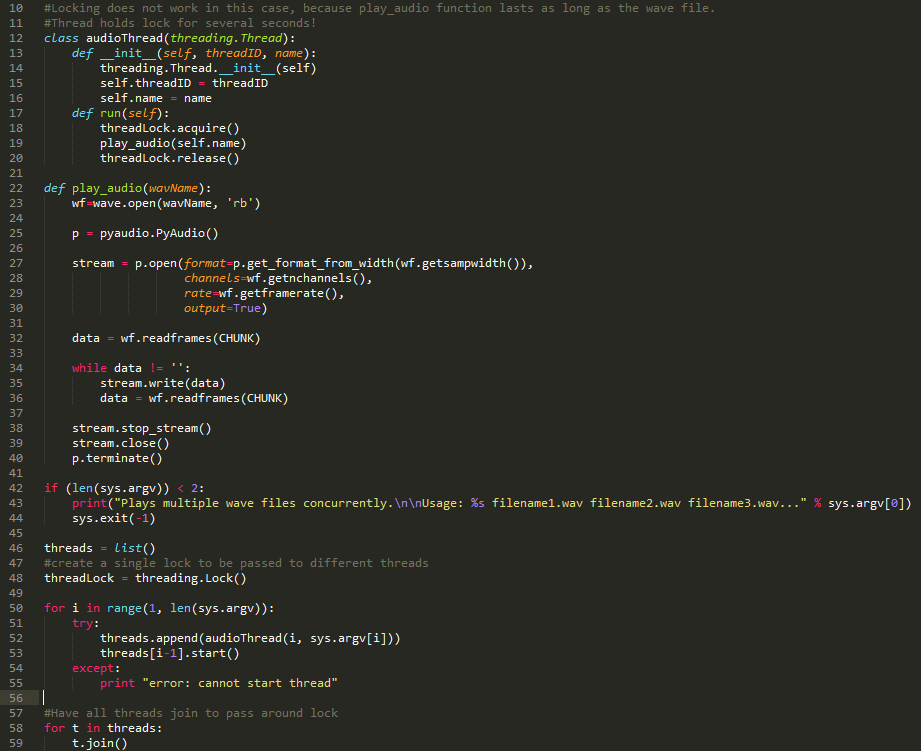


Figure 5 - PyAudioThreadingLock.py

**Compilation and Running Code**

To run the code, any USB microphone in Linux environment should work. Python 2.7.9 was used, but later versions will likely work as well, although this has not been tested. A network connection is required to set up and run the code.

Several Python libraries are needed to run the program. The following commands were used:

* sudo apt-get install python-pyaudio python2.7.9-pyaudio
* sudo apt-get install portaudio19-dev
* sudo apt-get install python-all-dev

If run in a Linux environment, and the files were just copied over, the .py files must be given executable privileges with chmod +x <filename>, and invoked with python <program filename> <sound files>. The test program provided by PyAudio takes a single .wav file as input for <sound files>, but the project code can play multiple files concurrently by listing out multiple .wav files delimited by a space.

Upon running any of the programs, PyAudio generates a large number of ALSA “errors” that are just for notification purposes only. Since none of the project programs have text outputs, these errors, shown in Figure 6, were not suppressed.

