Mathematically Generating Sound Waves for Music

Oliver Flatt

The aim of this paper is to demonstrate how to use mathematical models to generate sound waves using computers, and how to manipulate these models for desired effects. The application of this paper is a low-level control over the generation of sound for use in computer generated music.

I am personally interested in this topic because of the video game I am developing called Bearly Dancing. This game serves as a large research project for me, and I have been building a sound-generator for use in the game. I am passionate about the project, and chose this topic so that I can better develop it. I am able to directly apply this paper to my work on the game. What makes the game unique is that it is a rhythm game with an instrument-like quality. This makes it necessary to generate audio for the instrument with a high level of control.

The method of investigation is to research and implement various models of waves, as well as derive new waveforms and tools to manipulate them. These will be implemented in the programming language Python, using a library called pygame for playing audio, and the library numpy for representing the audio in a buffer. A basic evaluation of the success of the implementation is whether the correct pitch is produced and sustained. Beyond that, it can only be judged subjectively, based on what varieties are produced that sound good.

This paper will first explore the basics of sound waves. It will then cover generating some basic waveforms using different methods, and how these can be manipulated. Finally, it will explain the important role of volume in creating nice sounds, and how to represent and integrate volumes of sounds over time as well.

Some background information on how sound waves work is needed. Sound is a series of small changes in pressure and displacement over time. A person's ear sends these changes to the brain, where it is interpreted as sound. This means that only changes can be detected, and usually in wave form. The "shape" of the wave is the series of pressures plotted, and the frequency of the wave is how quickly it oscillates between different pressures. These two relative to time determine the sound we hear (Cole).

