Making Waves

An Exploration of Digital Sound Synthesis
Mathematics and Techniques

John Ruffer
ECE 4974
Fall 2012

Table of Contents

Introduction

Since the advent of accessible digital computing, musicians have looked to digital electronics to broaden the spectrum of sounds at their disposal and manipulate the listener's perception of their works. Notably, composers such as Brian Eno and Steve Reich pushed the envelope of academic music (and later, popular music) using computational techniques. The power of computation has since been developed and applied to a vast range of audio applications, providing various sophisticated methods by which unique, interesting, and complex audio signals may be produced digitally.

So, the aim of this independent study was to ascertain a better understanding of how sound is synthesized through hardware and software, thereby constructing a strong foundation upon which I may develop an advanced knowledge base of audio engineering techniques. As both a musician and a computer engineer, I have a great interest in learning and applying such knowledge to my work. The following report provides a brief summary of the topics I encountered as well as the projects I worked on to solidify the concepts I studied.

Topics

Digital Audio Basics

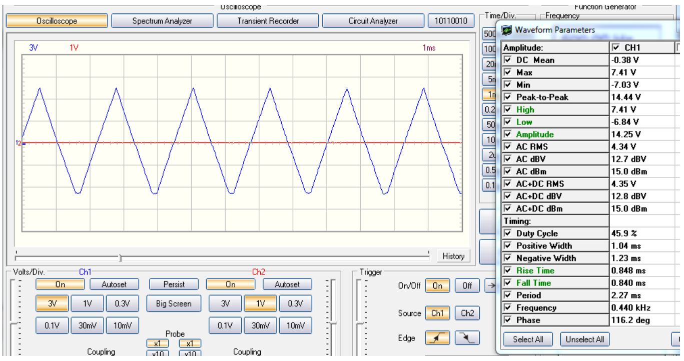
Sound in its natural form exists as a conglomerate of continuous waves. However, values and ranges of values can only be stored digitally as discrete units. So, this discrepancy necessitates that these continuous wave forms need to be somehow translated to a discrete representation. This is accomplished by taking values from a waveform at even time intervals, an action known as *sampling*. The time interval period at which samples are taken is known as the *sample period*. More commonly, however, audio engineers and electronic musicians refer to the samples taken per second, or *sample rate*. This is, of course, not to be confused with the more colloquial version of sampling, in which a musician will take clips of sound and music from other recordings and use those “samples” in his or her own music.

One may infer from the nature of sampling that, since samples are taken at time intervals, a certain amount of data is lost in the translation of continuous waves to discrete waves. Additionally, samples taken which fall in a particular step range will actually be quantified as the same discrete value, depending upon the kind of data representation used in hardware/software for the samples. For example, using a C unsigned char data type offers 255 values by which a waveform may be represented. So, some arbitrary wave may have several samples that, when converted to a value between 0 and 255, all result in the same value due to data truncation, despite having distinctly different initial values. As such, the resolution, or *bit depth*, of the samples is quite important. In WAV file formats, the standard bit depth is 16 bits.

Geometric Wave Forms

Geometric wave forms, while not naturally occurring waves, can be quite useful in electronic music composition and production, and are also a bit of an interesting exercise to implement in code. Square waves, rising and falling sawtooth waves, and triangle waves may all be classified as geometric wave forms. Interestingly enough, these wave forms can actually be synthesized using a technique known as *additive synthesis.*

Pictured below are a few example geometric waves:

