CACOPHONY

A Capstone Project

by

Jake Buzzell

Submitted to the Faculty of the

Department of Software Technology

in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE

in

COMPUTER SCIENCE and INNOVATION

Champlain College

2020

ABSTRACT

In the age of computer music, there are no hardware devices that will algorithmically create new melodies given a melody defined by the user. A digital hardware device known as Cacophony was designed to meet this need. Cacophony serves to tighten the link between computer music and human performance by allowing musicians to record various notes as they play, which will in turn be used by the device to create new melodies by adding one additional harmonically relevant note at a time. Cacophony is written in C++ for the Teensy 4.0 Arduino-based microcontroller. Melodies created by Cacophony's note generation algorithm were generally accepted to be similar in emotion with greater complexity than melodies recorded into the device; these melodies were described by the majority of testers to sound human-generated rather than computer-generated.

Table of Contents

List of Figures

CHAPTER 1: INTRODUCTION

Problem Statement

Effects modules for the guitar, commonly referred to as effect pedals, offer control over an instrument's sound, but are typically unable to create melodies and accompaniment autonomously given a set of limitations.

Purpose Statement

The goal of this project is to create an effect pedal that uses the guitarist's signal to algorithmically generate unique scales and melodies (a la the modular synthesizer) and send them to various devices over MIDI.

Context

This project is highly influenced by the design principles of the modular synthesizer. The modular synthesizer, or more simply the modular, is a synthesizer system entirely based on individual *modules,* which each serve a different role in the system. Some modules create sounds from scratch, whether they generate a tone or play a recorded sample. Others modify sounds, such as filters which remove certain frequency bands from sounds. Finally, control modules exist that create voltages by which other modules can be controlled (LeoMakes, 2019). Modules can be selected and interacted with by the user, and are connected with patch cables that route a given signal from one module to another. In this sense, the modular synthesizer is similar to effects pedals in that one can pick and choose what to include in their system.

The synthesizer came to be in the 1960s, with electronic engineers and synth pioneers Don Buchla and Dr. Robert Moog. Moog's synthesizer designs—the first of which being the Moog Modular Synthesizer in 1964—were intended for use with an orchestra, and featured a full bed of piano-like keys (120years, 2014). Moog's approach to synthesizers as traditional musical instruments became known as *east-coast synthesis*, the counterpart to the more avant-garde *west-coast synthesis* championed by Don Buchla.

Buchla intended to create synthesizers explicitly for the creation of experimental music. His machines did not feature a traditional key bed, and relied on complex *wave-shaping*, a form of *additive synthesis*. Buchla's machines were particularly complex compared to synthesizers being developed on the east coast, and as such a large amount of the nomenclature and development standards for them have by the more simplistic language of east-coast synthesis. However, they have influenced a great deal of musicians and engineers, most notably Moog himself, who said of Buchla, "For the past four decades, Buchla instruments have consistently set the standards for innovative musicians… the panel layouts are nothing short of elegant, while the underlying functions are the most advanced and musically rich available today" (Buchla USA, 2019).

This project is heavily predicated on the idea of *generative modular synthesis*. Generative modular synthesis, or more simply generative modular, is an ideology of musical composition that relies on modular synthesizers to create music. This concept has roots in a genre of early electronic music known as *computer music*—a genre that developed from the use of early computers to algorithmically create music (Britannica, 2015). Generative modular is typically accomplished by setting a number of control modules in tandem such that the sound generation

portions of the system will “play themselves”—an illusion created by intelligent systems that can generate sequences, modulate the parameters of sound generation, and perform typical computer operations such as logic functions and random number generation. The player can then use these controls to set constraints within the system will generate musical phrases and ideas that (ideally) morph over time (Mylarmelodies, 2016).

Significance of Project

This project will draw inspiration from computer-generated music to aid the guitarist with composition, accompaniment, and sound creation. While modular synths are not inherently incompatible with guitar, they are incredibly costly. The creation of an open-source project that applies certain facets of the composition styles afforded by modular synthesizers to the guitar will be valuable for personal musical purposes while also potentially providing inspiration for professional guitarists and pedal designers.

CHAPTER 2: LITERATURE

Literature Overview

The research necessary for this project is partially in the realm of electronics, but also in mathematics and physics. A fundamental understanding of how hardware works is essential for the creation of this project, and furthermore an understanding of the mathematics behind music (specifically pitch) is necessary to develop an algorithm that creates music from scratch. Additionally, the project's inspirations are crucial to its context both as a device and as a passion project.

This project would not exist if not for the contributions of synth pioneer Bob Moog to the music world. *Moog: A History in Recordings* by Thom Holmes (2016) provides a timeline of Moog and his company's ever-lasting impact on music. Having an understanding and appreciation for the history of synth hardware is crucial to the introduction of new hardware to the market. Moog's impact on music hardware is undeniable and must be credited in such a situation—the culture derived from machines made by Moog and his contemporaries (e.g., Roland, Sequential Circuits) has provided great inspiration for this project.

Music played by humans has long been argued to be "robotic" or simply inferior to music generated by computers. Roberto Bresin and Anders Friberg's *Emotional Coloring of Computer-Controlled Music Performances* (2000) attempts to quantify the difference between these two methods of music creation. Bresin and Friberg outline a number of different emotions and how they are quantified by an outside observer, and compare the emotional responses towards each method of music creation. This project aims to programmatically create music that is both beautiful and human. Using the parameters set forth by Bresin and Friberg (2000), one can

quantify how "human" the music created by an algorithm is (and if that level of humanity is consistent).

As this project involves the creation of a melody generation algorithm, having an idea of how other melody-generation algorithms work is greatly beneficial. Z.W. Geem's *Music-Inspired Harmony Search Algorithm: Theory and Applications* (2009) provides an in-depth look at the algorithmic creation of harmonies. Geem's (2009) algorithms are referred to as "meta-heuristic"–defined as using higher-level techniques to find the most optimal value. Geem (2009) includes examples of mathematical algorithms that were tested as well as corresponding pseudo-code for each example. The algorithms are not intended for the creation of the "perfect" harmony, but rather the most optimal harmonies with room for randomization. The purpose of Geem's (2009) research into harmony algorithms is not for the purpose of programmatically creating music; the applications are more geared towards using harmony as the inspiration for optimization problems. Though the applications of Geem's (2009) research are at odds with its relevance to this project, the algorithms developed will provide a great starting point for creating a unique melody generation algorithm.

A key element of understanding how to create scales algorithmically is understanding the mathematical relationships that musical tones have to one another. A series of webpages entitled *Harmony* from Georgia State University's HyperPhysics (n.d.) department outline the mathematical relationships that different notes in an interval have from one another (i.e., a perfect fifth has a 3:2 ratio between its fundamentals). The informational content of *Harmony (n.d.)* is fairly basic, but provides deep insight into the generation of scales. This project will be determining scales from raw pitch information whose melodic content will be read out in Hertz

(Hz). Knowing the math that goes into scale building will allow for less overhead on conversions from pitch to MIDI note–the conversion will only have to be performed once, after the pitches in the scale have been generated.

