Making a Keyboard Synthesizer with Raspberry Pi/Python

Maxwell Bowen and Noah Schumacher

June 12, 2018

1 Modules and Materials

In this project, we use several new modules to create a keyboard synthesizer that plays one keyboard octave of audio and produces a live waveform and power spectrum graph of the audio. The new modules required for our program are the PyAudio module and the keyboard module, both of which can be installed from the Linux command line on a Raspberry Pi. To begin, elevate to the root user by entering "sudo su" into the command line. Once in root, run "python3 -m pip install keyboard" to install keyboard and "sudo apt-get install python-pyaudio python3-pyaudio" to install PyAudio. These modules, in addition to the modules already installed on the Raspberry Pi by default when running the Physics 129 update script, are all the software that is required for this program.

For hardware, the normal USB keyboard and either a HDMI monitor with speakers or independent speakers/headphones with an auxiliary jack are required to put inputs into the synthesizer and listen to the output. When testing our program, we used AudioTechnica and Skullcandy headphones with an auxiliary jack purchased from Amazon. Any standard pair of headphones or small speaker with and auxiliary connection should work fine.

In order to run the program, the user must remain elevated in root, and run the python file labeled "final_project_audio.py" This file has a hashbang line to use python3, so the file name is all that is required to run it.

2 Design

In our program, we wanted to produce pure, tonal sounds with 13 keys on the keyboard to simulate playing one octave of a musical keyboard. Pure, tonal sounds are simply sine waves that follow the equation:

$$y = V * sin(2\pi * f * t) \tag{1}$$

where V is a volume corresponding to the amplitude of the wave, and f is the frequency of the wave. In the standard musical notation, there are 12 distinct notes per octave, with the 13th note corresponding to twice the frequency of the first note. In this way, each note's frequency is $2^{1/12}$ times the last one. The standard note on which all the other notes' frequency is based around is called the "A4", which has a frequency of $440Hz$. The "A5", which is one octave above the "A4", therefore has a frequency of $2 * 440Hz$ or $880Hz$. Similarly, the "A3" is one octave below the "A4" and has a frequency of $440Hz/2$ or $220Hz$. Any note's frequency can be calculated by its integer distance of notes above or below the "A4" note by the equation:

$$f = 440 * 2^{n/12} \tag{2}$$

where negative numbers represent notes below the "A4".

In order to produce sound waves, speakers and headphones move a solid surface of different types back and forth at the given frequency to create pressure waves which then propagate through the air to our ears and are heard. To do this, the device the speakers or headphones are plugged into modulates the voltage applied to the speakers or headphones in order to create an electromagnetic force and move the surfaces. PyAudio binds to the device audio jack, takes arrays of different data types and converts them to voltages to be transmitted through the audio jack to create sound.

Our program, therefore, needs to read keyboard inputs and for each input produce an array numbers corresponding to a sine wave with the frequency of the traditional music scale. To do this, we use the *keyboard.is_pressed("key")* function, which returns a boolean argument based on whether the "key" specified is pressed or not. Based on whether or not each of the keys on our keyboard are pressed, we add up the arrays of sine wave data and feed it out to the PyAudio module. In order to speed up the addition of these arrays, we use numpy arrays, which can be added in parallel at a much faster rate. This process we put in a continuously looping "while True:" statement so that the program continuously outputs audio.

In addition, we want to be able to graph the waveform and power spectrum of the audio output. To do this, we will use matplotlib.pyplot to make both graphs. By using the matplotlib.pyplot.subplots() function, we can create both graphs in the same window. To create the power spectrum, we used the numpy.fft.fft() function on the audio output to produce a frequency-space data set, which we place in the subplots graph along with the raw data set from the audio output.

3 Results

In theory this project seemed quite straightforward but we quickly discovered there are huge limitations intrinsic to our project. The main problem we encountered was being able to produce continuous / uninterrupted sound. The nature of computer programs is linear in the sense that they complete tasks in an order. Because of this nature and the design of our program, we kept encountering consistent clicking or popping noises in our audio output. Figuring out what was creating this inconsistency and how to fix it took up 90% of our project time.

When we first produced sound it was terribly scratchy and poor sounding. We quickly realized that PyAudio required our passed audio data to be 32bit floats. This greatly cleaned up the audio quality but we were still left with consistent clicking. We now speculated that each click corresponded to a cycle of the loop playing the music and then having the regenerate the data each time. At first we tried to speed up the loop process, creating dictionaries for the keys and octaves, eliminating any redundant code. We eventually came to the conclusion that we needed to somehow overlap the playing sound with a pre-generated sound so that as one loop was ending another was already beginning. At this point our research led us to the PyAudio callback function.