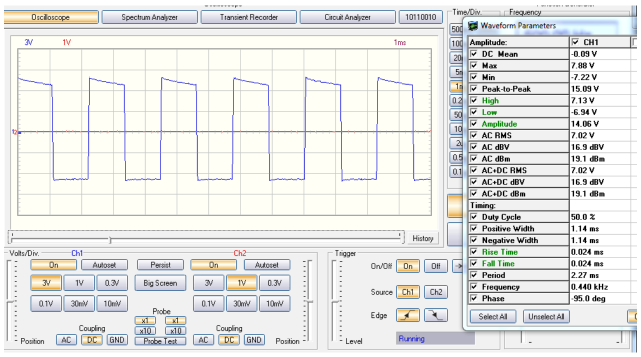


Waveform Parameters

Amplitude:	✓ CH1
✓ DC Mean	-0.38 V
✓ Max	7.41 V
✓ Min	-7.03 V
✓ Peak-to-Peak	14.44 V
✓ High	7.41 V
✓ Low	-6.84 V
✓ Amplitude	14.25 V
✓ AC RMS	4.34 V
✓ AC dBV	12.7 dBV
✓ AC dBm	15.0 dBm
✓ AC+DC RMS	4.35 V
✓ AC+DC dBV	12.8 dBV
✓ AC+DC dBm	15.0 dBm
Timing:	
✓ Duty Cycle	45.9 %
✓ Positive Width	1.04 ms
✓ Negative Width	1.23 ms
✓ Rise Time	0.848 ms
✓ Fall Time	0.840 ms
✓ Period	2.27 ms
✓ Frequency	0.440 kHz
✓ Phase	116.2 deg

Select All Unselect All

Illustration 1: Triangle

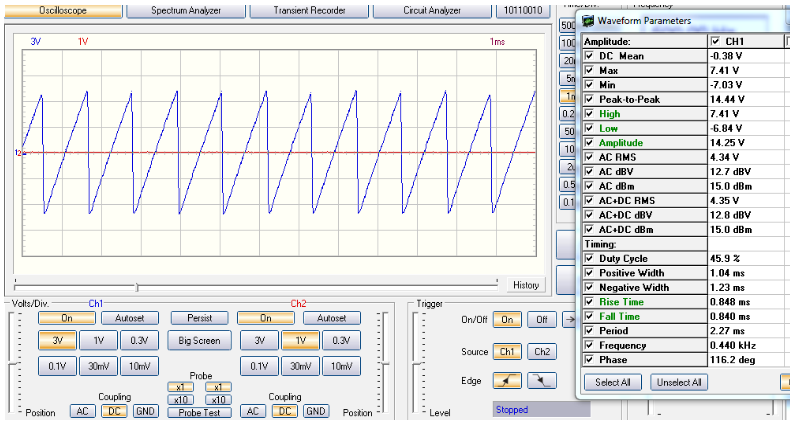


Waveform Parameters

Amplitude:	✓ CH1
✓ DC Mean	-0.09 V
✓ Max	7.88 V
✓ Min	-7.22 V
✓ Peak-to-Peak	15.09 V
✓ High	7.13 V
✓ Low	-6.94 V
✓ Amplitude	14.06 V
✓ AC RMS	7.02 V
✓ AC dBV	16.9 dBV
✓ AC dBm	19.1 dBm
✓ AC+DC RMS	7.02 V
✓ AC+DC dBV	16.9 dBV
✓ AC+DC dBm	19.1 dBm
Timing:	
✓ Duty Cycle	50.0 %
✓ Positive Width	1.14 ms
✓ Negative Width	1.14 ms
✓ Rise Time	0.024 ms
✓ Fall Time	0.024 ms
✓ Period	2.27 ms
✓ Frequency	0.440 kHz
✓ Phase	-95.0 deg

Select All Unselect All

Illustration 2: Square



Waveform Parameters

Amplitude:	✓ CH1
✓ DC Mean	-0.38 V
✓ Max	7.41 V
✓ Min	-7.03 V
✓ Peak-to-Peak	14.44 V
✓ High	7.41 V
✓ Low	-6.84 V
✓ Amplitude	14.25 V
✓ AC RMS	4.34 V
✓ AC dBV	12.7 dBV
✓ AC dBm	15.0 dBm
✓ AC+DC RMS	4.35 V
✓ AC+DC dBV	12.8 dBV
✓ AC+DC dBm	15.0 dBm
Timing:	
✓ Duty Cycle	45.9 %
✓ Positive Width	1.04 ms
✓ Negative Width	1.23 ms
✓ Rise Time	0.848 ms
✓ Fall Time	0.840 ms
✓ Period	2.27 ms
✓ Frequency	0.440 kHz
✓ Phase	116.2 deg