As such, converting from MIDI note number to fundamental frequency is highly important, and a page from Michigan Technical University's Physics department entitled *Formula For Frequency Table* (n.d.) provides a formula for calculation of pitch for each note in an equal tempered scale based on MIDI note numbers. MTU (n.d.) provides a formula for the determination of the pitch of each note on a standard 88-key piano, though the formula could be used for higher or lower notes above the range of human hearing. While this project's algorithm will not rely entirely on the relationship between note names and fundamental frequencies, being able to convert between the two will be valuable for testing the algorithm. Given a scale output in frequencies, a programmer might not be able to readily determine the viability of the scale. An output of notes, however, can easily be played back on an instrument to determine whether or not the scale is usable. Furthermore, a formula such as MTU's (n.d.) could be used to make sure that notes created by the algorithm are valid notes in the equal tempered scale.

Creating hardware, which will be a significant portion of this project's development cycle, can be daunting for novices. *Handmade Electronic Music: The Art of Hardware Hacking* by Nicolas Collins (2014) provides excellent info on creating music hardware from the perspective of a musician rather than an engineer. Collins (2014) provides advice and example projects to get musicians started creating musical devices with a flair for DIY. Though Collins' (2014) projects can be fairly basic, they are an excellent starting point for someone with a music background looking to begin creating their own instruments.

Similar to *Handmade Electronic Music* (2014) is a blog post titled *Homemade Guitar Looper with Arduino* by Blogspot user elboulangero (2009) on the creation of a guitar looper pedal with an Arduino. The looper is relatively low-spec, as Arduino is a relatively limited platform in terms of memory. However, eboulangero's (2009) findings contain schematics and a codebase, which will be useful as a resource for interacting with and processing audio on Arduino, the hardware platform on which this project will run.

An important part of any hardware project is choosing a platform. Welch, Wright, Morrow, and Etter's *Teaching Hardware-Based DSP: Theory to Practice* (2002) is meant for instructors to aid in the selection of hardware and software for the teaching of digital signal processing. Though the text is from 2002, much of the software and concepts mentioned in the article are still relevant to this day (e.g., MATLab, C). Despite the fact that the text is meant for instructors, it provides inspiration for this project in terms of examples of relevant platforms and projects to try as a warm-up before delving into the hardware development process.

A unique opportunity afforded by hardware is its ability to influence software–in this project's case, the melody creation algorithm. In a blog post by Emmett Corman titled *Simple Synthesis: Part 11, Sample and Hold* (2017), Corman describes how a sample and hold wave is used in modular synthesis. A sample and hold wave is a waveform that provides pseudo-random voltages over time, which can be used in a number of ways in this project. Aanalog random voltage generators are less stringent than digital random number generators, which could add to the "humanity" of the generated melodies.

The software chosen to prototype and/or run hardware projects can also have a great impact on the project's success. *Pure Data: Another Integrated Computer Music Environment* by

Miller Puckette is a 1996 document on then-upcoming visual programming language Pure Data. Pure Data (or Pd) is a language designed in competition with Max, a common platform for digital synthesizers and effects. Though Puckette's (1996) paper is twenty-three years old, Pd's design philosophy has remained consistent throughout its life cycle; in addition to this, *Pure Data* (1996) is an overview of the language's potential rather than a piece of documentation, so its age is less problematic. Pd's objects are written in C (or optionally C++), meaning that any algorithms or code written for this project can be easily ported to Pd for prototyping.

Hardware

Two related pieces of hardware are also major inspirations for the project. Chase Bliss Audio's *Thermae* analog delay pedal provides sequenced digital control over the delay time of the effect, creating harmonically relevant melodies to the input signal. The melodies that *Thermae* generates are the same as whatever notes were played into the pedal, but sped up so as to appear as melodies a fourth, fith, octave, and more above or below the input signal ("Thermae", 2018).

Another related piece of hardware is the Music Thing Modular *Turing Machine*. The Turing Machine creates no sound of its own. Instead, it creates stepped random voltages based on a tempo. Voltages from the Turing Machine can be sent to other synth modules via a patch cable; the voltages are then used to create melodies and modulate parameters throughout the system. When the user is satisfied with a random sequence, they can lock the sequence in place so that it plays back on repeat. The Turing Machine's random generation is completely random, with no basis on an input pitch (Music Thing Modular, n.d.).

CHAPTER 3: METHODS

Design

The design spec of this project is such that the final product will be a device similar to a modular synthesizer's control modules in a guitar pedal format. It will be a physical device that fits into a standard guitar pedal enclosure—specifically, the Hammond 1590DD. The layout of the interface (as of December 2019) is as follows:

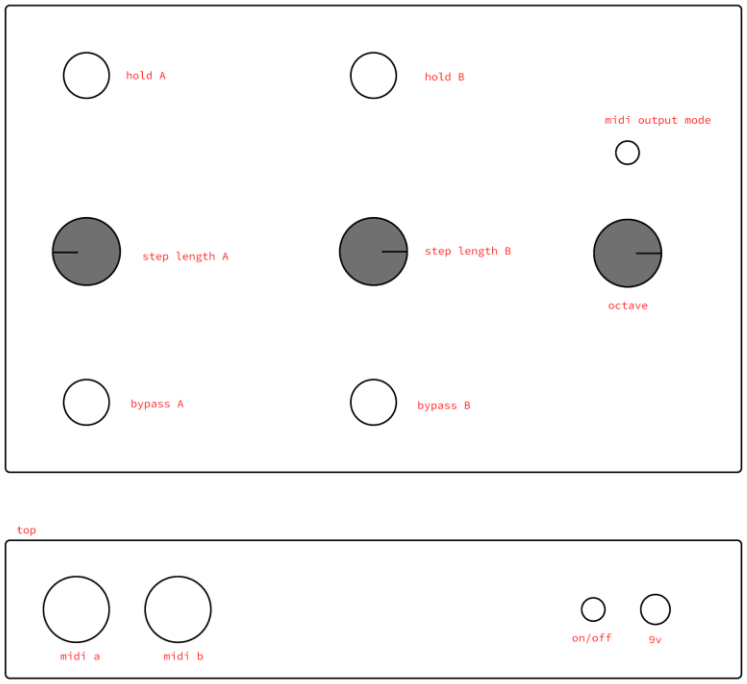

Fig. 3.1: Prototype of enclosure and user interface

The design philosophy of this interface is that everything is large and spaced out properly. Footswitches, such as those used for bypass and hold, need to be spaced far apart from one another such that someone in a performance situation can activate each switch without worry of

accidentally pressing the wrong switch. Knobs controlling constraints for melody playback are all in the middle of the device or near another relevant control. MIDI jacks are aligned with respect to each control lane.

