Algorithmic Music Composition

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree B.A. in the
Departments of Computer Science and Mathematics at The
College of Wooster

by
Thomas Matlak

The College of Wooster
2018

Advised by:

Nathan Sommer (Computer Science)

Nathan Fox (Mathematics)

THE COLLEGE OF
WOOSTER

Abstract

Algorithmic music composition is a popular area of research in computer aided music; it is the application of computer algorithms to create music. One subtopic of this area is the automated creation of pleasing melodies. This paper explores two machine learning approaches to algorithmically creating melodies: Markov chains and genetic algorithms. Markov chains can generate a rough imitation of the corpus of music upon which they are based. Though Markov chains can generate a rough approximation of a musical style, we want to refine the music to bring it closer to the desired target style. To accomplish this we use genetic algorithms to take an initial population of melodies and tweak and remix them to create better melodies. Genetic algorithms have been used extensively in previous research, but this paper proposes the use of a long short-term memory artificial neural network to act as a surrogate fitness function, rather than defining a set of rules for good music and penalties for breaking those rules.

This work is dedicated to the future generations of Wooster students.

Acknowledgments

I am thankful towards Professor Sommer and Professor Fox for their guidance through the research and writing for this project. Additionally, the Computer Science and Mathematics faculty as a whole have been nothing but encouraging over the past four years, and the courses they taught all helped me determine my interests within the field.

I am grateful for the Music department for allowing me to continue my music education and for providing a creative outlet during my time at Wooster.

Special thanks to my father for first introducing me to programming, and to Tim Dove for solidifying my interest in Computer Science with his Intro to HTML course in middle school.

Finally, my family and friends have provided me with the love and support needed to complete my I.S.

Vita

Fields of Study Major field: Computer Science and Mathematics

Minor field: Music

Contents

List of Figures

List of Tables

CHAPTER 1

Background

Computer Music is a field of computing focused on the creation of music, either as a tool to assist composers or by using the computer to create its own music without human intervention. The field includes the development of programs such as digital audio workstations, which provide a library of sounds for the composer to use and an interface to create music; electronic instruments; and computer generated music. This paper focuses on a subset computer generated music: algorithmic creation of melodies.

The paper is organized as follows: the rest of the first chapter contains background information on musical, mathematical, and computational concepts necessary to fully comprehend the project, the second chapter discusses some of the mathematics related to what makes music pleasing, the third chapter discusses the use of Markov chains to generate melodies, the fourth chapter discusses the use of genetic algorithms in melody generation, and the fifth chapter discusses the software used to implement the concepts in chapters 3 and 4. Finally, the last chapter discusses the results obtained from the project, including some examples of music produced by the software.

1.1 Musical Terminology and Notation

This paper contains the some musical terminology which not all readers may be familiar with. This section contains definitions of musical terms that are important to fully understand the rest of the paper. It is intended as a reference for readers unfamiliar with the terms defined here, so they are provided in alphabetical order, rather than in a guided exploration of the fundamental concepts of music.

Beat: The beat is the fundamental unit of rhythm. What a beat looks like is context dependent. For example, in one piece the beat may be the quarter note, in a second piece it may be a dotted

quarter note, in a third piece it might be the half note. A tempo (speed) is generally provided to determine how long each beat should last. This may take the form of a specific number of beats per minute or a more general term, such as *Allegro* for fast, or *Largo* for slow. The beat can be multiplied or divided into smaller parts to create different rhythms.

Chord: A chord consists of several notes played as a single unit. Most commonly, three or more notes are played simultaneously, though two notes may also constitute a chord. Common chords are based on the major and minor scales, with variations depending on the order of the notes from lowest to highest.

Chord tone: A chord tone is a note contained in a chord. For example, C, E, and G are chord tones of a C Major chord.

Harmony: Harmony consists of all the notes played at the same time as the melody that are not part of the melody. In Western music, these harmonies generally follow a progression of chords based on the key that enhance the melody.

Key: The key of a piece of music determines the chords that appear, as well as the scale the music generally follows. The name of a key takes the form of <Pitch> <Modifier>, where the pitch is one of fifteen pitches, and the modifier specifies the mode of the key, most ofter Major or Minor. The pitch indicates the starting pitch of the scale. Key is notated by the key signature on the music staff, a set of sharps or flats on specific lines and spaces.

Measure: In music, a measure is a collection of notes grouped together in order to give a piece some structure. Measures within a piece generally contain the same number of beats.

Melody: The melody is the most prominent part, often played in the highest voice. It is the recognizable tune that generally defines a piece of music.

MIDI: (Musical Instrument Digital Interface) This is a standard that defines a digital interface, communication protocol, and electrical connectors that allows computers, electronic instruments, and other audio devices to communicate.

Non-chord tone: A non-chord tone is a note that appears while a chord is played but is not part of the chord. Non-chord tones may be used to insert dissonance and drive a piece of music toward resolution. For example, F is a non-chord tone when played over a C Major chord.

Note: A note is a pitch coupled with a duration. The pitch is given as a pitch class and the octave to which the note belongs. For example, C4 refers to the fifth C to appear from the left on a standard 88 key piano. When discussing notes we often ignore the duration and only consider the pitch.

Note length/rhythm: Durations of notes used to indicate how much of each beat a note should occupy and are divided as follows: The quarter note (♩) is often the most basic division of the beat, at

one quarter note per beat. The eight note (♪) is half the duration of the quarter note, while the half note (𝅗𝅥) is double the duration of the quarter note, and the whole note (𝅝) is four times the duration. The names for these note lengths come from how many of that type of note can fit in a common time measure (which lasts for one whole note, or more often four quarter notes). These durations continue in ether direction, so there exist sixteenth and thirty-second notes, as well as double whole notes. Dots can also be applied to notes to add half the duration of the base note. For example, a dotted quarter note (♩.) takes the same amount of time as three consecutive eighth notes (♪♪♪).

Pitch classes: When two notes have the same letter, but are not necessarily in the same octave, they have the same pitch class. Pitch classes divide the octave into distinct parts and relate to the frequencies of the pitches. When discussing the harmonic structure of music, notes of the same pitch class are equivalent, so the octaves in which notes appear do not change the harmonic structure. Along with the seven basic pitch classes (A, B, C, D, E, F, and G), these may also appear with modifiers such as a sharp (♯) or flat (♭) symbol, indicating the pitch should be raised or lowered. In total there are twelve distinct pitch classes.

Scale: A scale is a way of choosing which notes to include in a piece and is closely related to the concept of a key. Scales begin on one pitch class, and ascend one octave before repeating. Different modes have different feelings. For example, the major mode (Ionian) sounds more bright and happy, while the minor modes (Aeolian, Dorian, and Phrygian) sound more dark and sad.

Tonic: The tonic pitch is the base note of a scale. For example, in the C Major scale, C is the tonic.

1.2 Writing Melodies

In the tradition of Western music, there exist some general guidelines for creating pleasing melodies, a small sample of which follow. Transitions between notes should primarily occur in a stepwise manner, and one should avoid leaps of more than an octave. A piece of music as a whole should follow a chord progression, or set of recognizable chords between which a piece moves. Additionally, if we use the seventh note (also called the *leading tone*) in the scale, the next note should be the tonic of the scale. These are not all of the rules for a good melody, and some of the rules are rather broad, but they provide a decent starting point to quickly identify good and bad melodies.

Let us examine a simple example of writing our own music. Perhaps the most common chord progression is I-V-IV-vi-I, which means the first chord is based upon the first note of the scale of the key of the piece, the next chord is based upon the fifth note in the scale, the next upon the fourth note, the next upon the sixth, and finally the final chord is based upon the first note in the scale. If

we are in the key of C Major, we might start by simply making our melody C G F A C, as in Figure 1.1, but this is rather boring, does not move in a stepwise manner, and breaks some more advanced part writing rules. Rather, we might rearrange the order in which notes appear in each chord to produce more interesting melodic and harmonic structure, as in Figure 1.2. In this version, we use the same chords, but the melody moves primarily stepwise and we do not break any of the more advanced part writing rules.

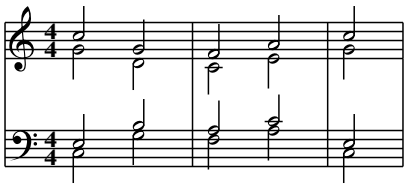

Figure 1.1: A simple example of a bad melody.

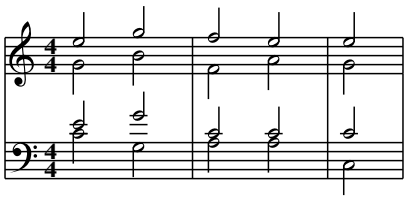

Figure 1.2: A simple example of a good melody.

1.3 Markov Chains

Intuitively, we may think of a Markov chain as a machine that accepts some former state or states of a system and produces the next state in the system. The decision of what the next state should be is based on probabilities of transitioning to different states from the current state of the system.

1.3.1 Definition

More formally, a *Markov chain* is a type of discrete-time stochastic process, which means a Markov chain is a sequence of random variables $\boldsymbol{X} = \{X_n | n \in I\}$ for some index set I. Additionally, Markov chains have the special property that they depend only on the immediate past state(s). That is, for a first-order Markov chain at time t,

$$P(X_t = j \mid X_0 = i_0, \ldots, X_{t-1} = i_{i-1}) = P(X_t = j \mid X_{t-1} = i_{t-1})$$

for a particular possible outcome j of X_t [26].