Figure 1- basic sine wave

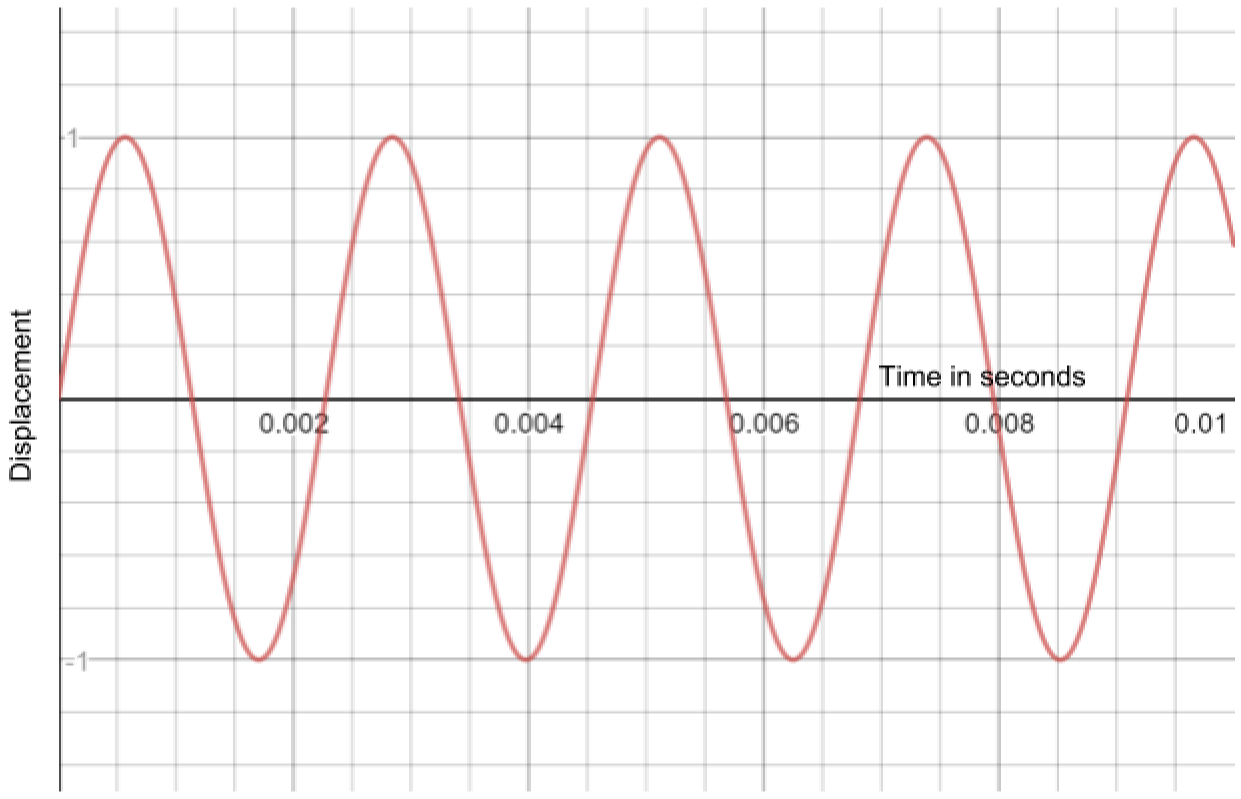


Figure 1 shows a basic sine wave. It is the core oscillation that all the other waves will be built around. This particular wave has a frequency of 440 Hz, which means that this wave goes through 440 cycles per second. It is commonly known as A4. The equation for a basic sine wave is below, in equation 1. First, we must negate the period of sin by multiplying by 2π. Then, multiplying the time in seconds by the frequency makes it oscillate that many times per second.

Equation 1- basic sine wave

$$f = frequency \quad A = amplitude \quad t = time\ in\ seconds$$
$$y = A * sin(2\pi * t * f)$$

The next basic functionality is changing the pitch of the tone. The pitch of a sound is determined by its frequency. To change the pitch of the sound, we modify the frequency variable according to Equation 2. There is a lot of music theory behind why this equation exists, but it follows the pitches that people have decided sound good.

Equation 2- equation for frequencies of notes

$$n = half\ notes\ away\ from\ note\ A4$$
$$a = \sqrt[12]{2}$$
$$f = 440 * a^n$$

One other problem to solve before getting into other types of waves is looping them. This is important so that a note can be played indefinitely without the entire thing being generated by the computer. This is easy enough to figure out because the period stems from the frequency, and the wave repeats after one period. After $1/frequency$ seconds, we can loop the generated audio.

Now that we can generate a basic sine wave and change its pitch, we can move on to other simple shapes for waves. It is also interesting to note here that when generating sound for a computer, the wave is encoded into two series of points over time in buffers, or lists. One represents the sound coming from the left speaker over time, and one represents the sound coming at the same time for the right speaker. Creating direction using computers by changing the amplitudes of the waves to each buffer adds a lot of interest to music. It is one of many examples of things that computers have enabled us to do that were not possible before.