Select All Unselect All

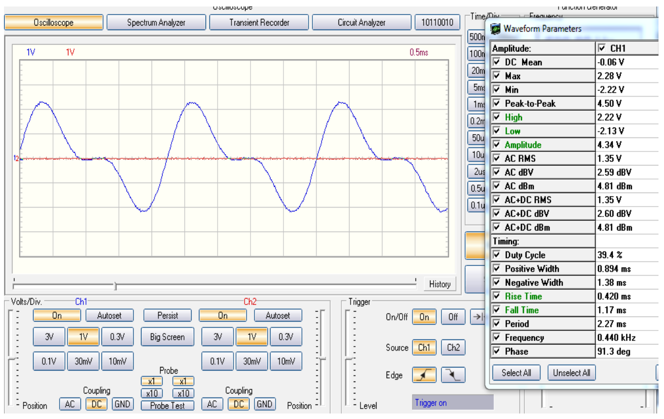
Illustration 3: Rising Sawtooth

The implementations of these wave forms are available on the blog for this independent study. See the Appendix for a URL.

Additive Synthesis

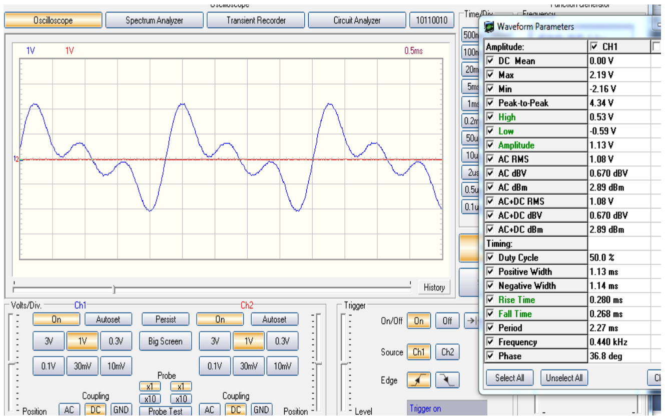
Simple wave forms such as the sine wave or the geometric waves previously mentioned are all well and good, but to provide wave forms of any auditory interest, algorithms to produce more complex sounds must be developed. One such solution is that of *additive synthesis.* Additive synthesis is rather straightforward in its implementation in that it transparently emulates the Fourier Series. As the name suggests, in the algorithm one simply sums wave forms at harmonic (integral) multiples of the fundamental frequency with the fundamental wave form. The result is much as the Fourier series describes: a complex wave form whose constituents are many simple wave forms. Of course, adding non-integral harmonics to the fundamental will result in more cacophonous, or inharmonic, sounds. It should also be noted that these sums are finite and most probably with in a very reasonable number due to computational limitations.

Pictured below are two wave forms generated with additive synthesis, summing two and three waves together, respectively.



Amplitude:	✓ CH1
✓ DC Mean	-0.06 V
✓ Max	2.28 V
✓ Min	-2.22 V
✓ Peak-to-Peak	4.50 V
✓ High	2.22 V
✓ Low	-2.13 V
✓ Amplitude	4.34 V
✓ AC RMS	1.35 V
✓ AC dBV	2.59 dBV
✓ AC dBm	4.81 dBm
✓ AC+DC RMS	1.35 V
✓ AC+DC dBV	2.60 dBV
✓ AC+DC dBm	4.81 dBm
Timing:	
✓ Duty Cycle	39.4 %
✓ Positive Width	0.894 ms
✓ Negative Width	1.38 ms
✓ Rise Time	0.420 ms
✓ Fall Time	1.17 ms
✓ Period	2.27 ms
✓ Frequency	0.440 kHz
✓ Phase	91.3 deg

Illustration 5: Sine with one harmonic



Amplitude:	✓ CH1
✓ DC Mean	0.00 V
✓ Max	2.19 V
✓ Min	-2.16 V
✓ Peak-to-Peak	4.34 V
✓ High	0.53 V
✓ Low	-0.59 V
✓ Amplitude	1.13 V
✓ AC RMS	1.08 V
✓ AC dBV	0.670 dBV
✓ AC dBm	2.89 dBm
✓ AC+DC RMS	1.08 V
✓ AC+DC dBV	0.670 dBV
✓ AC+DC dBm	2.89 dBm
Timing:	
✓ Duty Cycle	50.0 %
✓ Positive Width	1.13 ms
✓ Negative Width	1.14 ms
✓ Rise Time	0.280 ms
✓ Fall Time	0.268 ms
✓ Period	2.27 ms
✓ Frequency	0.440 kHz
✓ Phase	36.8 deg

Illustration 4: Sine with two harmonics

Frequency Modulation Synthesis

Frequency modulation, or FM, synthesis arose as an alternative to additive and subtractive synthesis techniques. Developed by John Chowning at Stanford in the 1970's, FM synthesis resolved many issues of efficiency that previous methods could not. Additive synthesis, while transparent and flexible, incurs a significant overhead as wave forms grow increasingly complex. For each additional harmonic added to the fundamental wave, a separate oscillator must be utilized. As such, creating arbitrarily complex wave forms can get quite expensive in terms of computations and memory.