This idea can also extend to higher-order Markov chains. A higher-order Markov process considers more than the single most recent state to determine the next state. An nth order Markov chain uses the previous n states as the input to find the next state.

1.3.2 Representations

In the case when $n = 1$, we can think of a Markov chain as a directed graph, where each state is a node, each edge is a transition between states, and the probabilities of transitioning between states are represented by the edge weights. See Figure 1.3 for a visual representation of this idea.

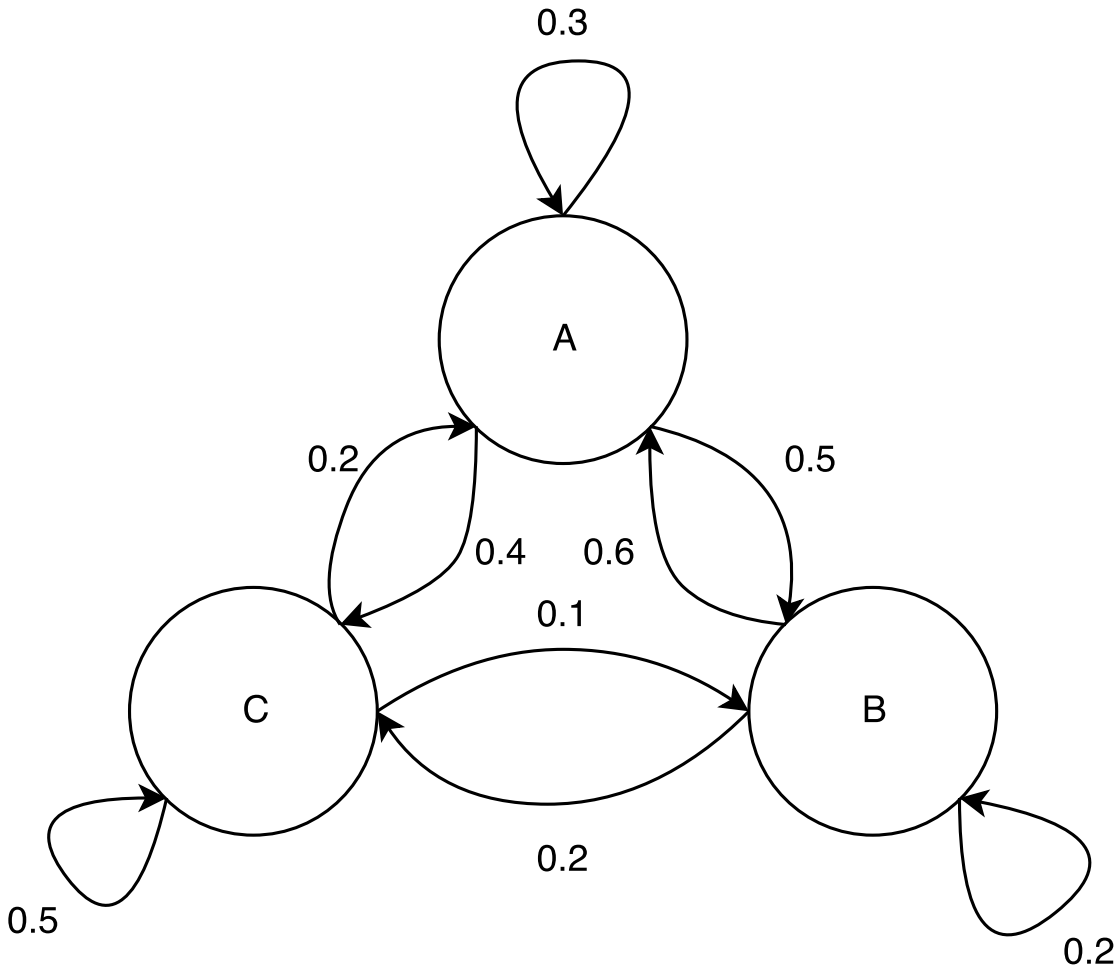

Figure 1.3: A Markov chain represented as a graph. Arrows between nodes represent transitions between nodes.

When implementing a Markov chain in code, however, it is perhaps easier to represent it as an

($n+1$)-dimensional array, where n is the order of the Markov chain. We call this ($n+1$)-dimensional array the *transition matrix*. It contains the probabilities of transitioning from one state to another. See Figure 1.4 for an example of the same Markov chain as in Figure 1.3 in matrix form. Note that the matrix representation is essentially an *adjacency matrix* of the graph, where edge weights are the probabilities of transitioning between nodes.

	A_1	B_1	C_1
A_0	0.3	0.5	0.2
B_0	0.6	0.2	0.2
C_0	0.4	0.1	0.5

Figure 1.4: A Markov chain represented as a transition matrix. Rows represent transitions from the labeling node to the labeling node of each column.

1.3.3 Limitations

A major limitation of Markov chains is their inability to generate truly novel output. In order for some state to appear, a transition to that state from the previous state must appear in the *source material*, the set of data from which the transition probabilities come. That is, no truly novel transitions may appear; all transitions that appear in the output of the Markov chain must have appeared somewhere before. Additionally, lower-order chains may produce nonsensical output, whereas a chain of sufficiently high order will exactly copy the source material. Another limitation is that the process may get stuck in a "local loop". This may happen when the chain proceeds to a state which only transitions to itself or transitions to a set of states that only transition to each other.

See Chapter 3 for more information on how Markov chains are used in this project.

1.4 Artificial Neural Networks

Artificial neural networks (ANNs) are a computing construct inspired by biological brains that contain a collection of connected *neurons*, which we also call *nodes*. An ANN consists of a layer of input nodes, a layer of output nodes, and zero or more layers of nodes in between the input and output layers called hidden layers. A neuron accepts one or more inputs and performs a weighted summation of the neuron's inputs and a constant input (*bias*) to produce its output. The *input layer* consists of nodes that accept numerical inputs from some outside source. The *output layer* consists of nodes that expose the weighted sum of their inputs to outside sources. *Hidden layers* of nodes accept numeric inputs from the previous layer and produce numeric outputs that pass to following layers. The

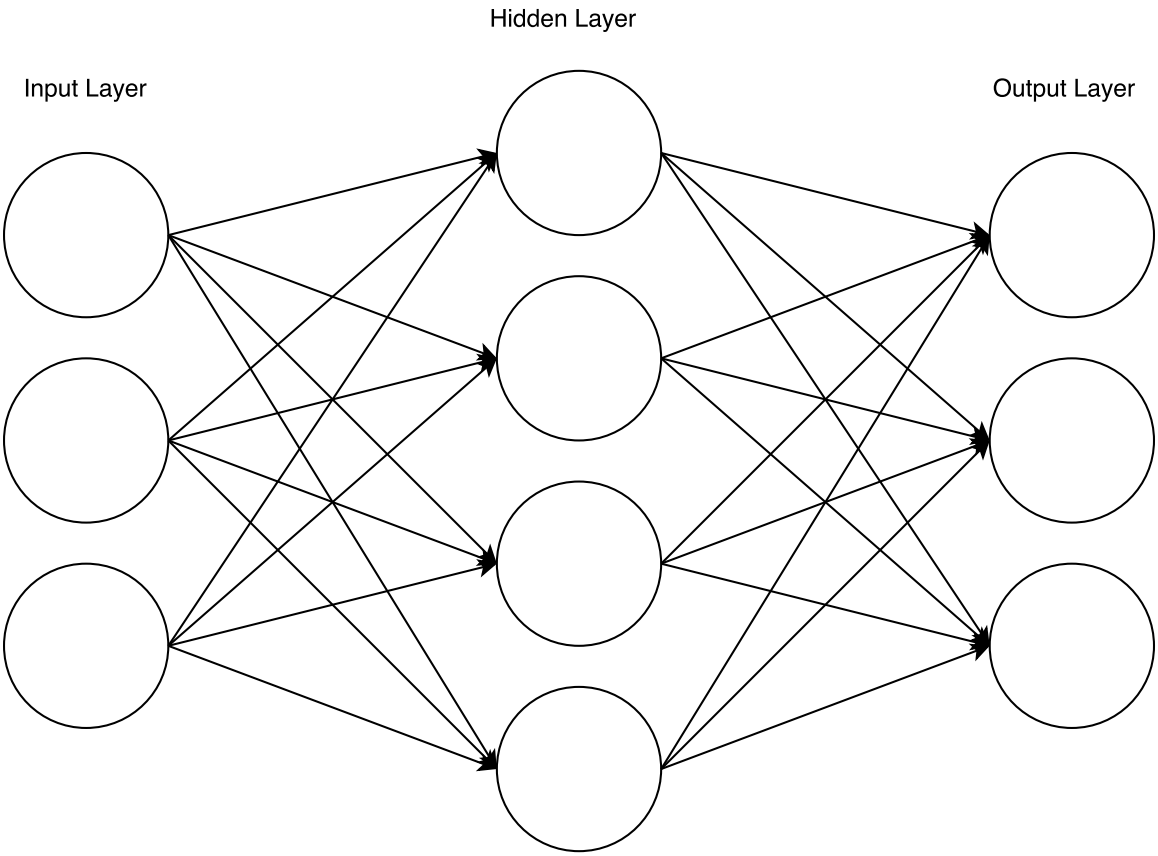

Figure 1.5: A simple ANN with an input layer, one hidden layer, and an output layer.

previous layer can be either the input layer or another hidden layer and the following layer can be either another hidden layer or the output layer. Hidden layers often all contain the same number of nodes, though this is not necessary. In general, the output from each node in a given layer is passed to every node in the following layer, and each node receives the output from every node in the previous layer as input. See Figure 1.5 for a visual of a simple ANN.