Frameworks

This project will run on Arduino (more specifically Teensy, a platform based on Arduino), and C++. Arduino is the most popular microcontroller on the market, making it a good choice for a novice in hardware development. There are a great deal of resources on the internet for Arduino such as blogs, forums like StackOverflow, and great documentation. Running C++ on the Arduino is necessary over C, as C++ has a bigger standard library, meaning that a lot of the distractions (e.g., the creation of data structures like linked lists) of building an algorithm for melody generation are taken care of immediately.

Algorithms

The majority of the project's codebase will contain or otherwise support the melody generation algorithm. The algorithm takes two input parameters, an array of the notes that the scale is to be based on, and the desired size of the output scale. The algorithm then determines the number of remaining notes available in the octave. For example, if the notes A5, C#5, and D5 are given to the algorithm, it will determine that A#5, B5, C5, D#5, and so on until A#6 (exclusive) are available for use in the scale. The algorithm will then test each available note, finding the average *harmony coefficient*—a comparison of two notes based on the ratio between their fundamental frequencies—of the given note and the current scale. The algorithm will then

determine which note has the best average harmony coefficient, add it to the scale, and then start the process again using those notes. It will continue this process until the scale reaches the desired size. Before being returned, the scale will be checked as to whether or not it is valid (i.e., a standard heptatonic scale ought to have one of each named note, be it sharp or flat). The algorithm in its current state (as of December 2019) can be represented in pseudocode as such:

```
Note[] generateScale(Note[] basis, int size){

     // we can assume that xorScale finds which notes
     // are not in any given scale

     Note[] missing = xorScale(basis)
     Note[] scale = basis

     while (scale.size != size){

          // gather all scores from missing notes
          dict<Note, int> scores = {}

          for note in missing{

               // determine how well note fits in scale
               // and store the result
               dict[note] = getAvgHarmony(note, scale)

          }

          // determineBestScore() gets best harmony
          // score and returns corresponding Note
          scale.append(determineBestScore(scores))

          // validate scale--check if all note names
          // are accounted for and removes offenders
          if(scale.size == size){
               validateScale(scale)
          }
     }
     return scale
}
```

Analytical Methods

As this project is designed to create music, analyzing its output will be difficult. From an objective standpoint, the project can be judged in the following manner:

1. Does the device create randomly generated melodies?
2. Are those melodies created from notes sampled from the user's playing?
3. If given constraints (e.g., octave range, step length, repeated sequences), does the device follow those constraints?

If the device checks all of these boxes, then it can be considered to be a working product. However, a product that works is not necessarily a product worth using. Given that this project is designed to create music, analysis of the music it creates is a more personal, philosophical question than it is a Boolean *yes* or *no*.

To call into question the intrinsic worth of any musical work is to invoke an ontological and epistemological discussion of what music is and how we understand it. A common fundamentalist ontology of music as outlined in the Stanford Encyclopedia of Philosophy's *The Philosophy of Music* (2017) is a *performance theory*—that works of music can be understood as actions. In Aitor Izagirre and Igor Petralanda's *On the Idea of Action in Davies' Performance Theory of Art* (n.d.), Izagirre and Petralanda explain this to mean that a work of music is entirely abstract until it is performed, whereupon the choices made by the performer (intentional or not) define it as a concrete work.

The Philosophy of Music (2017) also grants that in most theories of determination of the value of a given musical work, emotional response plays a large role. Tying into performance theory, this means that the emotional response created by certain actions taken by a musician are

the basis for its worth. In her landmark work, *Elements of Expression in Music*, Kate Hevner (1936) developed what is known as the *adjective circle*, a non-comprehensive list of adjectives that can be used to describe the emotional response that a given listener might have upon hearing a musical work. This adjective circle can, for the purposes of this project, serve as a way to quantify emotional response to music.

Using performance theory as our basis for what musical works are and emotional response in order to judge the worth of the music created by this project, what an *action* is in regards to music created by the device itself must be defined. Within the parameters set for a given performance, the device can make choices on the following:

- The scale to be played in
- What note to play when
- The velocity of the given note

As these three points are the only influence the device itself has over the music being played, these can be determined to be the actions it can take during any given performance. These actions must be judged in the context of a given performance. In an improvisational performance, the music created by the device can be judged on whether or not it evokes any kind of emotional response. In a performance of a song, the music created by the device can be judged on the emotional response created by the improvised playing in tandem with a structured performance.

Features

The core features of the project are as follows:

- The device will receive an input signal and output it unchanged.
- The device will read and analyze the input signal for the root frequency of the note/chord being played.
- The device will build an equal-tempered melodic scale based on the notes gathered from the input signal.
- The device will send random notes within that scale via MIDI based on a tempo set by the user.
- The user may trigger the device such that one MIDI channel plays the last x notes again, where x is the step length (of a range from 1-16) set by the user.

These features are essential for the product to function as a whole. The following are stretch goals to be implemented should there be time:

- The user may select a MIDI output mode such that MIDI channels A and B play:
    - the same notes in parallel (and can be bypassed independently).
    - different notes in parallel (and can be bypassed independently).
    - the same notes, but only one channel selected by the user plays at a time.
    - the same notes and alternate MIDI output from A to B per step such that output of each channel is mutually exclusive—if one output is playing a note, the other is silent.
- The user may trigger the device such that *both* MIDI channels play the last x_A or x_B notes again, where x_A and x_B are the step length (of a range from 1-16) set by the user for channels A and B, respectively.
- The device will play a third melody on a built-in synthesizer circuit.

- The device will use a sample and hold circuit to influence externally what notes to play and when.
- The device will output its pitch data in 1 volt per octave-tuned modular synthesizer control voltage.

Test Plan

The melody generation algorithm will be tested both mathematically and qualitatively. The music produced by this device will be judged using the performance theory-based paradigm for judging the worth of a piece of music discussed in the *Analytical Methods* section of this chapter, which will be analyzed in correspondence with mathematical tests. To test the mathematical validity of a scale, the device will determine and output to a console the average harmony coefficient of the scale. Lower average harmony coefficients indicate a more harmonious scale. Scales will then be evaluated by ear to determine if they are musically viable. A developer will take notes using Hevner's (1939) adjective scale on what kind of emotions a given scale evokes, afterwards marking each adjective scale with its harmony coefficient.