The FM equation

$$f(t)=a(t)\cdot\sin(2\pi\cdot f_c+i(t)\cdot\sin(2\pi\cdot f_m))$$

appears to be quite obfuscated as compared to the simplicity of the additive synthesis, though, in no way lacking elegance. Essentially, one can interpret the equation as a sinusoidal wave containing a rather interesting phase function. The carrier frequency, f_c , is the fundamental frequency which is then altered by another sinusoidal function oscillating at a modulating frequency, f_m .

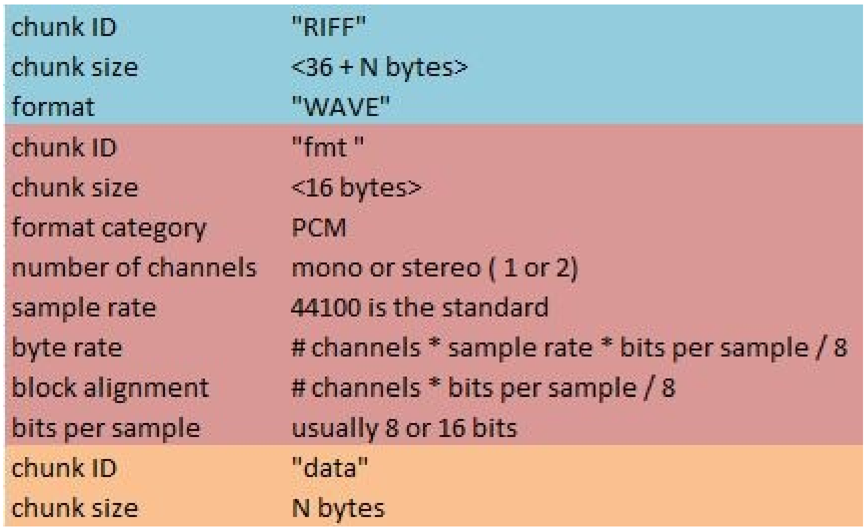
Using several trigonometric properties, it can be shown that the FM equation may be represented as an infinite sum of sinusoidal functions at varying amplitudes and frequencies. These waves, known as *sidebands*, are clustered about the carrier frequency. The width and intensity of the sidebands are determined by the modulation index, $i(t)$, and the modulating frequency. Oftentimes it is easier to express the modulating frequency in terms of the carrier frequency. This ratio, called the *harmonicity*, is defined as

$$H=\frac{f_m}{f_c}, H\geq 1$$

So, given these properties which the FM synthesis technique exhibits, acoustic instruments can be modeled with reasonable accuracy without sacrificing too much computational power. As an aside, it is important to note that it is not simply the timbre of the wave which lends realism to a synthesized wave form, but also its *attack, sustain, and decay* behavior. The attack is the manner by which the wave begins, sustain is how it acts for the duration of the wave, and the decay describes how the wave ends. These parameters may be emulated by varying the amplitude and modulation index of the equation over time. Chowning has written a paper which illustrates several attack, sustain, and decay envelopes for an FM synthesizer which attempt to mimic acoustic instruments.

WAVE File Format

WAVE, or Waveform Audio File Format, is an uncompressed audio format developed by Microsoft in conjunction with IBM. WAVE adheres to a specification known as RIFF, or Resource Interchange File Format. This format is a tag-based format, meaning that each chunk of data written into the file is preceded by a label describing the role and size of the data. Data in a simple WAVE file would be structured as so:



chunk ID	"RIFF"
chunk size	<36 + N bytes>
format	"WAVE"
chunk ID	"fmt "
chunk size	<16 bytes>
format category	PCM
number of channels	mono or stereo (1 or 2)
sample rate	44100 is the standard
byte rate	# channels * sample rate * bits per sample / 8
block alignment	# channels * bits per sample / 8
bits per sample	usually 8 or 16 bits
chunk ID	"data"
chunk size	N bytes

The first chunk, in blue, designates the file format and the data size in bytes. The red chunk contains specifications for this particular WAVE file. The final chunk, in orange, contains all the actual wave form data. You can see that in each chunk, there is an ID and a size field, which aligns with the RIFF specification.