To constrain the output of the neurons, we use an *activation function* to provide upper and lower bounds for the output and to force these values towards those new bounds. Common activation functions include the identity, sigmoid (or logistic), binary step, and arctangent functions. The identity function preserves the computed value as is, without any constraint on its bounds. The binary step function forces the computed value to either zero or one, while the sigmoid function merely bounds the computed value to between zero and one. The arctangent function bounds the output between $\frac{-\pi}{2}$ and $\frac{\pi}{2}$, and provides a faster ramp up from the minimum value to the maximum value than the sigmoid function. See Figure 1.6 for some visualizations of these activation functions.

Each activation function has a different case where we might want to use it. The binary step

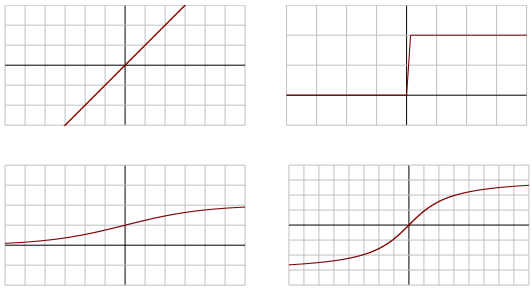

Figure 1.6: Some common activation functions. Clockwise from the top left: identity, binary step, arctangent, sigmoid.

function works well when we want a binary classification, such as in the output layer: does the input belong to a particular category? The sigmoid and arctangent functions both constrain the output to a range of real numbers, but with different bounds and at different rates of convergence. One possible use case for this type of function is to provide the percent certainty that an input belongs to some category. We use the identity function when we do not want to alter the output in any way.

In order for an ANN to be useful, it must go through training to set the bias for each node and the weight of every connection between nodes in order to accurately predict data. The most common algorithm to train ANNs is the *backpropogration* algorithm. In this, the weights are randomly initialized, then inputs are passed throught the network and the output is compared to the expected output. Then, the error is used to determine the magnitude of the change to the weights.

For time series data, normal ANNs are not suitable because they consider only a single input at once. Instead, we can use *recurrent neural network* (RNN) setups, which consider the results from previous inputs when evaluating the current input. In particular, we utilize a long-short term artificial neural network, which stores information about previous input until it determines that information is no longer relevant, at which point it is forgotten.

1.5 Genetic Algorithms

A *genetic algorithm* (GA) is an iterative process that takes an initial population of individuals and remixes and mutates that population to produce a new set of individuals to become the initial population of the next generation. This new set is produced by performing various operations on the initial population, then using a fitness function to choose the best performing individuals. The idea is that over the course of many generations, fitness tends to increase, as only the best performing individuals are allowed to survive to the next generation.

An essential operation for GAs is *crossover*, that is, splicing individuals together to produce new sequences. Other operations can also be defined to operate on or more individuals. After creating the new candidate population, we introduce mutations to keep the population from stagnating. These mutations make random changes to the candidate population. For example, in a binary string, a 0 may get flipped to a 1, which may be a small change, but it could improve the fitness of the individual.

To choose the new initial population, we define a fitness function which measures how close an individual is to the desired output. If some individuals reach a certain level of fitness, they are the output of the genetic algorithm. Otherwise, the best performing individuals become the initial population of the next generation. This process continues until some individuals perform well enough, a maximum number of generations is reached, or fitnesses stagnate.

As a simple example, consider a population of four random 4-bit bit strings. Our goal is to produce a bit string consisting of all 1s. As our fitness function, let each 1 in the bit string add 0.25 to the fitness, so a bit string of all 1s has a fitness of 1.0. Let the initial population contain the bit strings 0100, 1001, 0010, and 1100, which have fitness values of 0.25, 0.5, 0.25, and 0.5, respectively. We put these individuals into the pool of candidate strings. Next, we can randomly select some of these strings to crossover. Consider the first and second bit strings: 0100 and 1001. For the sake of simplicity, let us perform crossover at the middle. This gives us the candidates 0101 and 1000. Next, consider the third and fourth bit strings: 0010 and 1100. Again, to keep the example simple, let us perform crossover at the middle. This gives us two more candidate strings, 1110 and 0000. At this point, the candidate population is 0100, 1001, 0010, 1100, 0101, 1000, 1110, and 0000. Suppose no mutations are chosen for this generation.

Choosing the fittest four of these, we get 1110, 1001, 1100, and 0101. Now perform crossover of the second and third bit strings, again at the middle. This gives us 1000 and 1101. At this point, the candidate population consists of 1110, 1001, 1100, 0101, 1000, and 1101. Let us preform some mutations this time. Suppose we randomly choose the first and fifth strings to mutate, each at position 4. This means we flip the bits of these strings at the fourth position. Now the candidate population is 1111, 1001, 1100, 0101, 1001, and 1101. We have a bit string consisting entirely of 1s, so our goal has been accomplished, and we may exit the process.

See Chapter 4 for more information on how genetic algorithms are used in this project.

CHAPTER 2

Mathematics In Music

2.1 The Harmonic Series

An important concept for pitch sounds is the harmonic series. Pitched sounds have a *fundamental frequency* from which an overtone series is produced. *Overtones* are any frequencies higher than the fundamental frequency. The *harmonic series*[1] is a restricted subset of the overtone series consisting of the integer multiples of the fundamental frequency. For example, consider the open A string of a cello which is pitched as an A3. In other words it has a fundamental frequency of 220 Hz. Doubling this frequency gives 440 Hz, or an octave above the fundamental frequency. Tripling the frequency gives 660 Hz, which is an E5. This continues, so multiplying by 4 gives a frequency two octaves above the fundamental, multiplying by 5 produces a C♯6, multiplying by 6 gives an E6, multiplying by 7 gives a G6, and multiplying by 8 gives an A6, three octaves above the fundamental frequency. We could go on, and eventually the frequencies would pass out of the human hearing range (approximately 20000 Hz).

The presence of the overtone series, and in particular the harmonic series is what makes a sound pitched. See Figures 2.1, 2.2, and 2.3 for examples of a sine wave, the first five harmonic sine waves overlayed on top of each other, and the sum of the first five harmonics, respectively. Percussive sounds such as hitting a drum or dropping a brick do produce pressure in the air, but the oscillations of the pressure are not regular, and so do not produce any pitched frequencies.

Brass instruments are built with the harmonic series in mind. For example, the tenor trombone has a fundamental pitch of a B♭1, or 58.87 Hz. The pitches that can be played on the instrument in

[1] Note that this harmonic series is different from, though related to, the mathematical harmonic series.