Criteria and Constraints

One major consideration when creating a device that algorithmically generates music is to make sure that the rules of the algorithm are not too hard and fast. This is a large part of the decision for the device to play melodies randomly as opposed to another alternative such as generating them with machine learning—if a melody-making algorithm is too targeted, too specific, it runs the risk of creating existing, copyrighted melodies. If this were to happen, the

project would run the risk of being taken down from websites like GitHub, YouTube, or other social media via a copyright claim.

CHAPTER 4: RESULTS

Final User Interface

The final user interface of Cacophony deviated largely from the initial design for reasons outlined in Chapter 5. Rather than using a permanent metal enclosure, the project is laid out on a breadboard as shown below:

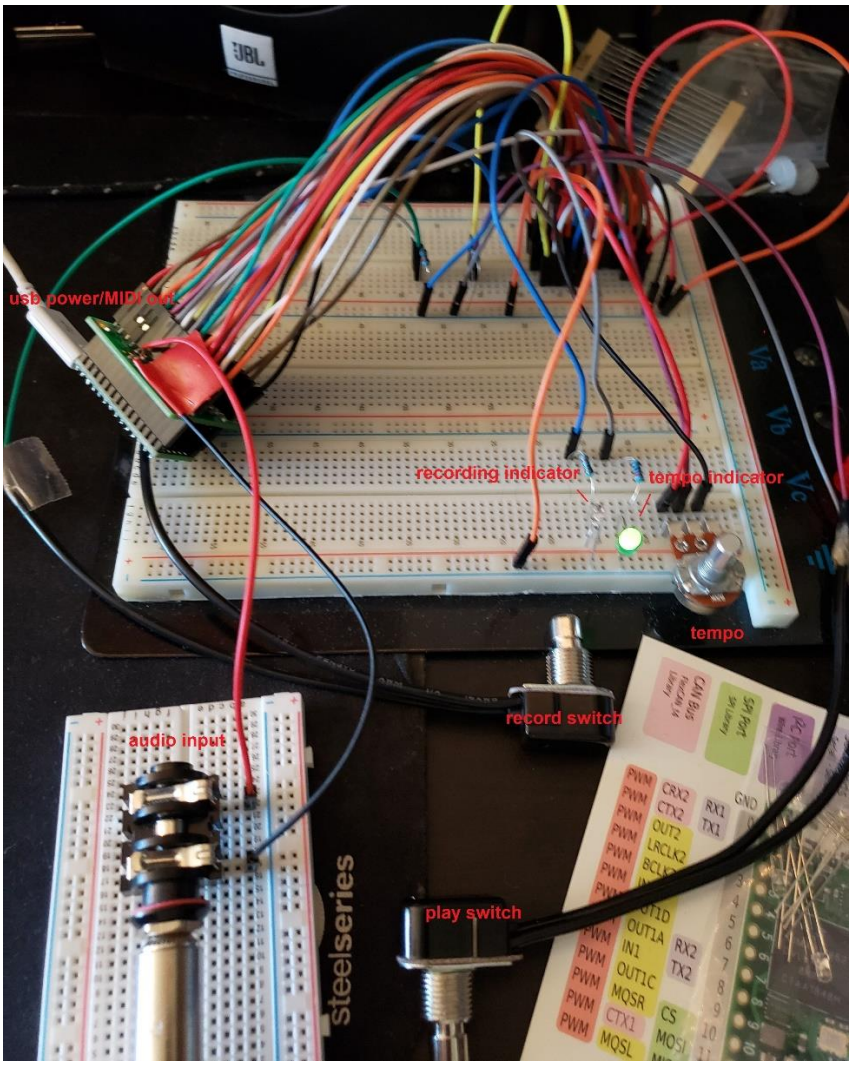

Fig. 4.1: Cacophony final layout and UI diagram

The main controls of Cacophony are the record and play switches, the tempo selector and indicator, and the recording indicator. Cacophony's audio input accepts high and low impedance

signals (with no perceptible loss of quality for high impedance signals) via its quarter-inch instrument cable audio in jack, and outputs MIDI signal via USB.

The device works by allowing the user to input new notes while they hold the record button. The user can play as many notes as they like while holding the record button. Each note played while holding the record button will be recorded by the device and placed into persistent storage. If the user plays a note more than once, the device will only store the note one time—even if the user plays the same note across several octaves.

Once the user has recorded notes into the device, double-tapping the record button will trigger the device to algorithmically generate a new note complementary to the already-recorded notes. Generating a new note from a happy, harmonious arpeggio will add a note that makes the sequence more harmonious and musically complex; generating a new note from a dissonant arpeggio will add a note that makes the sequence more dissonant, and so on. The device will also generate a new note at random if the user activates the note generation feature before recording any notes.

The user can play back an arpeggio of the recorded and algorithmically generated notes based on the current tempo setting by holding the play button. The playback tempo can be increased or decreased before, during, or after playback using the tempo knob. The tempo indicator LED will blink at the rate of the current playback tempo at all times (except when the device is in record mode). The core user experience loop can be visualized with the flowchart below:

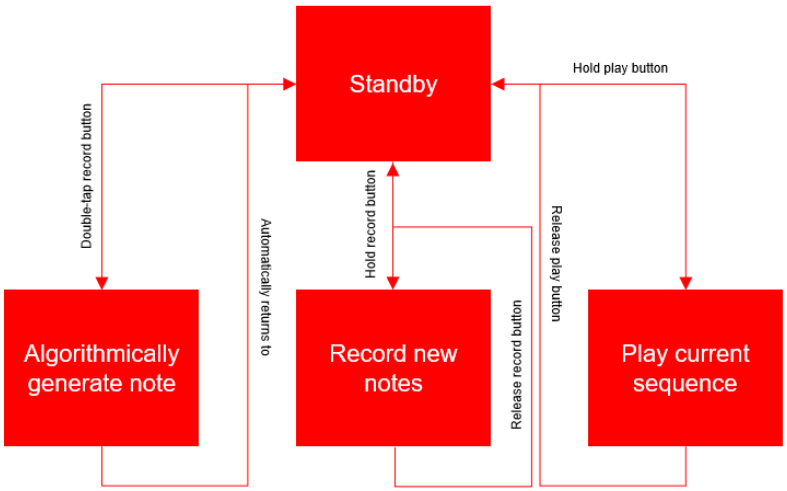

Fig. 4.2: Flowchart of Cacophony user experience loop

Cacophony sends MIDI signals via USB to the user's computer. On Windows, device setup is automatic. Once the device is plugged into the computer, it will install itself as a MIDI device and appear in the system's Device Manager as "Teensy MIDI".