Figure 2- basic square wave

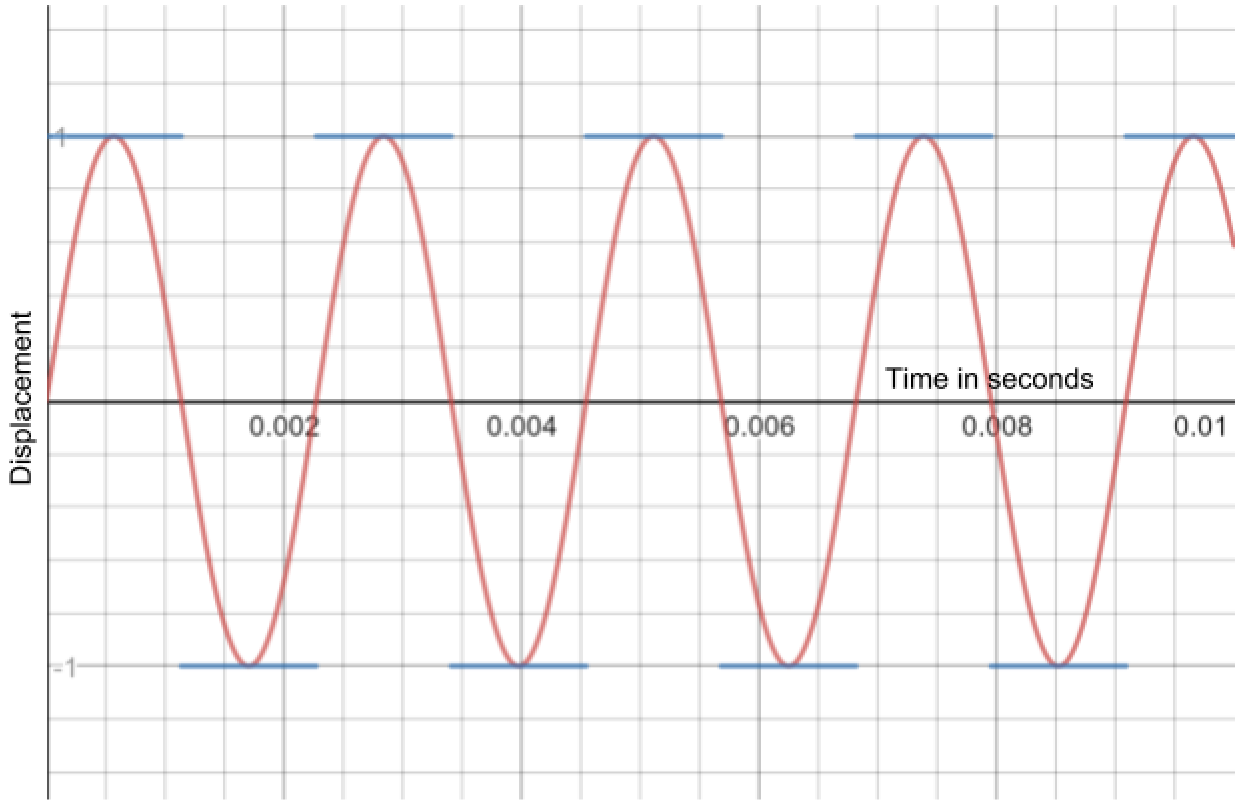


Figure 2 depicts a square wave in blue on top of the original sine wave from Figure 1. Notice that there is no line connecting the peaks and valleys of the square wave. This is because this wave was generated using the integer function. It rounds values, truncating them towards zero. This allows us to "jump" after reaching the next whole number, generating this wave. This method creates a pure, harsh sounding square wave that is famous for being one of the common sounds of early computers as it is fairly easy to create. Equation 3 used for that wave is below.

Equation 3- equation for basic square wave

$$y = (-1)^{int(f*2*t)}$$

Square waves sound extremely harsh compared to sine waves. One way to generate pseudo-square waves is through summations using sine waves with varying depths. These approximations of square waves through adding sine waves sound much better and natural than square waves, through the rough sound of the square wave has its uses. This variation in sounds relative to each other can make my video game more interesting and interactive.

This introduces what are called Fourier Series (Weisstein). From a musical standpoint, it is actually the same thing as adding and layering specific harmonics at lower amplitudes to the base sine wave. Harmonics natural occurrences where people also hear higher frequencies of a note at the same time that it is being played, with an additive effect (Bain). This adds depth to the sound. By using the square wave as a model and using Fourier analysis, we can create varied sounds with more depth.

Equation 4 below is a series that adds sine waves *n* number of times. Adding odd harmonics that have proportionally smaller amplitudes creates a wave that approaches the square wave. The approaching of the jump between peaks through this method is called Gibbs Phenomenon (Hazewinkel). Taking the original sine wave and adding smaller waves as harmonics generates waves such as those in Figure 3.