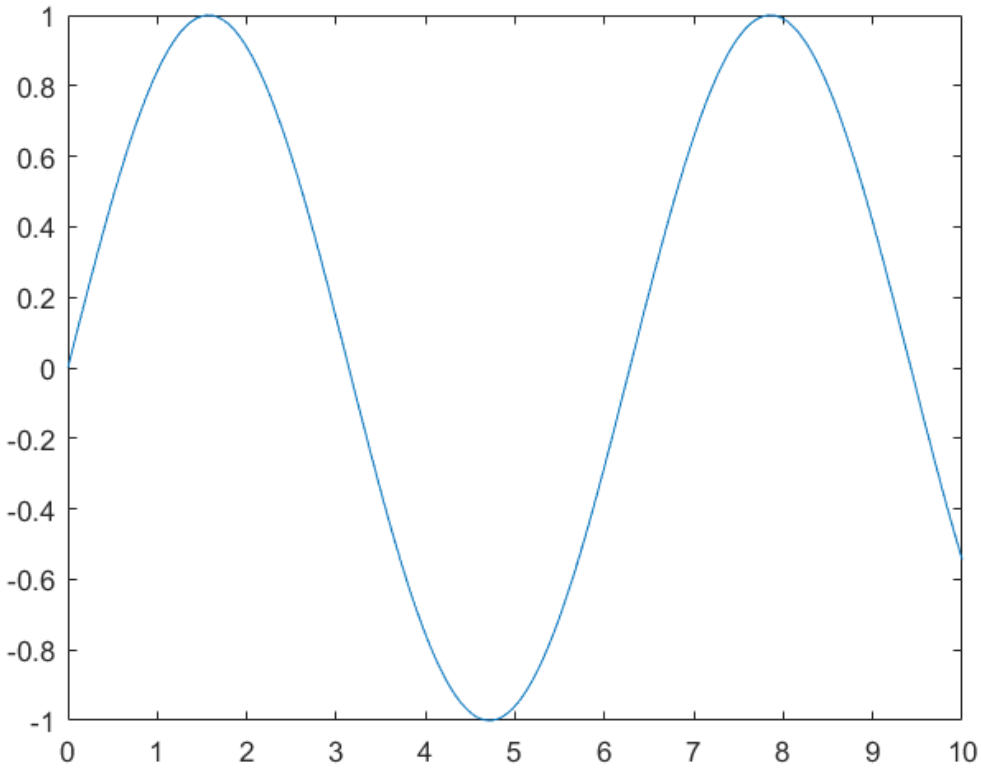

Figure 2.1: A pure sine wave

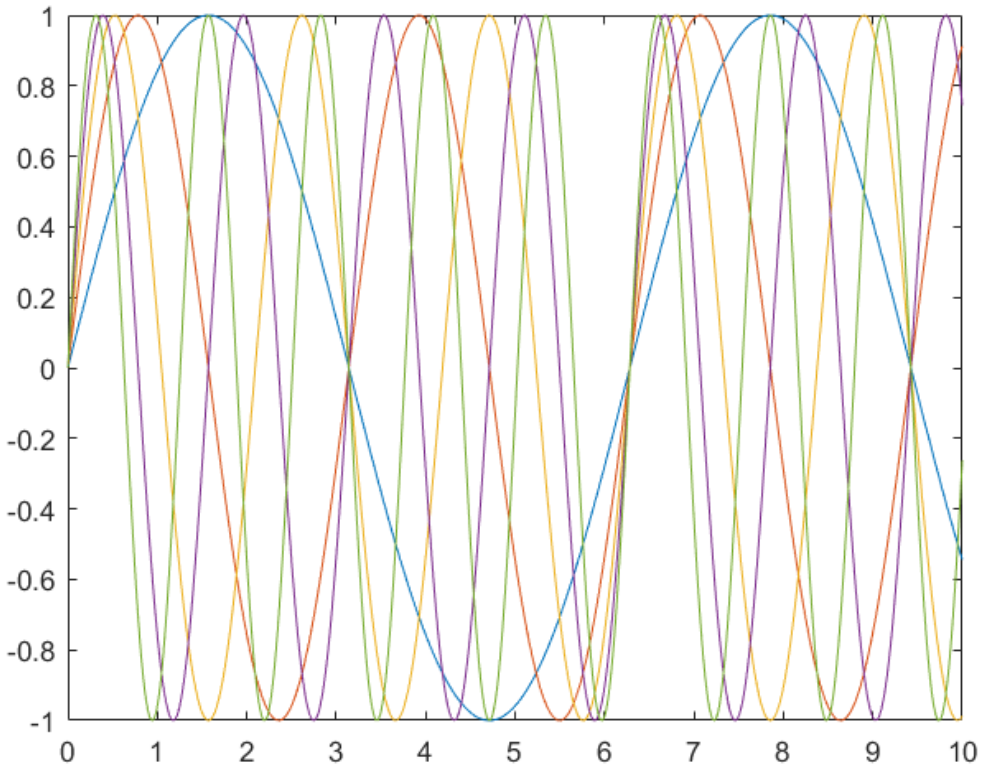

Figure 2.2: The first five harmonic frequencies of a sine wave.

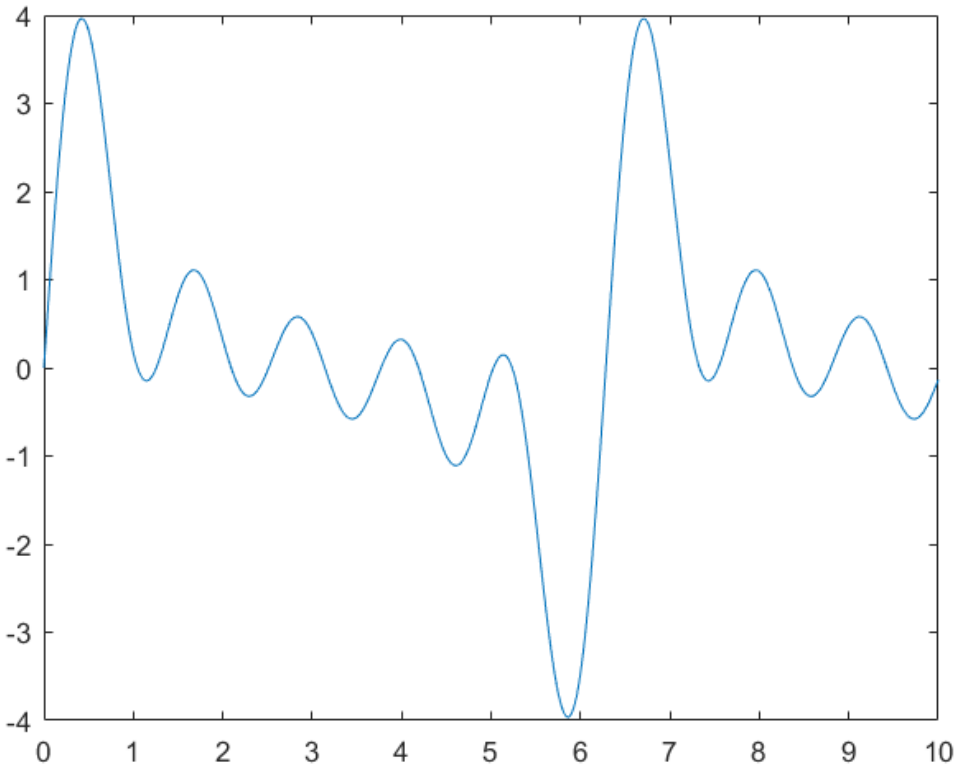

Figure 2.3: The sum of the first five harmonic sine waves.

Semitones From Base	Ratio	Interval Name
0	$\frac{1}{1}$	unison
1	$\frac{16}{15}$	minor second
2	$\frac{9}{8}$	major second
3	$\frac{6}{5}$	minor third
4	$\frac{5}{4}$	major third
5	$\frac{4}{3}$	perfect fourth
6	$\frac{45}{32}$	diminished fifth/tritone
7	$\frac{3}{2}$	perfect fifth
8	$\frac{8}{5}$	minor sixth
9	$\frac{5}{3}$	major sixth
10	$\frac{9}{5}$	minor seventh
11	$\frac{15}{8}$	major seventh
12	$\frac{2}{1}$	octave

Table 2.1: Ratios for intervals under just intonation.

first position (i.e. without moving the slide, which would change the fundamental frequency) are those of the harmonic series based on B♭1. Thus the trombone can theoretically play B♭1, B♭2, F3, B♭3, D4, F4, a very flat G♯4, B♭4, C5, D5, a very flat E5, F5, and so on. In practice however, due to the physical demands of the player, any note above B♭4 is uncommon, and any note above D5 is rare.

When discussing brass instruments, there are some modifications made to the frequencies by the shapes of the the bell and mouthpiece of the instrument, but those are beyond the scope of this project.

2.2 Intervals Between Notes

In order to discuss intervals between notes, we must first choose a tuning system. The two most common tuning systems are just intonation and equal temperament.

2.2.1 Just Intonation

Just intonation is based on the concept of perfect intonation, whereby an interval is defined as multiplying a pitch be some rational constant. Table 2.1 contains the rational ratios for intervals through the first octave.

The just tuning system produces the best sounding intervals and is used to tune intervals by ear. The downside is that the size of an interval is different than the sum of semi tones to reach the same

interval. For example, the frequency of major third above a given note is $\frac{6}{5}$ that of the given note. Building that same interval (4 semi tones) by moving up the chromatic scale, however, the ratio is $(\frac{16}{15})^4 = \frac{65536}{50625} \approx \frac{6.5}{5}$. This is a significant difference from the ratio for a major third.

2.2.2 Equal Temperament

The equal temperament tuning systems breaks the octave into twelve equal parts based on the logarithmic frequency distance from the previous semitone. That is, each semitone away from the base note is a multiple of a power of the twelfth root of two. The formula for an interval is $b \times 2^{\frac{x}{12}}$ where b is the frequency of the base note and x is size of the interval in semitones.

The result of this tuning system is that it produces intervals that are close enough to the perfect intervals of just intonation to sound decent for any interval, but do not sound as good as justly tuned intervals. Modern instruments with fixed pitches such as the piano and guitar use this tuning system.

CHAPTER 3

A Markov Model for Melody Generation

3.1 Setup of the Model

In this model, we use two Markov processes: one to determine interval changes between notes to generate pitches, and the other to generate rhythms. To create the interval transition matrix, we iterate over the source – in our case a corpus of music by J.S. Bach – keeping track of the previous interval at each step and incrementing the matrix at position $a_{i,j}$, where i is the previous interval and j is the current interval between notes. Note that for a matrix m, $m_{i,j}$ indicates the element in row i and column j. The rhythmic transition matrix is similarly generated. While iterating over the source to generate the interval transition matrix, we can also build the rhythmic transition matrix. To do this we keep track of the duration of the previous rhythm at each step and increment the transition matrix at position $b_{k,l}$ where k is the previous rhythmic duration and l is the current rhythmic duration.

Because the number of possible intervals and rhythms are theoretically infinite and the transition matrix would be very sparse, we do not want to try to store a matrix capable of holding every possible transition. Instead, we can use a hash table to only store transitions that actually appear in the source.

An example rhythmic transition matrix is given in Figure 3.1, based on the *Little Fugue in G Minor* by J.S. Bach.