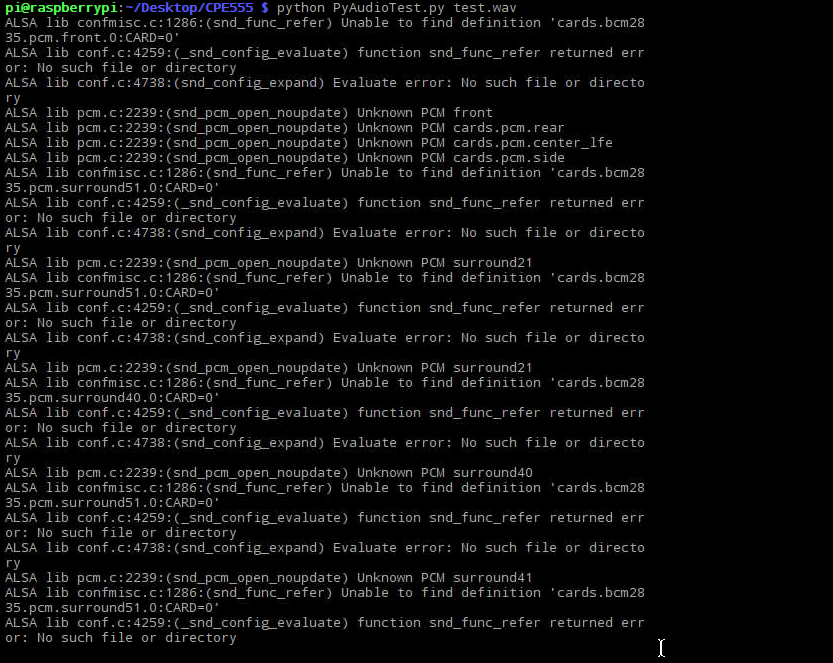


Figure 6 - ALSA Warning Messages

The code also sometimes reports API errors, due to the way that PyAudio calls on its API. A solution was not found for this problem, but the code will run after repeated attempts.

**Lessons Learned**

The first major difficulty in the project involved finding a suitable development environment.

The initial plan was to take in audio input through the GPIO pins, on the Raspberry Pi, and use Fast Fourier Transform (FFT) to process audio. Not only would this have been a more complicated solution, but the Raspbian operating system is not a true real-time system, and there was concern of low processing quality due to missed deadlines on FFT operations.

CMU Sphinx, Julius, and Jasper were also investigated. These applications are development tools designed specifically for Speech Recognition, and can be run in a Linux environment. While these tools provide more than enough functionality for speech recognition, they were not in the spirit of the class, since they took all of the “real-time” development out of the picture. The goal of the project was to develop something lightweight for the sake of learning.

A Voice Processing tool written by Steven Hickson was also tested, and while the speech recognition features worked pretty well, this tool did not allow much flexibility for developers to write their own voice processing modules.

In the end, PyAudio was chosen, with Python 2.7.9 as the development environment. Unlike the other tools mentioned, PyAudio provides flexibility for developers while remaining a relatively lightweight package.

While CMU Sphinx, Julius, and the other tools were not used in the final result, the documentation and resources made available by those projects were instrumental in developing an understanding of speech recognition. To develop speech recognition software with PyAudio, microphone data can be read in, signal processing can be used to detect features such as phenomes, and those features can be used to develop an acoustic model. After significant training, the acoustic model can be paired with a language model for speech recognition.

Unfortunately, the time to develop such software was underestimated, and the project was severely over-scoped. Development then shifted towards a much simpler goal─ create a development environment where audio can be read in and processed concurrently. It would have been nice to work with Speech Recognition and Machine Learning, but regardless of the outcome, a ton of lessons were learned while developing the project, which is a fine outcome for a project like this.

**References**

Reference 1 was used as a start to help configure the Raspberry Pi for audio input, as well as for general information on different resources available. Reference 2 was used for code examples and for the PyAudio library. References 3, 4, and 5 were used for information on different speech processing tools and methodologies. References 6, 7, 8 and 9 were other hardware/software resources used for the project. Reference 10 was instrumental in understanding how multithreading libraries work in Python.

1. [**https://diyhacking.com/best-voice-recognition-software-for-raspberry-pi/**](https://diyhacking.com/best-voice-recognition-software-for-raspberry-pi/)
2. [**https://people.csail.mit.edu/hubert/pyaudio/**](https://people.csail.mit.edu/hubert/pyaudio/)
3. [**http://jasperproject.github.io/documentation/**](http://jasperproject.github.io/documentation/)
4. [**https://cmusphinx.github.io/wiki/tutorial/**](https://cmusphinx.github.io/wiki/tutorial/)
5. [**http://julius.osdn.jp/en\_index.php**](http://julius.osdn.jp/en_index.php)
6. [**https://www.canakit.com**](https://www.canakit.com)
7. [**https://www.raspberrypi.org/**](https://www.raspberrypi.org/)
8. [**http://www.tightvnc.com/**](http://www.tightvnc.com/)
9. [**https://www.sublimetext.com/3**](https://www.sublimetext.com/3)
10. [**https://www.tutorialspoint.com/python/python\_multithreading.htm**](https://www.tutorialspoint.com/python/python_multithreading.htm)