Equation 4- series for square wave approximation

$$n + 1 = \textit{number of partial sums added}$$
$$sin(2\pi * t * f) + (\tfrac{1}{3})sin((3) * 2\pi * t * f) + (\tfrac{1}{5})sin((5) * 2\pi * t * f) + \ldots$$
$$= \sum_{k=0}^{n} (\tfrac{1}{2k+1}) * sin((2k+1) * 2\pi * t * f)$$

Figure 3- square wave approximation graph

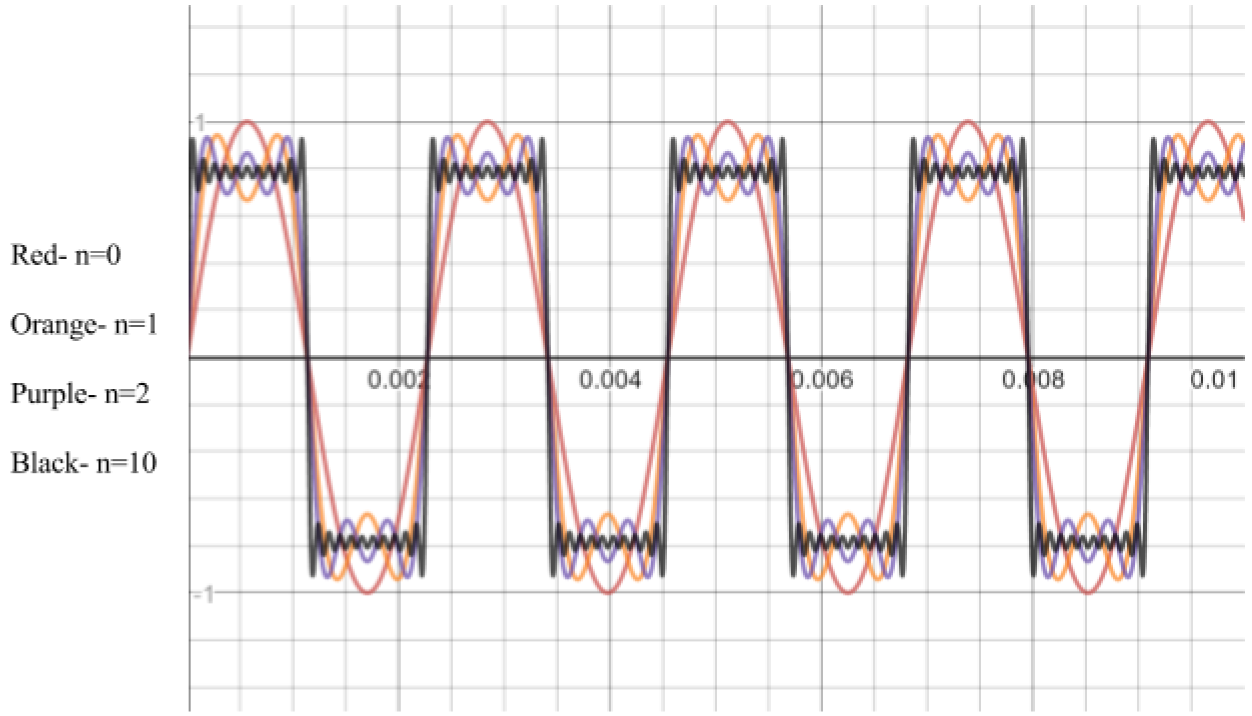


Figure 3 shows the summation approaching a an approximate square wave according to the Gibbs Phenomenon. By experimenting with adding harmonics in series, it is easy to discover another elegant function that creates a saw wave. The formula is more simple than that for the square wave because it simply adds every harmonic instead of the odds.

In fact, by experimenting with these Fourier Series you can create many interesting waves that are both visually and audibly pleasing. Instead of the classic saw wave, a more interesting example I created by adding two series together is shown below. Equation 5 adds a series that generates a saw wave and a series that adds every fourth harmonic above the original.