3.2 Generation

After the transition matrices are created, the program randomly selects a rhythmic value, based on the probabilities that correspond with the previous note, and a pitch. The pitch value comes from

	0.25	0.5	0.75	1.0	1.25	1.5	2.0	2.25
0.25	606	21	1	2	1	2	0	1
0.5	23	109	0	7	1	0	0	0
0.75	1	0	0	0	0	0	0	0
1.0	1	6	0	3	0	3	2	0
1.25	2	0	0	0	0	0	0	0
1.5	0	5	0	0	0	0	0	0
2.0	0	0	0	2	0	0	0	0
2.25	1	0	0	0	0	0	0	0

Figure 3.1: An example rhythmic transition matrix. A value of 1.0 indicates a quarter note.

the interval from the previous note. The interval is chosen based on the probabilities that correspond with the previous interval.

The program stops generating notes when the stopping criteria are met: the generated melody is at least eight measures, the melody ends at the end of a measure, and the last note is a tonic chord tone. Because the generated melody could theoretically be infinite, the program terminates generation after one-hundred measures are generated whether or not the stopping criteria are met.

CHAPTER 4

Genetic Algorithms

4.1 Fitness Function

Several previous papers discuss the use of interactive fitness functions [27, 25]. With this method, a human listens to the melodies and selects the best from each generation to be used as the parents of the next generation. This method does well at picking the most pleasing music to human ears, but it requires humans to listen to the music, which is slow and can lead to fatigue on the part of the evaluators.

Rather than rely on humans who may become tired, and may not be completely objective, we want to use an automated fitness function. Some authors discuss techniques for fitness functions as applied to music [27, 14**?**]. For example, Alan de Freitas and Frederico Guimaraes use a fitness function that penalizes any note outside of the C Major scale [14], while George Papadopoulos and Geraint Wiggins use a fitness function that considers several characteristics of the melody, including consecutive intervals, note durations, and melodic contour [27]. Music from the common practice period of Western music, which lasted from the late Baroque period through the Romantic period (1650-1900), generally followed a complex set of rules regarding harmony, rhythm, and duration. We could manually define a function that applies some penalty for breaking one of these rules, but this approach could be highly error prone and require lots of manual tweaking. Additionally, if we want to create music resembling a different era, we would have to write a new fitness function.

Rather, a fitness function can be approximated using what we call a *surrogate model*. A surrogate model is used in a context where direct measurement is computationally expensive or otherwise difficult to define. In this project we use an artificial neural network as the surrogate model. We expect the neural network to pick up on which rules are the most important, altering its weights

accordingly. Because music has temporal properties, the type of neural network used in this project is a *long short-term memory* (LSTM) neural network. An LSTM network is a special type of *recurrent neural network*, where nodes link back to themselves. This property of saving information about previous elements in the input sequence allows the LSTM neural network to build relationships between parts of the inputs that are far apart in a time sequence. Figure 4.1 shows the idea that the input to the LSTM neuron is kept in subsequent iterations by the neuron passing a value back to itself. in_k and out_k indicate the values passed to and from the cell at time k, respectively.

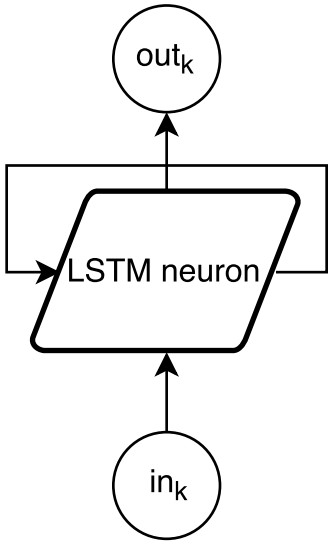

Figure 4.1: An LSTM Neuron

We have two classes of training data: "good" melodies, which were written by a real composer, and "bad" melodies, which are entirely randomly generated before training the fitness function. We expect the network might pick up on attributes such as how long a note lasts, what the pitch of a note is, and its relation to the notes around it. Notes can be related to those around them in terms of pitch – pitches might form a broken chord – or in terms of rhythm – eighth notes or sixteenth notes often appear in groups. However, we cannot know for certain what the network decides is important before training.

In our corpus of music, the sixteenth note is most commonly the smallest rhythmic value in a piece of music, so we first preprocess the music data by converting all rhythms to consecutive sixteenth notes. The motivation for this method, rather than trying to encode the rhythm in another way comes from Peter Todd [31]. See Section 5.3.1 for more information about how the data is presented to the neural network and the configuration of the network.

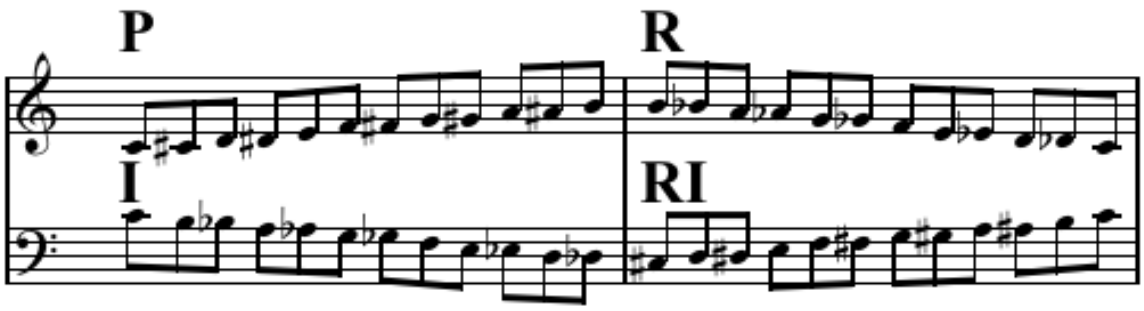

Figure 4.2: Common clockwise from the top left: prime (original form), retrograde, inverse, and retrograde-inverse.

Using this network setup, we regularly achieved 99% accuracy determining which samples are real music and which are random notes.

A detailed description of the software implementation of the fitness function is in Section 5.3.1

4.2 Mutations

There are several natural mutations to consider when working with music. Two common compositional techniques are inversion and retrograde. Inversion takes a section of music, generally a couple of measures, and inverts all the intervals, so an interval up becomes the same interval down. Retrograde reverses the order of one or both of the rhythms and pitches of a section of music. Retrograde and inverse can also be combined, to yield retrograde-inverse. Typically, the section is inverted, and the new notes are then read backwards, though some composers, such as Igor Stravinsky, reversed the order of the notes first. See Figure 4.2 for an example of inversion, retrograde, and retrograde-inversion. These techniques are especially common in twelve tone music, which was developed during the early twentieth century by Arnold Schoenberg, though they can also be found in other types of music. An example of a twelve tone piece is Schoenberg's *Wind Quintet* Op. 26. In our genetic algorithm, we apply these compositional techniques to the melodies from the previous generation to produce new candidates for the current generation.

4.3 Crossover

An important idea when working with genetic algorithms is *crossover*, where members of the population are spliced together to create new candidates. In crossover, the fittest members of the previous generation are "mated" by choosing genes from two parents, changing which parent the

genes come from at each of a set of crossover points. One may use any method they want to choose these crossover points. For example, they might be random or crossover might always occur at the middle. Finally, to maintain diversity in the population, we introduce some random changes to the population at a rate determined by some volatility factor, which can either stay constant or change over time.

A detailed description of the genetic algorithm software is in Section 5.3.

CHAPTER 5

The Software

This chapter discusses the various components of the software used in this project. All components use `music21`, a Python library for music developed by MIT, to handle input and output of music notation to and from a Python-native format. Input files are in MIDI format.

5.1 Overview

The data flow for the set of programs used in this project are as follows: Optionally, the Markov chain program reads a corpus of music to create its transition matrices. Because good music exists in many keys, when reading the corpus of music, we normalize all the music to the key of C Major or A Minor. These keys are chosen because their key signature contains no sharps and no flats. We also cache the normalized corpus, so we do not need to go through the process of determining the key of a piece and transposing it every time we run the program. These Markov chains are then used to produce the initial population for the genetic algorithm. If we choose not to use the Markov chain program to generate the initial population, we use a randomly generated initial population instead. The genetic algorithm uses an LSTM neural network (see Section 4.1) as a surrogate fitness function, which was trained with its own music corpus. See Figure 5.1 for a visual representation of the broad flow of data.

5.2 Markov Melody Generator

In the code repository for this project, the directory `intervalMarkovChain` contains Python code to generate melodies using Markov chains of arbitrary order. The theory behind this software component is discussed in Section 1.3 and Chapter 3. The software uses two Markov chains to

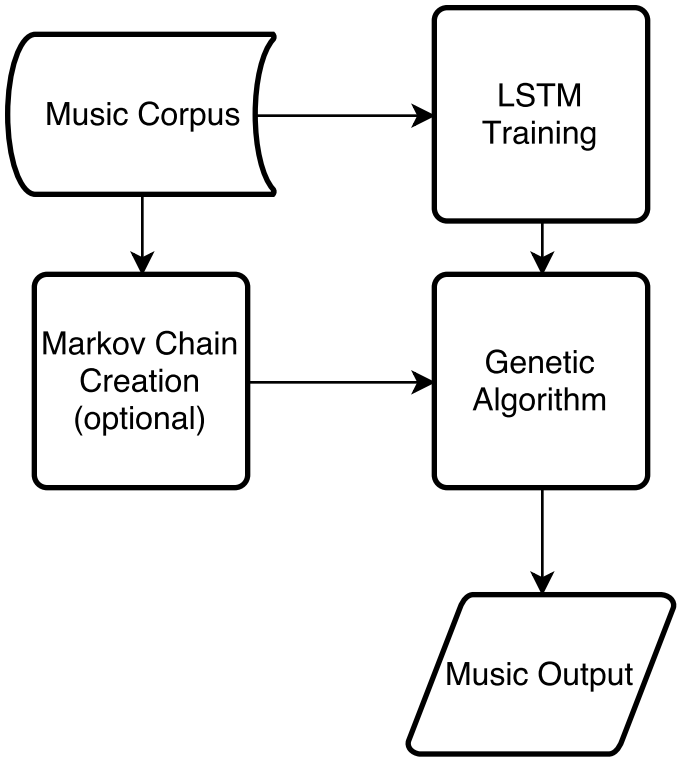

Figure 5.1: The flow of data through the software.

generate a melody: one for the intervals between notes and the other for rhythms of notes. See Figure 5.2 for a visual representation of the flow of data through the Markov melody generation.