While there are a fair few formats which compress audio files to a reasonable size, WAVE is certainly one of the more straightforward formats to implement. Additionally, there is no loss in resolution between the raw samples and the saved data in an uncompressed format such as WAVE. The trade-off is, however, size – WAVE files may easily reach several hundred megabytes of data for sufficiently long audio files.

JACK Audio Connection Kit

JACK is not so much a general audio synthesis concept as it is a tool by which audio production may be facilitated. I discovered JACK while searching for a means by which I could interact with the audio hardware in Linux. While ALSA, the Advanced Linux Sound

Architecture, is the low level interface between the actual PC hardware and the Linux system, JACK is an attractive alternative to directly utilizing ALSA libraries in that the framework, as it abstracts away any OS-specific audio implementations. As such, JACK is (conceivably) portable across platforms. Additionally, the JACK programmers designed the toolkit to be a client-server model. So, your application, the client, hooks into an active JACK server, which is either running locally or on a distributed network. It is then the task of the server to schedule client access to the underlying audio hardware.

I use JACK in several of my small test projects as well as my large final project, a sequencer. I shall discuss this project and how it interacts with JACK later on in this paper.

Projects

Here I shall discuss some of the projects and activities I pursued to gain a better understanding of the various concepts I addressed in the course. Throughout the semester, I wrote a fair number of small, isolated programs which tested different portions of the topics. These programs were short, and often took less than five or so hours to implement properly. However, two projects I approached this semester were much larger in scope. These were meant to be amalgamations of the work approached in the smaller programs.

8-Bit Wave Form Generator

The first large project I worked on was the 8-bit wave form generator. This project utilized an AVR microcontroller and a simple DAC and amplifier circuit to output an 8-bit waveform. The resulting product was capable of producing a sine, square, triangle, or rising/falling sawtooth pattern. The generator could also output a sine wave with additional harmonics, constructed using additive synthesis. The microcontroller output waves in parallel, using eight digital out pins. These pins flowed into an R-2R digital-to-analog converter constructed of several 10kΩ and 20kΩ resistors. The R-2R DAC converted the parallel signals into a single analog signal, which was then amplified using an op amp circuit. The final output was then sent through a small 8Ω speaker for the user's listening pleasure.

The software on the microcontroller stored the waveform as a wavetable of 256 1-byte samples, ranging from 0 to 255. While the resolution is quite low, the oscilloscope used to capture the output wave form revealed a reasonably smooth result. However, upon auditory observation one may hear the distinct “roughness” of the low resolution samples. Samples were output using timer interrupts; the frequency of the interrupt was set such that the entire wave table would be output at the appropriate rate, calculated by multiplying the number of samples in the table by the frequency of the waveform.

This project, while sufficient in its introduction to low-level wave form production techniques, possessed several shortcomings in the end. Firstly, I did not write a way to dynamically exchange waveform types. Providing button functionality would probably have been ideal – perhaps it shall a future feature to implement. The microcontroller clock frequency also caused some issues. The AVR microcontroller I used (a rather inexpensive model) had only a 16MHz clock. Outputting a high frequency wave form using interrupts becomes impractical, especially if there was processing code inside the main loop of the program.

Pictured below is the microcontroller and DAC circuitry. Note that yes, the speaker (essentially a magnet) is resting on what would be the hard drive of the computer. Rest assured, however, the computer was already dead.