Equation 5- two series added

$$S1 = \frac{1}{2}\sum_{k=0}^{n}\left(\frac{1}{k+1}\right)sin((k+1)*2\pi*t*f)$$

$$S2 = \frac{1}{2}\sum_{k=0}^{n}\left(\frac{1}{4k+1}\right)sin((4k+1)*2\pi*t*f)$$

$$y = S1 + S2$$

Figure 4- two series added

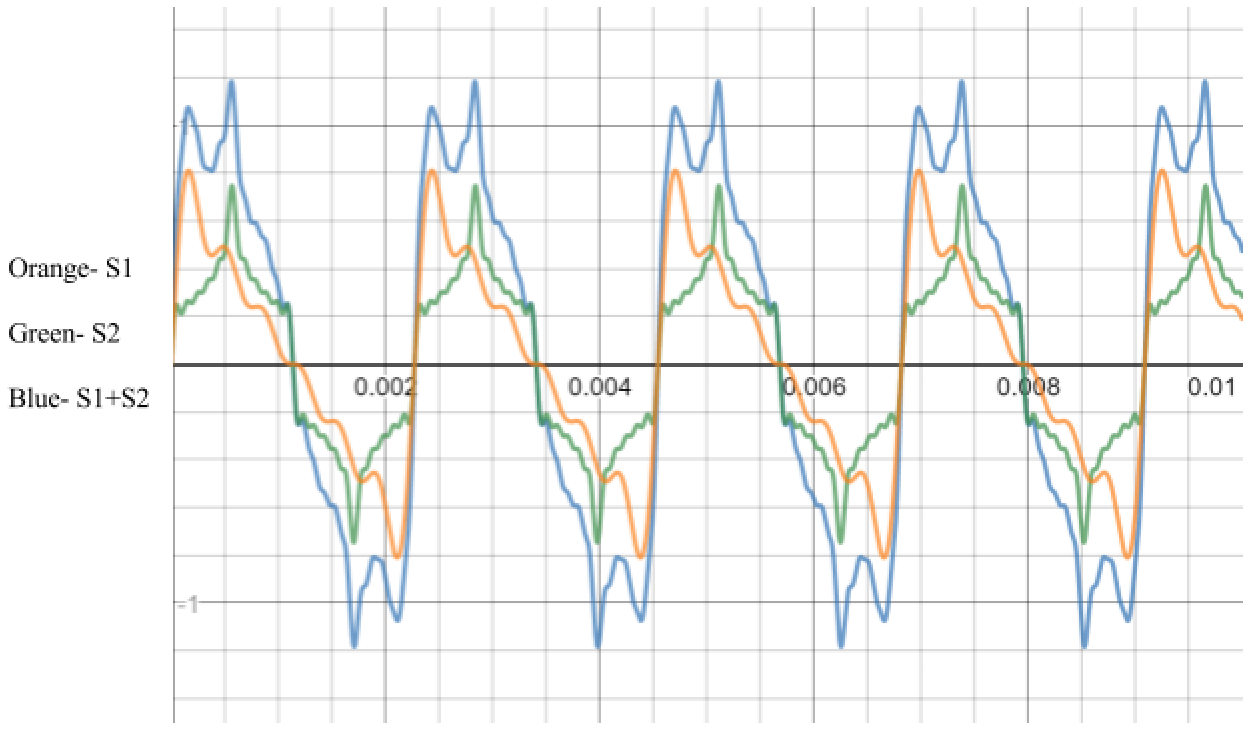
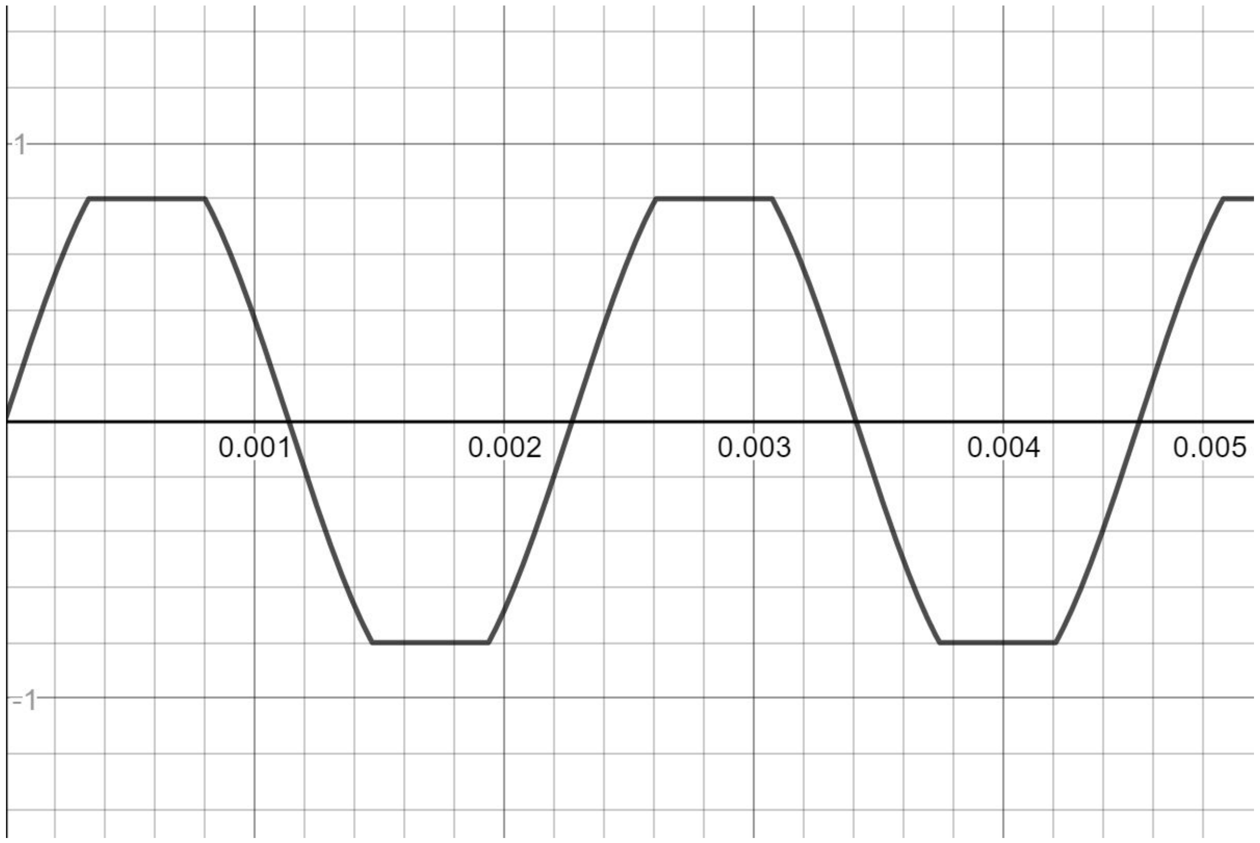