In order to facilitate the creation of Markov chains in a more general way, we create a `MarkovChain` object, which provides methods to create and use the Markov chain. When a `MarkovChain` object is created, it requires an order to use. By default, the order is 1. To create the transition matrix, the function `create_transition_matrix(streams, chainType)` accepts a list of the `music21 Streams` to use as the training source, as well as what type of Markov chain it should be – should it calculate transition probabilities for rhythms or for the intervals between notes?

An nth order Markov chain is represented using `defaultdicts` nested $n+1$ deep. Two interesting functions in this class are `arbitrary_depth_dict_get` and `arbitrary_depth_dict_set`. These functions allow one to read from and write to `dicts` of arbitrary depth. These are necessary to be able to create and use the transition matrices because the order of the Markov chain may not always be the same, so the number of index operators desired cannot be hard-coded.

The function `arbitrary_depth_dict_get(subscripts, default, nested_dict)` works by recursively accessing `nested_dict` with the first element of `subscripts` until either `subscripts` is empty or a key from `subscripts` does not exist. In the former case, the function returns the value of

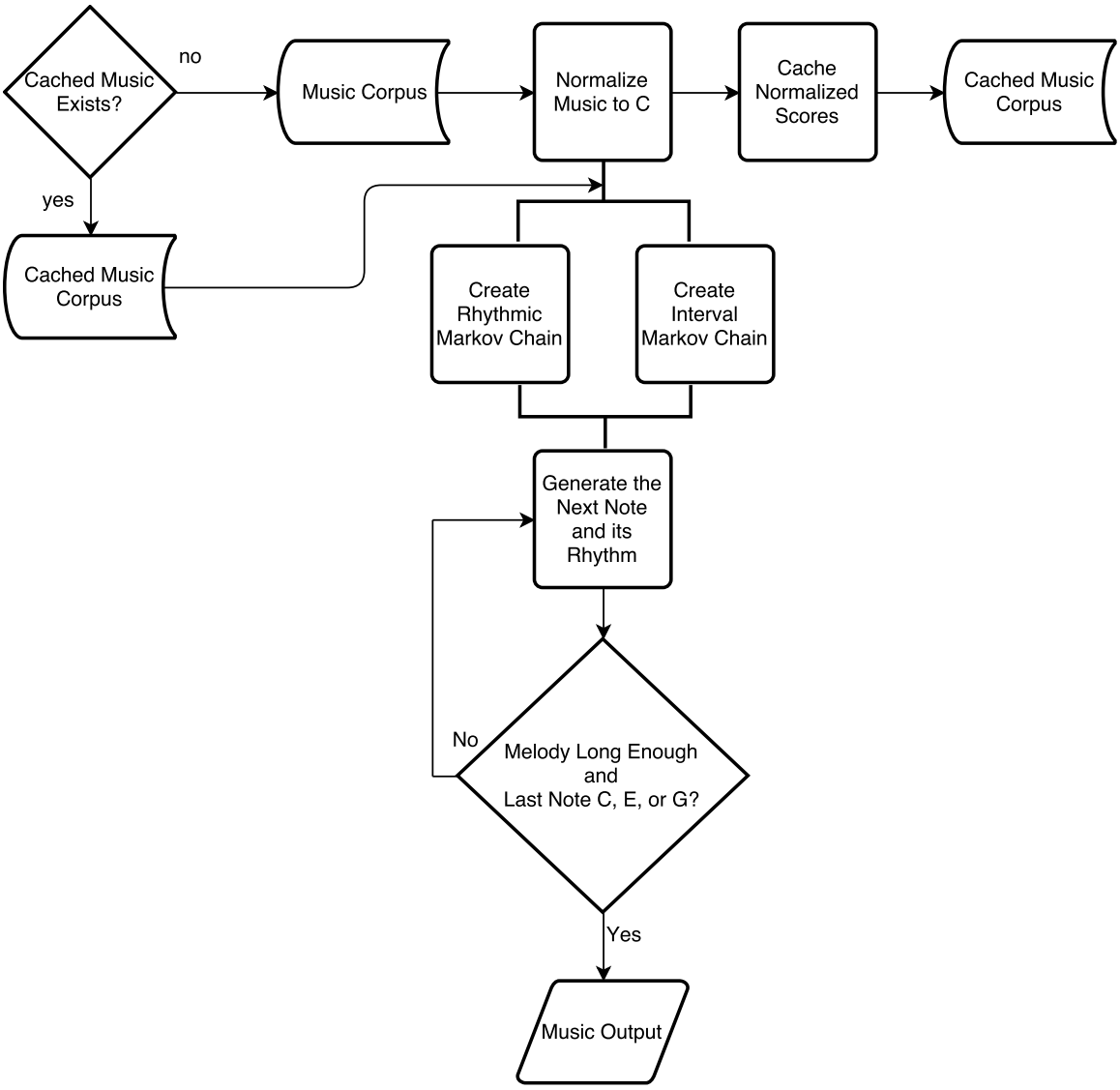

Figure 5.2: The flow of data through the creation of a Markov melody.

`nested_dict` specified by `subscripts`. In the latter case, the function returns the specified default value.

The function `arbitrary_depth_dict_set(subscripts, _dict, val)` works by iterating through the `subscripts` of `_dict`, going a level deeper at each step. At each level, an empty dictionary is created if one did not already exist at that index. This function makes use of the fact that the `dict` data structure in Python can be assigned to a new variable by reference in order to go into the next index. On the final index, the dictionary is assigned the value at the (possibly nested) index as specified by `subscripts`.

The `MarkovChain` class is implemented in `intervalMarkovChain/markovChain.py`.

In order to create a Markov chain, we need a corpus with which to build it. Conveniently, `music21` provides the ability to read and write a variety of file formats. In this project, we use MIDI files to store the music we use as a corpus and that we generate.

Because a Markov chain requires some predetermined seed notes to refer to, the first four notes of each generated melody are C, D, E, and D. These pitches were chosen because the intervals between them (Major seconds up and down) will appear in almost any corpus of music at least once. To keep some rhythmic diversity, the duration for each of these four notes is randomly chosen to be a sixteenth note, eighth note, or quarter note. The decision to include four seed notes was subjectively made by the author's thoughts that melodies produced using a fourth order interval Markov chain were not sufficiently better than melodies produced using a third order chain to warrant the extra computational cost. In order to use a Markov chain with an order higher than four, more seed notes must be provided.

Once the interval and rhythm Markov chains are generated and we have our seed notes, we can generate a melody.

The process of reading reading a corpus of music, creating the Markov chains, and generating a melody is implemented in `intervalMarkovChain/intervalMarkovChain.py`.

5.3 Genetic Algorithms

The directory `geneticAlgorithms` contains Python code to manipulate existing melodies with genetic algorithms to produce better output. The theory behind this software component is discussed in Section 1.5 and Chapter 4.

5.3.1 Fitness Function

This program uses an LSTM neural network created using `TensorFlow` as a surrogate fitness function. `TensorFlow` is a machine learning library for Python developed by the Google Brain team. The file `geneticAlgorithms/lstm.py` contains code to convert music data to a form usable by the artificial neural network, train the LSTM neural network on that modified music data, and save the generated model for later use. See Figure 5.3 for the flow of data through the creation of the fitness function.