The PyAudio callback function is a specific function that works slightly differently from the normal PyAudio.play() function. "In callback mode, PyAudio will call a specified callback function (2) whenever it needs new audio data (to play) and/or when there is new (recorded) audio data available. Note that PyAudio calls the callback function in a separate thread."(PyAudio documentation) This sounded like the perfect solution to our problem. The separate thread would allow for seamless playing of the generated sound and it would all be integrated by PyAudio itself. However once we implemented this mode we encountered as faster clicking in the audio then before. For some reason, in this mode the sound would not play the desired amount of time, almost as if PyAudio would cut the output and then look for new output on its own. We also realized that the Python interpreter never actually creates a separate process for the callback function, meaning that the program would still pause audio output to generate more sound. After playing around with this for awhile we chose to revisit our previous looping method.

Finally we started to make real progress when we realized that clicking was not ony a product of the lag time between the loops but of the phase mismatches in the ending and starting audio data. We discovered this when we decided to pre-generate the all the audio data for an octave outside of the while loop so that the lag time between each iteration would be significantly less. When this still did not affect the clicking sound we began inspecting the the actual audio data produced. After this we realized that the clicking might be a product of the new sound beginning at a different amplitude than the last was ending, producing a rapid change in voltage which produces the click. The strange thing was that the clicking did not seem to align with the length of music produced for each cycle. In other words if we generated 1 second worth of audio to play, we would hear 4 clicks per second. In an attempt to fix the phase matching we rounded off all the frequencies so that our length of audio generated would be a 2π multiple of the frequency. This was the fix we were looking for. All the notes now played nearly seamlessly with minor crackling. The crackling seems to be related to the limited computation power of the RasPi as there is no crackling or imperfections when the program is run on my Mac OSX laptop. In order to further increase the efficiency, we decided to compile all the numpy arrays for the sound data only when the octave is first set. This helped to further cut down on computation times between sound playing, making the crackling noise more bearable.

Now that the sound was acceptable we started to implement the real time plotting of the waveform and the power spectrum. As expected when we first implemented the live plotting the audio became incredibly poor. In a first attempt to fix this we researched ways to create faster live plotting in matplotlib. We came across the matplotlib.pyplot blit and draw_artist. This work to selectively redraw only the parts of the graph that have been changed in each update. Now this works fairly well but still deteriorated the audio. To completely mitigate the plotting interfering the with audio we decided to create a separate python program for plotting alone. This program is called final_project_graphing.py. We call this program in a subprocess call on our main program. With the two separate processes running we now have incredibly fast and accurate live animation of the power spectrum and the waveform, without any deterioration in the audio. Below are several examples of the plots produced from different notes or chords.

Once the plots and sound was working correctly the last bugs we had to fix were with the output and matplotlib.pyplot hot keys. The matplotlib hot keys would register when the produced graphing window was selected, such as any time this window was in fullscreen. For example, when the s key was hit, the ”save figure” window would pop up. We simply reassign the matplotlib hot keys to empty strings to get rid of this effect. Secondly we wanted the terminal output to not display the keys being hit so the information displayed would not be altered. To do this we added another subprocess command which executed ”stty -echo”. This suppresses the standard output of the keys to the terminal. We then call ”stty echo” at the end of the file to un-suppress the keys again.

In conclusion, the bulk of this project was in debugging the audio problems and increasing the efficiency of the live plotting. We also discovered that their are real limitations inherent in using a Raspberry Pi as a digital audio device. This is contrasted with a laptop computer where there are no audio impurities noticeable. This is simply due to the processing power of the device. However, using subprocesses and special plotting techniques we have effectively created a electronic keyboard, capable of producing and plotting 73 notes.

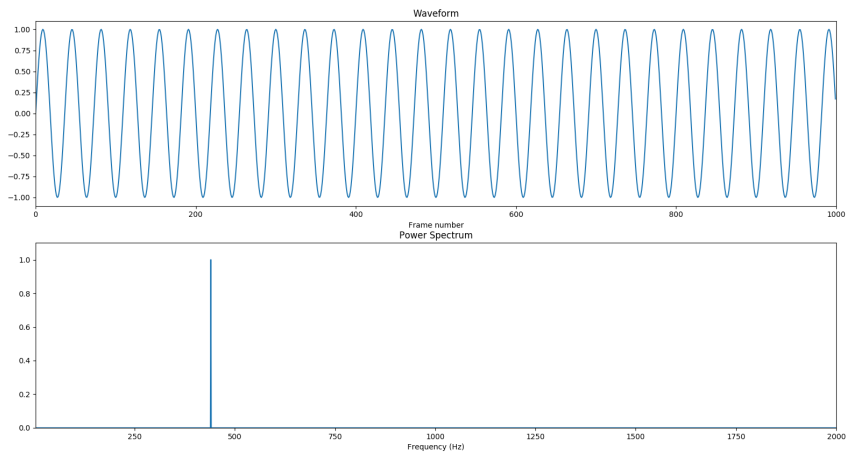

Figure 1: This shows the power spectrum and wave form of a Single A4 Note. The power spectrum clearly finds the frequency of the single note at 440Hz. The waveform is simply a pure sin wave.