Figure 4 above shows these two equations and their sum. There are an infinite number of waves that can be generated through adding these types of summations, and this demonstrates just a few. A future project could be a random wave generator that procedurally generates equations for waves. I would like to explore this topic more, because it is very interesting. That being said, there are many more tools that can be used to shape waves besides Fourier Series. Modern synthesizers can employ many techniques, including those in this paper and more. One of these techniques is clipping. Originally discovered through trying to play sounds louder than possible with a speaker, this is the flattening of any part of a wave. This also creates very different types of waves. Figure 5 shows a sine wave that is clipped.

Figure 5- clipped sine wave



This clipping effect was achieved through a piecewise function that set a bound on the value of the normal sine wave at either end. Although it may not look like it, it has a large impact on what someone hears. It is difficult to describe the change, but it is like putting a mute on an instrument, making the sound softer. People's ears are very sensitive to these kinds of changes.

One last piece of generating good audio remains, and that is volume. By changing the volume of a wave over time, very nice effects can be produced. Part of the unnaturalness of much computer-generated music is its sudden and constant volume. As an example, the volume of a bell being rung over time will be used. It grows suddenly loud and then tapers off. Equations of volume over time can be calculated using a few points. Equation 6 below shows a piecewise function that models this change in volume of a bell.

Equation 6- linear simulation of bell volume over time

$$(0,\ 0)\ \mathit{to}\ (0.1,\ 1)\ \mathit{to}\ (1,\ 0.25)$$

Line 1

$$m = \frac{y2-y1}{x2-x1} = \frac{1-0}{0.1-0} = 10$$

$$y = mt + c$$

$$y = 10t \quad 0 \le t \le 0.1$$

Line 2

$$m = \frac{y2-y1}{x2-x1} = \frac{0.25-1}{1-0.1} = -\frac{5}{6}$$

$$y = mt + c$$

$$y = -\frac{5}{6}t + 1 \quad 0.1 \le t \le 1$$

Equation 6 produces values for an increase then decrease in volume that is similar to a bell sounding. However, linear representations of volume do not sound natural. Instead, using truncated sine waves between the points creates smoother transitions between volumes. This involves creating equations that have the correct bounds and periods to fit the points, from a min to a max. Calculations for this example are shown in Equation 7.

Equation 7- sinusoidal representation of volume over time

$$(0,\ 0)\ \mathit{to}\ (0.1,\ 1)\ \mathit{to}\ (1,\ 0.25)$$

First curve- equation for incr.

$$p = (1/\Delta x) \quad a = \Delta y/2$$

$$y = a * sin((t - x1 - \tfrac{\Delta x}{2}) * \pi p) + y1 + a$$

$$\mathrm{Y} = 0.5$$

$$y = \tfrac{1}{2} sin((t - 0.05) * \pi * 10) + Y$$

$$0 \le t \le 0.1$$

Second curve- equation for dec.