The first step is to read a corpus of music to train the function on. If no music corpus is provided, the program uses the collection of Bach chorales that come with `music21`. As with the Markov chain module, this module normalizes the melodies to the key of C before training, using cached copies of

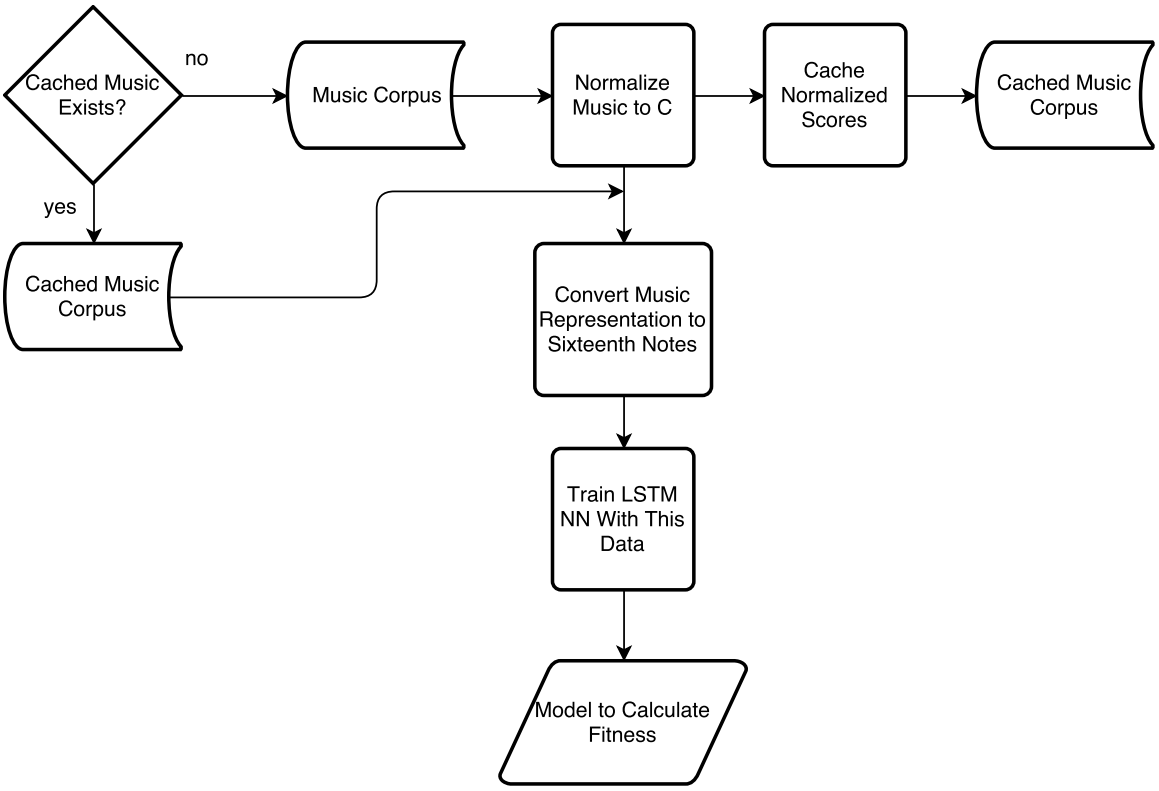

Figure 5.3: The flow of data through the creation of the LSTM neural network to use as the surrogate fitness function.

the normalized scores if they are available. This removes the need to take the key a piece is into account.

After the training input is normalized it is converted to a form the artificial neural network will be able to recognize. We perform some preprocessing by one-hot encoding the pitch of each note. To limit the range of octaves that can appear in our music, we use the MIDI values for each pitch. Because the MIDI standard defines values from 0 to 127, we can *one-hot encode* the pitch as an array of zeros with a 1 at the index of the MIDI number we want to represent. One-hot encoding is a method used in data science to represent categorical data in a numeric format. Even though the MIDI values are already numeric, we one-hot encode them because the difference of a half step (one MIDI value) can be the distinction between a good melody and a better melody. Additionally, MIDI values are only ever integer values, so we do not need to represent them as floats. Importantly, one-hot encoding the MIDI values improves the accuracy of the network. Now that the good music data is in a form the neural network can recognize, a number of random sequences of sixteenth notes is generated to act as samples of bad samples. We want the neural network to train on an evenly distributed sampling of good and bad data, so we shuffle the list.

Now the program is ready to create and train the LSTM network. Unlike a traditional neural network, an input sample is not entered all at once. Instead, each note enters the network one at a time. For this project, we consider four measures of 4/4 music as input to the neural network. With our rhythmic encoding of sixteenth notes, this means for each music sample we input $16 * 4 = 64$ notes to the network. In more general terminology, each note is a *chunk*. Since each note consists of a single one-hot encoded pitch, the *chunk size* is 128. Somewhat arbitrarily, we pass each input through 256 neurons in the network. The output of the network is an array containing two values, which are the network's certainty that the input is or is not a good melody.

To train the model, we run the training data through a function to reshape it to a N x 64 x 128 list, where N is the number of training samples. 70% of this reshaped data is then fed through the neural network for five epochs. That is, the data is fed to the network and weights are adjusted five times. At this point, the accuracy of the model is found by evaluating the remaining 30% of the training data and comparing the output to the actual labels. If the accuracy is sufficient, the `TensorFlow` session and the model are returned to be used on new data later.

Using the `evaluate_part()` function, the fitness for a melody is the output of the mathematical squashing function $f(x) = \frac{\arctan(\frac{x}{5})+\frac{\pi}{2}}{\pi}$, where x is the difference between the outputs of the LSTM neural network. We use this function to normalize all fitnesses to the range $(0, 1)$. Arctangent naturally has limits of $\frac{-\pi}{2}$ and $\frac{\pi}{2}$, so we add $\frac{\pi}{2}$ to shift the fitness up, then divide by π to squash the fitness to less than 1. We divide x by 5 to make the output more uniform across the interval $(0, 1)$. Otherwise, the function approaches 0 and 1 too quickly as x moves further from 0.

5.3.2 Mutations

Section 4.2 contains the theoretical background of these mutations.

Because we are working with music data, there are some specific operations we can perform to act as a type of mutation, as well as crossover.

The six functions we use to operate on `music21 stream` objects are `transpose()`, `inverse()`, `retrograde()`, `retrograde_inverse()`, `inverse_retrograde()`, and `crossover()`.

`transpose(stream, interval)` is simply a wrapper for `music21`'s `Stream.transpose()` method. It accepts a `Stream` and an amount to transpose by, measured in semitones.

`inverse(stream)` returns the inverse of `stream`. That is, it takes the inverts the intervals between notes. For example, the sequence CDE becomes CB♭A♭.

`retrograde(stream, reverse_notes, reverse_rhythms)` returns the retrograde version of `stream`, with the options to reverse the pitches, the rhythms, or both.

`retrograde_inverse(stream)` and `inverse_retrograde(stream)` are wrapper functions for `inverse()` and `retrograde()` together, one for each order.

`crossover(parent1, parent2, crossover_points)` provides the ability to perform crossover of two `Streams`. The function returns two new `Streams`. The first `Stream` we return is obtained by copying notes from `parent1` until the first index in `crossover_points` is reached, then copying notes from `parent2` until the second index in `crossover_points`, and so on, alternating which parent's notes are copied, until there are no more indices in `crossover_points`. The second returned `Stream` is obtained similarly, but starts by copying notes from `parent2`.

These functions are implemented in `geneticAlgorithms/mutations.py`

5.3.3 Evolving Melodies

The file `geneticAlgorithms/geneticAlgorithms.py` contains code to carry out the genetic algorithm that takes some initial population of melodies, remixes and mutates them, and eventually produces some more fit output as measured by the fitness function. It uses all of the previously discussed components to achieve this. See Figure 5.4 for the flow of data through the use of genetic algorithms to generate melodies.

The program starts by creating the fitness function using `lstm.py`. Then the initial population is produced. This can either use `intervalMarkovChain/intervalMarkovChain.py` to generate the initial population, or it can use completely random music to accomplish this. Starting with the Markov chain will generally produce an initial population with a higher fitness, but it will take more time to generate the population. The fitness values of the initial population are calculated while the initial population is being generated.

At this point the program is ready to enter the main loop. Its stopping conditions are when a melody has a high enough fitness or the maximum number of generations has been reached. Inside the loop, the program uses a pool of worker processes to remix the population using the functions inside of `mutations.py`. After this remixing, the loop generates some melodies using crossover, and mutates some of the melodies. This mutation is to ensure the population does not stagnate. At this point the loop calculates the fitnesses of the new population, chooses the seed population for the next generation, and goes back to the beginning of the loop.