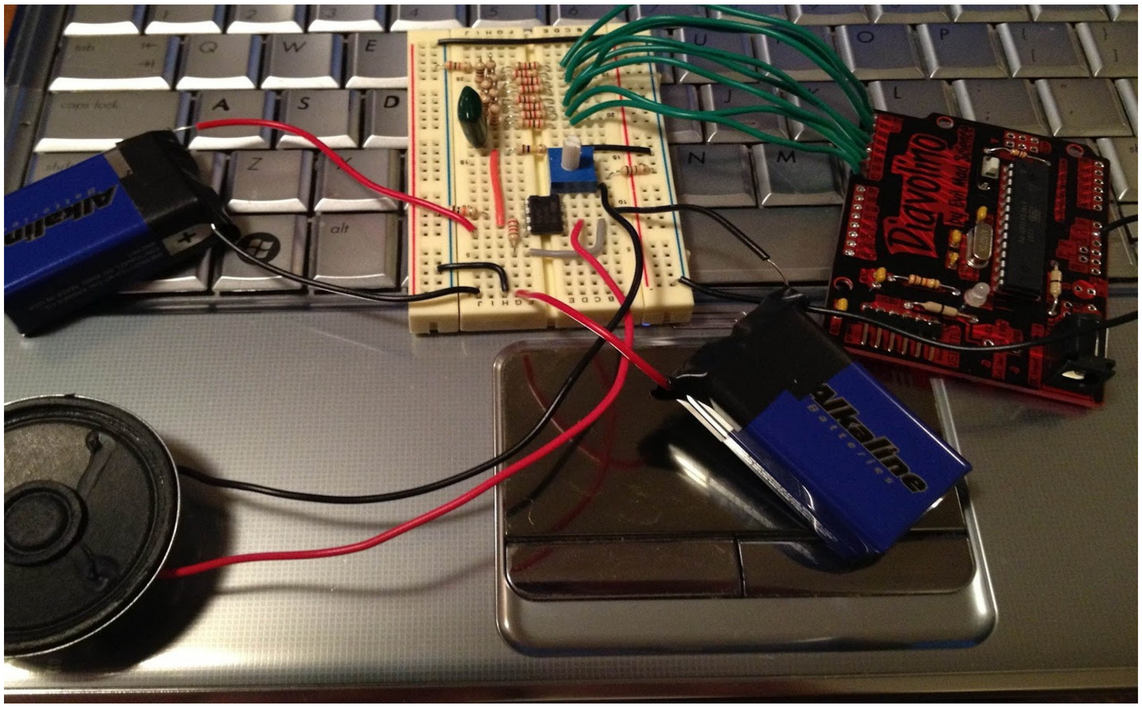


Illustration 6: AVR Microcontroller and DAC circuit

Code for the microcontroller can be found on the course Github. See the Appendix for a URL.

Sequencer

The final project I approached this semester attempted to combine most or all of the course topics I researched. I believe that, since I ultimately wish to apply the knowledge I have and will attain with regards to digital audio synthesis to the production of live digital music performance software (see Ableton Live, Max/MSP), an appropriate introduction would be to create a simple sequencer. The sequencer features polyphony (multiple notes at once), the ability to swap wave parameters on the fly, and to save the the wave form as a WAVE file.

The sequencer hooks into JACK as a client program. Essentially, the process is as such: the sequencer program asks a running JACK server to register a client name. Then, it requests a port for the client. It is through this port that the sequencer client is registered. The sequencer then proceeds to assign a callback function to itself through which the JACK server requests audio data. Every time the user selects 'play' in the sequencer GUI, the sequencer tells the JACK server that it is ready to begin processing, and the JACK server then adds the callback function to its handling queue.

The GUI of the sequencer is created using Qt4, though care was taken to make sure that the logic behind the sequencer remained framework agnostic. Pictured below are several screenshots from the sequencer application. Source code for the app is available on the course Github.

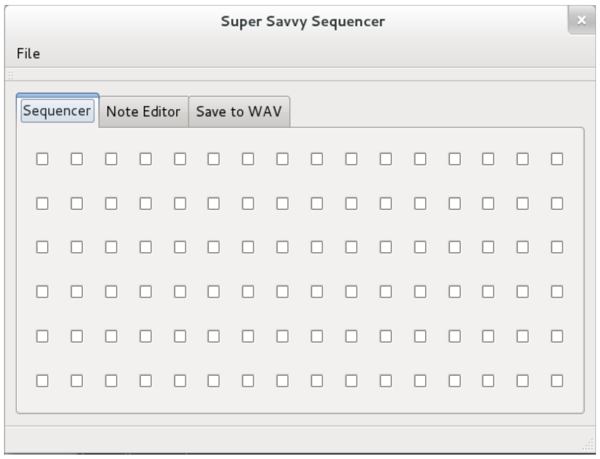
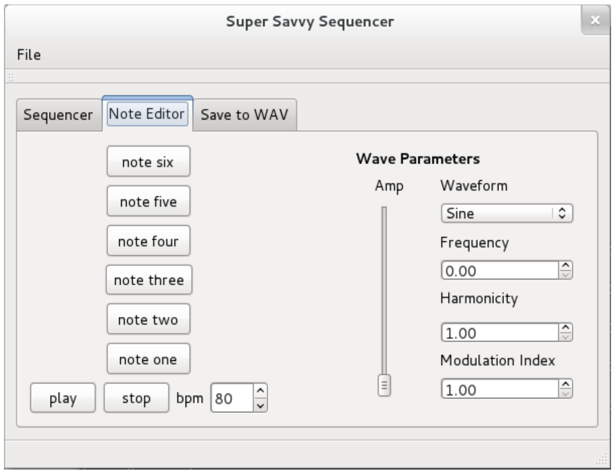