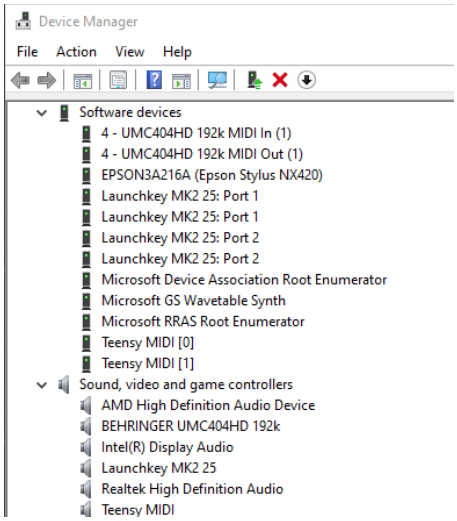

Fig. 4.3: Windows Device Manager displaying Teensy MIDI input and output

Once plugged in, the "Teensy MIDI" device is universally compatible with all audio software with support for MIDI devices.

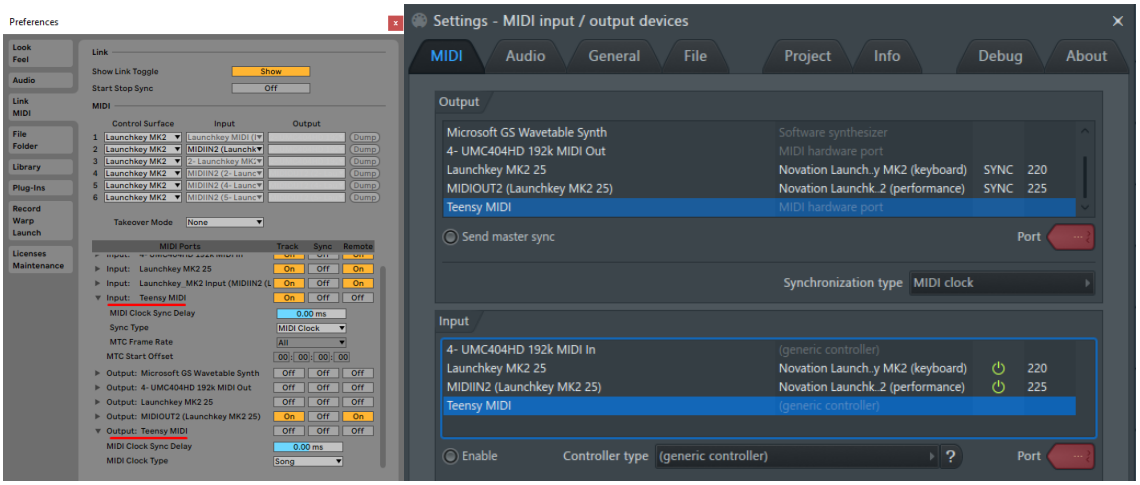

Fig. 4.4: Teensy MIDI device automatically appears in Ableton Live and FL Studio, respectively

Analytical Results

Results from qualitative tests were varied. Testers were asked via a short online survey to listen to a series of melodies before and after the influence of the Cacophony note generation algorithm.

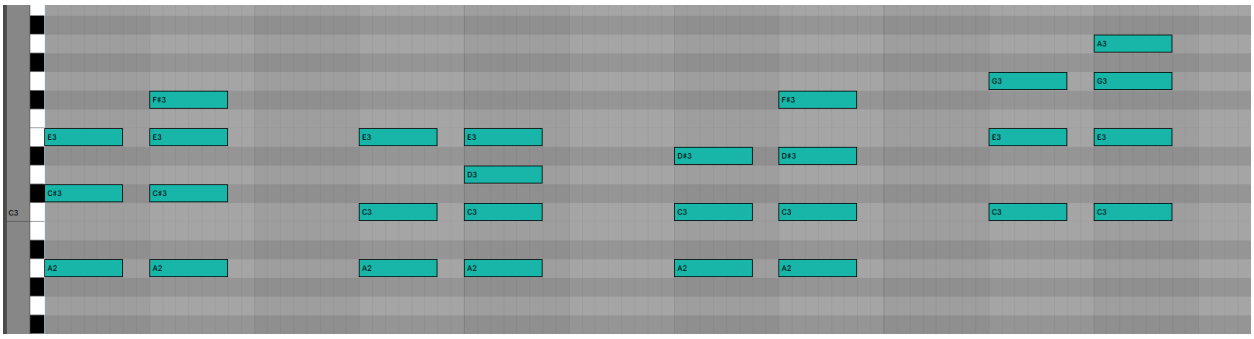

Fig. 4.5: Chords before and after algorithmic influence

After listening, testers were asked to choose one or many categories that most fit each melody (before and after algorithmic influence). Testers chose from a list of adjective categories most commonly used to describe music as demonstrated by Hevner's (1936) *adjective circle*, shown below.

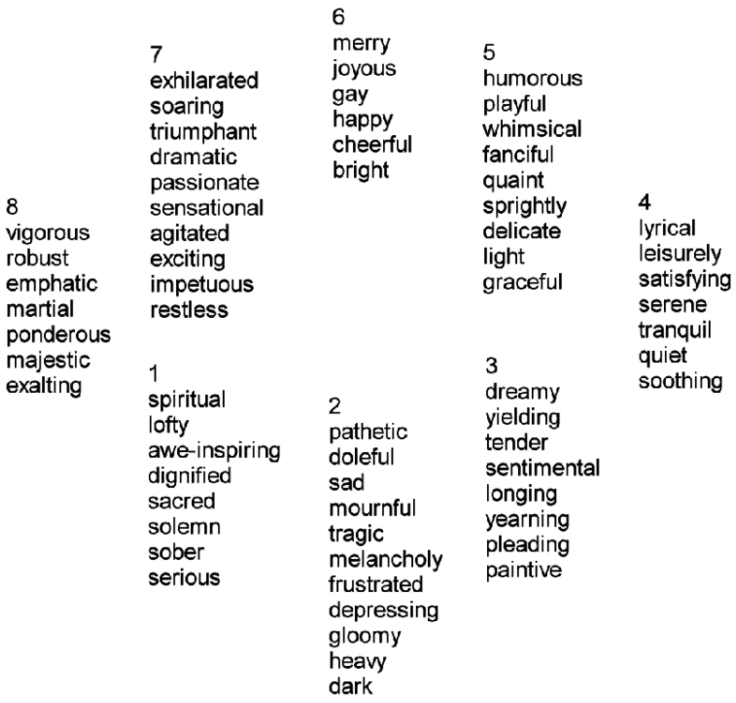

Fig. 4.6: Hevner's (1936) adjective circle

The first chord testers were asked to listen to was an A major chord, which is a traditionally harmonious chord. The major chord is typically thought to be bright, happy, and uplifting. Half of testers responded that categories 5 and 6 described the chord, and one third of testers additionally responded that categories 1, 3, and 4 described the chord. These are all categories with generally positive connotative adjectives, which is to be expected given the chord.