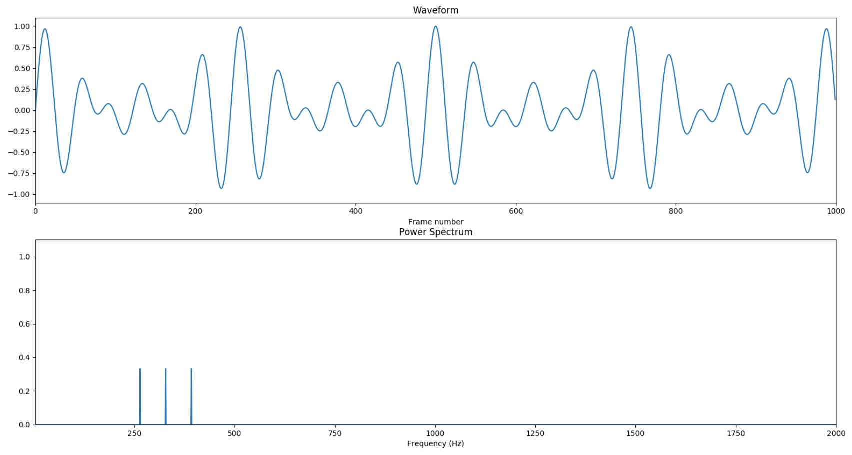

Figure 2: The C chord, consists of c, e, and g note. This composite sin wave is displayed in the waveform plots. The power spectrum accurately finds the three notes frequencies and identifies them. It also accurately finds the relative magnitude of each.

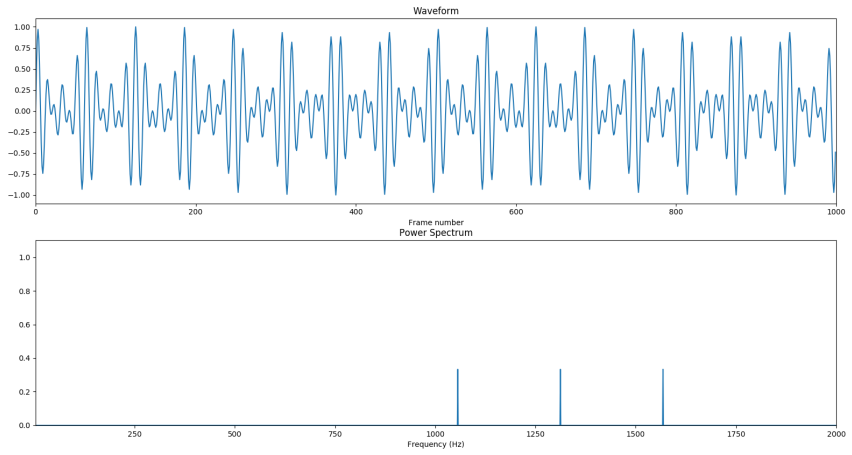

Figure 3: The same C chord as Fig. 2 but in the fifth octave. This plots shows the same thing as the above but at a higher frequency. This can be seen in the similarities of the waveform.

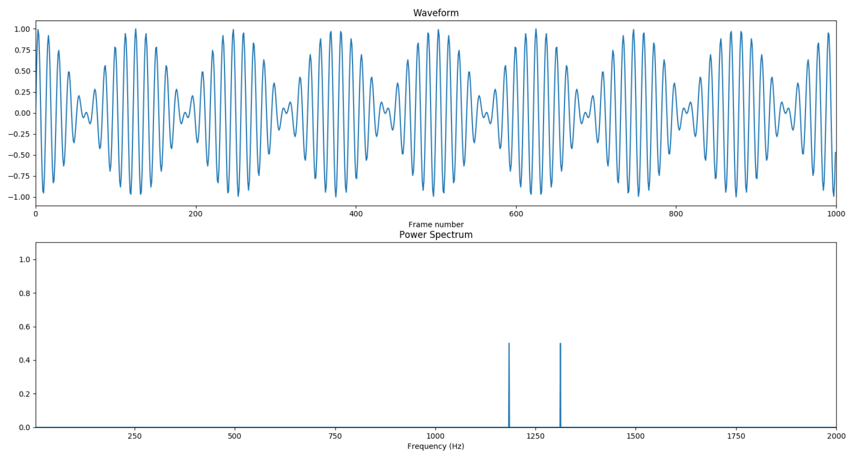

Figure 4: This is a D5 and a E5 playing at the same time. In this case the two notes being played are close in frequency and therefore create a beat frequency. This is expected and is audible in the audio playback.