Illustration 8: The main sequencer UI



bpm 80

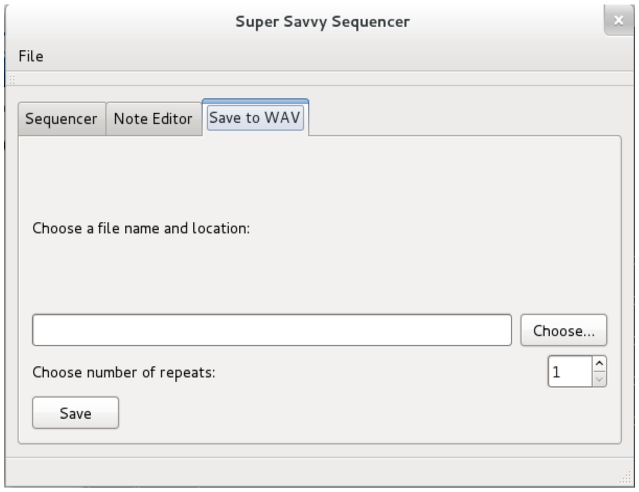
Waveform Sine

Frequency 0.00

Harmonicity 1.00

Modulation Index 1.00

Illustration 7: Note and waveform parameter editing



Choose a file name and location:

Choose number of repeats: 1

Illustration 9: WAVE file save options

Reflections

This independent study has been an enlightening and inspiring foray into digital audio synthesis and techniques. This course, however, did not act solely as a introductory endeavor in digital audio; it also has shown me, though dead ends and trial-and-error, the amount of effort involved in discovering on one's own the appropriate course resources, relevant and irrelevant topics, and feasible yet meaningful projects. Upon unearthing a seemingly valuable resource, I was often hit with “rabbit hole syndrome”, in which I would continuously delve deeper into technical references made by the texts, technical references made by prior technical references, and so forth. While having these nearly endless references and resources at my disposal was fantastic, it was of little actual benefit to my efficiently accruing a more general understanding of the topic.

My next steps will ideally be to join the Audio Engineering Society. The membership provides access to an excellent journal covering advanced and contemporary topics in audio engineering. These topics may be of value as I progress to implementing more powerful sound synthesis techniques and algorithms. Additionally, there are several open source projects for digital music which would be of value for me to contribute. The Linux MultiMedia Studio project seeks to be a viable alternative to the commercial software Ableton Live. While it still has a fair few areas in which to catch up, many of the plugins the LMMS supports is also supported in the major commercial software packages. So, writing plugins for this project will doubtlessly benefit me when working with the commercial analogs. Ultimately, I do believe that through this independent study I have achieved my intention of building a solid foundation of audio synthesis concepts with which I may expand in the coming months and years.

Appendix

Course Materials

Blog:
http://eightbitheart.blogspot.com/

Github:
https://github.com/rokthewok/ECE4974

Course Resources

The Audio Programming Book, Richard Boulanger, Victor Lazzarini, Max Mathews

The Theory and Technique of Electronic Music, Miller Puckette
http://crca.ucsd.edu/~msp/techniques.htm

Sound Synthesis Theory, Wikibook
http://en.wikibooks.org/wiki/Sound_Synthesis_Theory

Frequency Modulation Mathematics
http://cnx.org/content/m15482/latest/

JACK Audio Connection Kit
http://jackaudio.org/

Ableton Live
https://www.ableton.com/

Max/MSP
http://cycling74.com/