The algorithm, given an A major chord, created an A seventh chord, which is an A major with an added major seventh—two semitones higher than the major chord's fifth, its highest note. The major seventh is a relatively dissonant interval, with a harmony ratio of 15:8. Adding this interval to the chord will add uncertainty and complexity. Two thirds of testers responded that category 7 described the new chord, with one third of testers responding that categories 5, 6, and 8 described the chord. These are similarly positive categories, but the majority response of

category 7 indicates a sense of energy and emotional complexity that the standard major chord lacks.

The second unaffected chord that testers were asked to listen to was an A minor chord. Minor chords are similar to major chords in construction, but use a minor third as opposed to a major third. The minor third is a slightly more dissonant interval than the major third—with dissonance ratios 6:5 and 5:4 respectively—which gives the minor chord its characteristic sadness. Two thirds of testers indicated category 1, half of testers indicated category 3, and one third of testers indicated categories 2, 7, and 8. The minor chord, sounding sad but not dissonant, provided predictable results, as categories 1, 2, and 3 contain negative connotative adjectives, while categories 7 and 8 contain neutral, hopeful connotations.

Given an A minor chord, the note generation algorithm created an A minor add-4 chord. This is a minor chord with an added perfect fourth. The perfect fourth is a very harmonious interval with a 4:3 dissonance ratio. Adding a harmonious interval to a somber-yet-relatively-harmonious chord should serve to bolster the chord's natural characteristics while making it slightly more complex. Roughly eighty percent of testers indicated category 2; additional responses varied widely with half of testers indicating category 8, one third of testers indicating category 3, and one vote each for categories 1, 4, and 7. A majority response of category 2 indicates that testers overwhelmingly thought the chord to be sadder than the minor chord—an unexpected but understandable result. The variation in responses is in line with the hypothesis that the added perfect fourth adds an emotional complexity to the chord.

Finally, testers were asked to listen to an A diminished chord. A diminished chord is a major chord with a flat fifth—otherwise known as the *tritone*. The tritone is the most dissonant

interval in the 12-tone equal temperament tuning system with a dissonance ratio of 45:32. As such, the chord is very grating and generally requires resolution with a more pleasant chord following. Testers' responses for the diminished chord were the most uniform of all tests, with roughly eighty percent of testers indicating category 2, half of testers indicating category 7, and one tester indicating 1. This result is to be expected given how grating this chord can be.

Given an A diminished chord, the note generation algorithm created an A diminished major seventh chord, which is a diminished chord with an additional major seventh. The major seventh, as mentioned previously, is a relatively dissonant interval that should only add chaos to the diminished chord. Two thirds of testers indicated categories 2 and 7, and one third of testers indicated categories 1 and 8. As expected, this chord remains incredibly dissonant and grating, with an added power and anticipation as indicated by responses of categories 1 and 8.

In addition to listening to chords, testers were asked a simple yes-or-no question, "did any of the melodies sound off-key or 'in-human'?" Two thirds of testers responded "no", while the remaining one third responded "yes". As one of the main goals of Cacophony is to generate relatively human, but not perfect melodies, asking this question helps gauge whether or not testers found this to be the case. A one-hundred percent "yes" response would indicate that the algorithm's melodies are too sterile, a one-hundred percent "no" response would indicate that they are too flawed, and a majority "no" response would indicate that the algorithm requires tweaking to reach a more pleasing balance. The majority "yes" response is ideal, though the two-to-one thirds split might indicate that some very minor rebalancing could be helpful.

Overall, though these results are varied, they help to determine whether or not the note generation algorithm produces its intended result, which is to add a note to a set of notes such

that the new interval matches existing intervals' dissonance level. In general, test results described that more harmonious input chords resulted in similarly harmonious—if slightly more complex—chords, while more dissonant chords resulted in similarly or more dissonant chords. These results indicate in large part that the algorithm is working as intended.

Testing Results

Unit testing for Cacophony was performed in Microsoft CppUnitTestFramework, a relatively bare-bones but highly usable C++ testing framework built into Microsoft Visual Studio. Parameterized tests were written for helper functions used in note storage as well as note generation. The test dataset used was based on the MIDI note to frequency conversion chart from Michigan Technical University, *Formula for Frequency Table* (n.d.). Tests indicated that all such functions performed as expected.

CHAPTER 5: CONCLUSION

Limitations

During Cacophony's development, a large series of limitations came up with little warning, which caused the project to change in its implementation rather drastically. Development started with the intention of creating a standalone hardware device that could interface with other hardware-based MIDI devices. Instead, it became a computer peripheral for studio musicians to interface with software-based MIDI devices. While its purpose remains similar, it relies on a computer to run, and does not allow for pass-through of the original input signal. These changes in design make the device more useful for studio musicians to write songs with rather than for musicians to add a layer of unpredictability to their live performances.

These changes are in large part due to the sudden change in access to Burlington maker spaces in light of COVID-19. In the state of Vermont, a "stay at home" order was issued on March 24th in response to an outbreak of a novel coronavirus which had just been determined a pandemic. The order prohibited all but "essential businesses" to close their doors by 5pm on March 25th (Hirschfield, 2020)—maker spaces were not considered essential, and therefore were shut down.

Because of this sudden lack of access to tools provided by maker spaces, such as drill presses and soldering irons, a number of features for Cacophony changed. The most visually noticeable of these changes is the lack of an enclosure for the project. As a drill press was not able to be accessed, creating a proper project enclosure became impossible. This significantly detracts from the polish of the final project. Polish aside, the lack of enclosure is a relative non-

issue during the stay at home order, as the project does not need to be transferred anywhere during development; keeping it on a breadboard is a viable option for the time being.

Another major roadblock that lack of access to tools created was a lack of access to a soldering iron. A number of the available components for the project were solder-mounted rather than board-mounted, meaning they had to be soldered to the project using cables as opposed to being slotted into a prototyping board. The most notable compontent that was only available as a solder-mounted component was Cacophony's audio output jack. Luckily, a single board-mounted audio jack was available for use as an audio input, but without another jack, the device would not be able to pass-through audio signals directly. For this reason, the project became a direct audio-to-MIDI device rather than a standalone guitar pedal.

Theoretically, new parts could have been ordered to replace the solder-mounted parts, but the pandemic also took a toll on shipping times and part availability. This meant that even if new parts were ordered, they likely would not have arrived in time for the device's initial release. Instead of gambling on shipping times, the project was adjusted to work with the parts available. This allowed for development to continue without being contingent on parts that could not be guaranteed to arrive.

Delimitations

A number of issues in the development of Cacophony were borne of development decisions made early on in the development cycle. These issues ranged from challenging and frustrating, to relatively trivial. These issues, in contrast to those mentioned earlier in the chapter, simply required a large amount of research and a careful workaround.