$$p = (1/\Delta x) \quad a = -\Delta y/2$$

$$y = a * sin((t - x1 + \tfrac{\Delta x}{2}) * 4\pi p) + (y2 - a$$

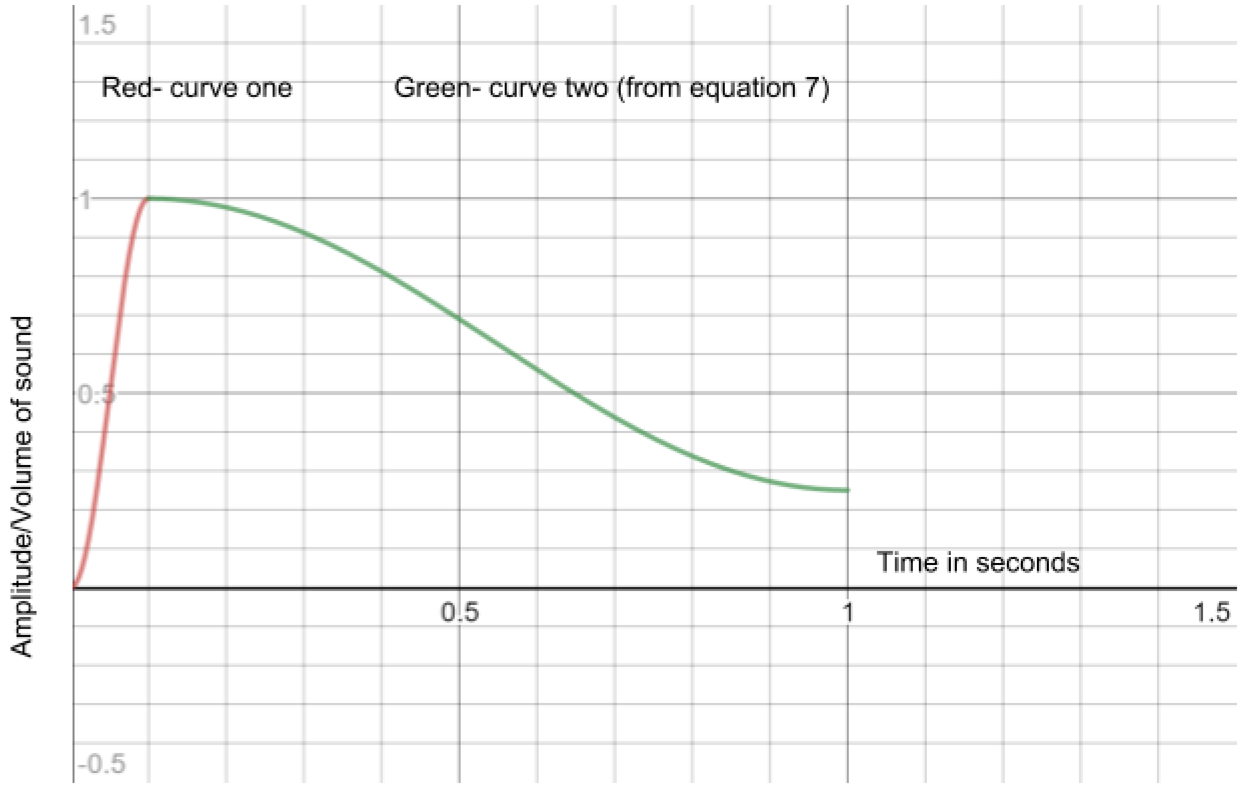
$$\mathrm{Y} = (1 - \tfrac{0.75}{2})$$

$$y = \tfrac{0.75}{2} sin((t - 0.1 + \tfrac{0.9}{2}) * \pi * \tfrac{1}{0.9}) + Y$$

$$0.1 \le t \le 1$$

The messy functions above that fit the graph of sine to a series of points create a nicer change in volume. Because it is laborious to write by hand, I have a script that creates these functions by looking at the points that I give it. Figure 6 below shows the two curves from Equation 7.

Figure 6- volume over time using two sine wave



After implementing the variation in volume over time, it became apparent that the original looping mechanism would not work anymore because the wave is generated with changes in volume. This was solved with a bit more infrastructure that first generates a wave with the sequence of volume changes, then a small portion of the wave to loop after it has finished so that the note can still be played indefinitely. It is also interesting to mention that when I first implemented this kind of looping, I was getting a blip between each looped segment. Initially I thought that my calculations were off, but it turned out to be a lag time when playing the sounds using pygame. I was able to reduce the latency by making the sound buffer argument much less when initializing the project.

This project has been very successful, and I have learned new things about math, coding, and music all at once. If I were to further develop the project, I would look into more ways the synthesizers alter sound. In this paper I explored additive synthesis, one of many types of wave synthesis used by modern synthesizers. Many of these synthesizers are still expensive. One application of this work is to build good synthesizing software that is open source, so that anyone may use it for free. This lowers one of the barriers to making music significantly. Currently there are projects being developed to do just that. One of these is Sonic Pi, a language for live coding that I have experience with.

My favorite thing that I learned during this project is using Fourier Series and generating many different kinds of waves. Synthesizing different skills that I have in a project such as this is fun, and allows me to express myself in a unique way. This paper has allowed me to explore sound generation in detail and improved my ability to manipulate sound for my personal project, Bearly Dancing. While this paper can provide many visuals of sound waves, I encourage that the reader listen to them for themselves. The source code to my project can be found using the link below:

https://github.com/oflatt/bearlydancing

Citations

Bain, Reginald. The Harmonic Series. University of South Carolina, 2003, in.music.sc.edu/fs/bain/atmi02/hs/hs.pdf. Accessed 1 May 2017.

Cole, Adam. “What Does Sound Look Like?” YouTube, NPR, 9 Apr. 2014, www.youtube.com/watch?v=px3oVGXr4mo. Accessed 1 May 2017.

Gann, Kyle. “Just Intonation Explained.” Kyle Gann, www.kylegann.com/tuning.html. Accessed 5 May 2017.

Hazewinkel, Michiel. “Gibbs phenomenon.” Encyclopedia of Mathematics, Springer, 2001, www.encyclopediaofmath.org/index.php/Gibbs_phenomenon. Accessed 5 May 2017.

Sievers, Beau. “A Young Person's Guide to the Principles of Music Synthesis.” Synthesis Basics, beausievers.com/synth/synthbasics/. Accessed 5 May 2017.

Weisstein, Eric W. "Fourier Series." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/FourierSeries.html. Accessed 28 April 2017.