The most notable of these delimitations is the choice to work on the Teensy 4.0 microcontroller. While the microcontroller is exceptionally powerful—with specs beating out microcontrollers more than triple its size—developing for it comes with a number of unexpected challenges. One of these challenges is its documentation, which is lacking in detail and expects that a Teensy developer has experience developing for embedded systems. Some concepts are not explained in documentation at all and require manual parsing of Teensy source code to understand. Other concepts, namely the platform's audio library, are documented, but are not displayed in the same location as the rest of the documentation. Instead, they are scattered throughout the developer's website on various pages not often linked to from the main website.

Another major challenge of working with Teensy is its mixed-bag support for modern C++ features such as STL containers and standard library functions (e.g. std::for_each). The device supports these constructs, but is either buggy in its implementation of them or only supports them in a very limited capacity. For instance, support for std::vector exists, but any vector object must be declared in the Arduino sketch file, and calls to the copy constructor or assignment operator incur a compiler error. Working around this was relatively simple—code relying on std::vector was placed in the Arduino sketch in addition to its own header and implementation files.

As Cacophony is designed as a MIDI output device, choosing a format over which to send MIDI signals was a great consideration. Initially, I had planned on sending output signals via standard 5-pin MIDI cable. As most MIDI jacks available for purchase are solder-mount, given the limitations set by the stay at home order, having a MIDI output port on the device was impossible. Luckily, Teensy offers a USB MIDI library as part of its core utilities. The Teensy

USB MIDI library is very user-friendly, and requires no set-up in code. The decision to use USB MIDI, as USB MIDI requires the device to be plugged into a computer, furthers the updated intent of Cacophony as a studio peripheral.

Finally, choice of testing framework was an ultimately limiting decision. Initially, Google Test, a C++ testing library created and maintained by Google, was employed for unit testing. While Google Test has many useful features for unit testing, its documentation is minimal and difficult to glean information from. For this reason, Microsoft CppUnitTestFramework, a unit testing framework built into Microsoft Visual Studio, was employed instead. While CppUnitTestFramework lacks many extended features such as parameterized testing, it is simple and well-documented. Parameterized testing, though formally missing from the framework, can be simulated by iterating on a single test given a large set of test data.

Challenges and Solutions

Cacophony was a difficult project to work on, not only due to the limitations outlined in earlier sections, but due to challenges that appeared during development. Given a lack of knowledge of music theory and a lack of experience working with hardware, developing a hardware-based musical device provided a number of challenges foreseen and unforeseen. Though each challenge varied in difficulty and time consumption, all were able to be solved or worked around in some manner.

The most pressing of initial challenges to Cacophony's development was designing the note generation algorithm. Early in the development cycle, the system was intended to generate a complete scale given a set of notes recorded by the user. This turned out to be rather inflexible,

especially given the goal (at the time) for the device to be performance-oriented. A “scale builder” algorithm, as it was called, would have only allowed for standard heptatonic scales with seven unique notes. For a majority of western music, this would have been a serviceable implementation of the algorithm, but there is much music that uses pentatonic scales (with five notes), and twelve-tone music that uses the entire chromatic scale (all twelve notes). The exclusion of these scales and many others from the program would be a disservice to players of blues, jazz, experimental, and non-western music.

To remedy this, the scale builder was re-specified as a note generation algorithm. The player can record a set of notes, and the device will generate one new note at a time as dictated by the player. Using this system, players can still create standard heptatonic scales as well as scales of any size they dictate. The algorithm still works in the same manner as it would have under the scale builder paradigm; the algorithm produces a single note at a time as opposed to recursively creating new scales until the scale reached a total of 7 notes.

Perhaps the most time-consuming challenge of Cacophony’s development was the design of an input buffer. An input buffer is a circuit that will allow an audio signal to pass through a device unchanged. Personal experience with guitar pedals provided the knowledge that an improper input buffer can ruin an otherwise great device—so-called “buffered bypass” can degrade the quality of the device’s output signal while the device is not active. Though the input buffer was rendered unnecessary due to the changes to the project caused by COVID-19, a great deal of research went into design of the circuit and understanding of how it worked. Parts for the circuit were ordered as well, meaning future iterations of the project would be able to include this buffer. The circuit design that Cacophony’s input buffer would have used is based on a

circuit posted to StackOverflow by user Golaž (2015) in a thread entitled *Single Supply Op-Amp Audio Amplifier*.

A number of minor challenges came about from the creation of the hardware itself. The most frustrating of these challenges was the breadboard compatibility of the Teensy itself with its audio shield installed. The audio shield, extending a little over half a centimeter past either side of the main microcontroller, blocks all lanes of any breadboard it is plugged into. This means that in order to connect any components to the Teensy, either pin sockets needed to be soldered onto the main board, or jumper cables needed to be used to attach the audio shield's headers to the breadboard. As soldering was not an option, the latter option was employed to great success. Though this jumper cable solution made finding which breadboard lane mapped to which Teensy pin difficult, all connections were perfect and made attaching other components to the microcontroller a breeze.

Perhaps the most disappointing of these minor challenges came during the development of a MIDI output. As mentioned in the previous section, the decision to use a USB MIDI connection came after realizing that soldering a MIDI jack to the microcontroller was not an option. However, before fully coming to this decision, an attempt was made to attach part of a MIDI cable to the device as an output. In order to do this, the MIDI cable was split in two, with one of the connectors being dissected for the purpose of determining which wire inside the cable went to which pin in the MIDI connector. It was not until after this that USB MIDI was determined to be an option, meaning the MIDI cable did not have to be dissected at all.

When creating a tempo selector, it quickly became apparent that the readings coming in from the potentiometer being used were unstable. Potentiometer readings on the Arduino

platform come in the form of a return value from zero to one thousand twenty-three. As the microcontroller took readings from the potentiometer, the return value would vary up and down within a range of around twenty without the potentiometer's value changing at all. Multiple potentiometers were tested to ensure that the issue did not lie with a broken component. In the end, it was determined that the voltage output of the Teensy, which the potentiometer relied on, was producing an unstable voltage. At present, the variation in readings is mitigated somewhat by code which regulates the reading value. Future iterations of the project could implement a third footswitch for tap tempo (where users can press the button multiple times at the rate they wish for the tempo to be set), or an outboard circuit that generates a tempo to be read into the microcontroller.

Given the multitude of challenges and limitations for development of Cacophony, the current version still is in keeping with the initial vision and spirit of the project. Though some drastic changes were required to get it to its current state, Cacophony is still a fun, intuitive, experimental piece of music hardware. It has a variety of uses either as a tool to use in long-winded jam sessions, to provide inspiration for controlled compositions, to trigger samples in a new and interesting way, and many more. Overcoming these challenges has been rewarding, and coming out of development with a device that meets the project's initial vision is even more rewarding.

Future Work

Any future work on Cacophony will be to expand upon the current feature set and re-tool the device to be more in line with the initial expectations for the project. Though the current

version has merit as a musical computer peripheral, developing the original project to specification is the end goal. Initial project specifications may require re-visitation given new ideas that have come up in light of development complications. For instance, a tempo selector was not initially planned, but may remain in future versions of Cacophony due to its usefulness.

The first priority of future development would be to build the device's input buffer and MIDI outputs such that the device could be used as a standalone guitar pedal rather than as a computer peripheral. As mentioned in the previous section, parts are available to create this circuit, meaning that as soon as access to a soldering iron is made possible, the buffer can be implemented.

The next priority is to re-develop a user interface and fabricate a project enclosure based on this interface. Having a solid enclosure is a very important part in creating a guitar pedal, as many guitar pedals face severe wear and tear in studios and on stage alike. The protection from harm that an enclosure will provide is something that is sorely lacking from the current iteration of Cacophony. Though the user interface will change given the new user experience loop of the current device, the layout will remain relatively similar in design and principle to Cacophony's initially proposed layout; controls will be spaced apart evenly, and more important controls will have larger knobs or buttons to draw the eye.

Lastly, some of Cacophony's user interactions leave something to be desired. The tempo selector provides unstable tempo values, and the microcontroller does not read the new tempo until the Teensy program's looping functionality returns to the beginning of its loop. This means that not only can tempo change from cycle to cycle, but tempo only updates after the current beat has finished playing. This can lead to jarring jumps in tempo, especially when transitioning to

faster tempos from slower tempos. Additionally, the “double-tap” function of the record button that triggers the generation of a new note only triggers about eighty percent of the time. Generally, when the two taps are performed too soon after one another, the code does not read the signal as a double-tap. Improving upon these interactions will greatly increase the polish of the project.

Project Importance

Working on this project has provided great personal benefit in both learning about circuitry and music. Building guitar pedals, digital or analog, is a skill of great personal interest; designing a hardware device—despite it not being a guitar pedal in its current state—has been confidence-inspiring moving forward into this hobby. Soldering and circuit design are less daunting given this new experience. Additionally, developing a melody generation algorithm has been very educational in terms of learning the basics of music theory and the metaphysics of why certain intervals sound the way they do. Finally, this project has helped demonstrate some modern features of C++, a language which can seem archaic and esoteric to some. Experience using C++11 features and standard library functions has greatly increased personal appreciation for the language and its power.

Outside of personal gain, there is much that this project can offer the world of music. Though many tools exist to create random melodies or to alter existing melodies, no other tool or device at present can create entirely new scales and melodies algorithmically based on notes recorded by the user. Cacophony can not only provide new inspiration to users of similar

devices, but also spark a conversation on the validity of algorithmically generated music in a world increasingly dominated by computers.

REFERENCES

120years. (2014, April 3). 'Moog Synthesisers' Robert Moog. USA, 1964. Retrieved from https://120years.net/moog-synthesisersrobert-moogusa1963-2/

Bresin, R., & Friberg, A. (2000). Emotional Coloring of Computer-Controlled Music Performances. *Computer Music Journal*, *24*(4), 44–63. doi: 10.1162/014892600559515

Britannica. (2015, May 13). Computer music. Retrieved from https://www.britannica.com/art/computer-music

Buchla USA. (2019). History. Retrieved from https://buchla.com/history/

Collins, N. (2014). *Handmade Electronic Music: the Art of Hardware Hacking*. Hoboken: Taylor and Francis.

Corman, E. (2017). Simple Synthesis: Part 11, Sample and Hold. Retrieved October 11, 2019, from https://www.keithmcmillen.com/blog/simple-synthesis-part-11-sample-and-hold/.

elboulangero. (2009). Homemade guitar looper with Arduino. Retrieved October 11, 2019, from http://arduino-guitarlooper.blogspot.com/2009/10/introduction-here-how-to-produce-small.html.

Geem, Z. W. (2009). *Music-inspired harmony search algorithm: theory and applications*. Berlin: Springer.

Georgia State University. (n.d.). Musical Scales. Retrieved October 11, 2019, from http://hyperphysics.phy-astr.gsu.edu/hbase/Music/mussca.html.

Golaž. (2015). *Single supply op-amp audio amplifier*. Retrieved January 28, 2020, from https://electronics.stackexchange.com/questions/153911/single-supply-op-amp-audio-amplifier

Hevner, K. (1936). Experimental Studies of the Elements of Expression in Music. *The American Journal of Psychology, 48*(2), 246-268. doi:10.2307/1415746

Hirschfield, P. (2020). *Gov. Extends State Of Emergency, Stay-At-Home Order, Until May 15*. Retrieved April 23, 2020, from https://www.vpr.org/post/gov-extends-state-emergency-stay-home-order-until-may-15#stream/0

Holmes, T. (2016). Moog: A History in Recordings . Retrieved October 11, 2019, from https://moogfoundation.org/moog-a-history-in-recordings-by-thom-holmes/.

Izagirre, A., Petralanda I. (n.d.). *On the idea of Action in Davies' Performance Theory of Art*. (Doctoral dissertation, The University of the Basque Country, Greater Bilbao, Basque Country, Spain). Retrieved from http://www.gabone.info/txt/izagirre_the-idea-of-action-in-performance-theory-of-art.pdf

Kania, A. (2017, July 11). The Philosophy of Music. Retrieved December 8, 2019, from https://plato.stanford.edu/entries/music/.

LeoMakes. (2019, April 26). What exactly is a Modular Synth? Retrieved from https://www.attackmagazine.com/technique/technique-modular-synthesis/what-exactly-is-a-modular-synth/

Michigan Technical University. (n.d.). Formula for frequency table. Retrieved October 11, 2019, from https://pages.mtu.edu/~suits/NoteFreqCalcs.html.

Music Thing Modular. (n.d.). 22 things to know about the Turing Machine. Retrieved October 11, 2019, from https://musicthing.co.uk/pages/turing.html.

Mylarmelodies. (2016, October 31). *An Intro to Making Generative Music on Modular.* Retrieved from https://www.youtube.com/watch?v=NvrxQbh6vAg.

Puckette, M. S. (1996). Pure Data: Another Integrated Computer Music Environment. *Second Intercollege Computer Music Concerts*, 37–41.

"Thermae: Analog Delay / Pitch Shifter || Mini-Doc." *YouTube*, uploaded by Chase Bliss Audio, 10 May 2018. https://www.youtube.com/watch?v=Zg70GAALnOY

Wright, C. H. G., Welch, T. B., Etter, D. M., & Morrow, M. G. (2002). Teaching hardware-based DSP: Theory to practice. *IEEE International Conference on Acoustics Speech and Signal Processing*. doi: 10.1109/icassp.2002.5745571