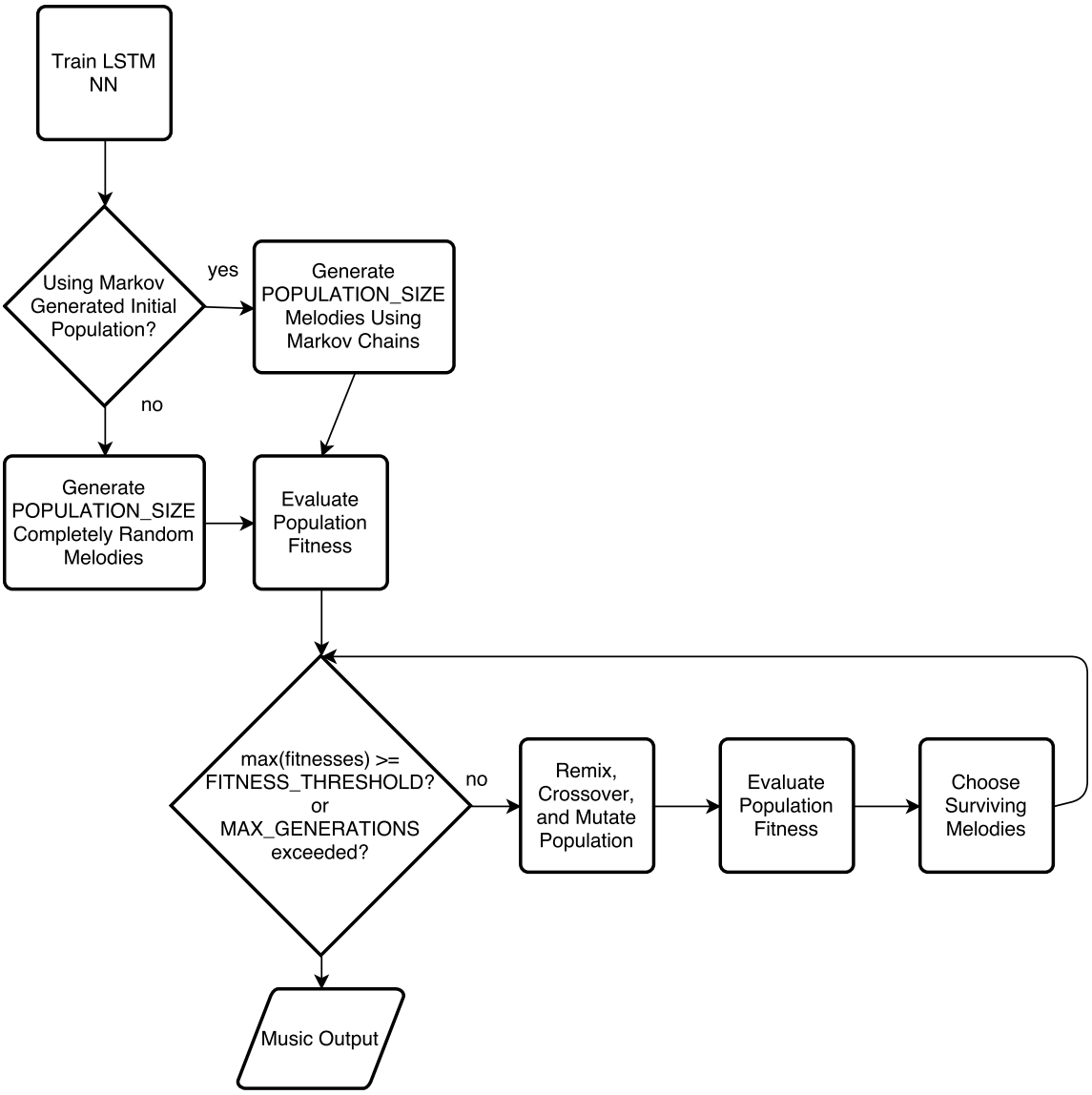

Figure 5.4: The flow of data through the use of genetic algorithms to generate melodies.

5.4 How to Use the Software

5.4.1 Markov Melody

To produce a melody using the Markov Melody program, call the program `intervalMarkovChain.py` along with the corpus of music to use, either directly as the MIDI files to use, or as a directory containing MIDI files. If no corpus is provided, it defaults to any `.mid` files located in the relative directory `../corpus`. From the command line, this might look like

```
python3 intervalMarkovChain.py song1.mid song2.mid
```

when directly specifying the MIDI files to use, or

```
python3 intervalMarkovChain.py
```

to use the default music corpus. Also, the function `generate_melody()`, which handles the creation of melodies inside `intervalMarkovChain.py`, checks if the parameter `corpus` is set to `False`. In the case that it is `False`, the function uses the built in `music21` collection of Bach chorales. This option is only available when importing the file from another Python script.

5.4.2 Genetic Melody

Run `python3 geneticAlgorithm.py` to generate a set of melodies produced by the genetic algorithm. The program accepts some optional flags:

-f, --desired-fitness value

Sets the fitness required to terminate the program to value

-s, --population-size value

Sets the size of the population of each generation to value

-g, --max-generations value

Set the maximum number of generation the program will run to value

-m

Sets the program to use Markov chains to generate the initial population

For example: To use 5000 generations with a population of 500 and a desired fitness of 0.9, with the initial population produced using Markov chains, you could use

```
python3 geneticAlgorithm.py -f 0.9 -g 5000 -s 500 -m
```

or

```
python3 geneticAlgorithm.py --desired-fitness 0.9 --max-generation 5000 --population-size
500 -m
```

CHAPTER 6

Results and Future Work

We found some interesting results and produced some interesting melodies throughout this project. This chapter presents these results, along with some ways in which one might expand upon the project in the future.

6.1 Results

In this section we discuss some of the results from the project, including the output from the fitness function for several pieces of music and the music produced by the software.

6.1.1 The LSTM Neural Network

With the neural network setup described in Section 5.3.1, the network is able to correctly predict whether a set of notes is Bach or not 99% of the time. This is the same whether it is trained on the Bach cello suites or the chorales provided by music21. As described in Section 5.3.1, fitness values are the output of a squashing function with the range $(0, 1)$. The input to the squashing function is the difference between how much the LSTM neural network thinks the melody is good and how much the network thinks it is bad. Table 6.1 contains the fitness values for music by Bach, Domenico Scarlatti, and Franz Joseph Haydn. Two different versions of the fitness function were used; one trained on music21's Bach chorales, and the other on Bach cello suites. As expected, the fast paced violin sonata by Haydn and the piano exercise by Scarlatti performed much better when using the fitness function trained with the cello suites, while the Little Fugue, which contains some longer rhythms performed better when the fitness function trained with the chorales. This is expected because the cello suites are fast paced and the chorales have a slow pace.

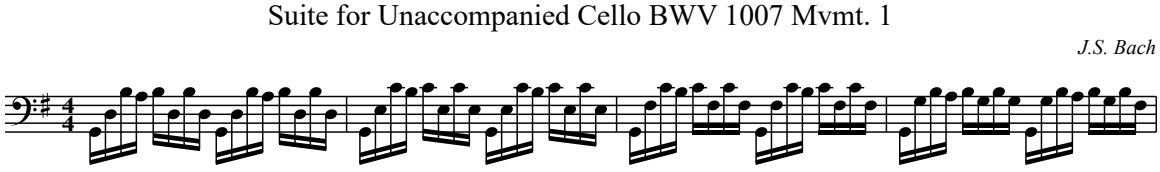

Figure 6.1: Opening of the first movement of the first Cello Suite by J.S. Bach.

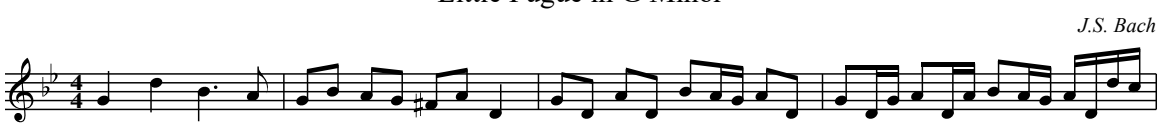

Figure 6.2: Little Fugue in G Minor by J.S. Bach.

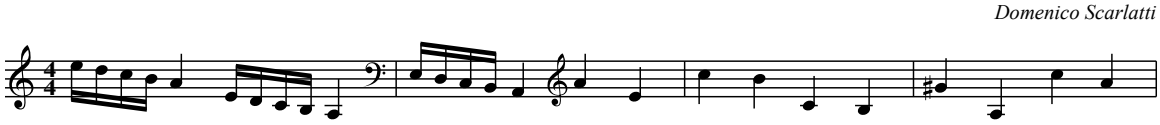

Figure 6.3: Piano Exercise by Domenico Scarlatti.

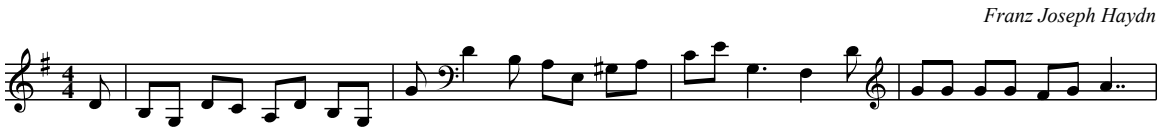

Figure 6.4: Violin Sonata in G Major by Franz Joseph Haydn.

Piece	Fitness (Chorales)	Fitness (Cello Suites)
J.S. Bach Little Fugue in G Minor	0.87171	0.86847
Domenico Scarlatti	0.23755	0.92652
Haydn Violin Sonata	0.77070	0.91706

Table 6.1: Fitness values of music by various authors using models trained with music21's Bach chorales and Bach's cello suites

See Figures 6.1, 6.2, 6.3, and 6.4 for the samples from the pieces that were tested for their fitnesses.

6.1.2 Markov Chains vs Random Notes

Although the initial population's fitness is generally greater when using Markov chains create the melodies, the use of the genetic algorithm quickly negates that difference within a few generations. Figures 6.5 and 6.6 show the top fitness values from the population for the first 500 generation of a run of the program. The first figure's initial population was generated by Markov chains, and the second figure's initial population was randomly generated. The initial fitness and fitness of the first several generations starts higher in the first figure, but after about the fifth generation, the fitness in the second figure matches, and eventually surpasses, that of the first. So the fitness of the randomly seeded population can quickly surpass the fitness of the Markov chain seeded population. Thus, since the time to generate the initial population using Markov chains is considerable, compared to using list comprehensions to create random notes, the Markov chain melodies should probably be relegated to generating one-off melodies.

6.1.3 The Music and Its Uses

Figures 6.7 and 6.8 show two melodies generated using Markov chains. They were generated using a third order Markov chain for the intervals between notes and a second order Markov chain for the rhythms. Of particular interest, the second measure of the second line of Figure 6.8 contains a figure repeated three times. Additionally, the first measure of the last line of that particular melody contains a perfect run down the B♭ Major scale (from B♭5 to B♭4) in its last two beats. With the exception of the *E*6 in the second beat of this measure not being an E♭, it is almost a perfect run down the scale starting on G6 and ending on B♭4.

Figures 6.9 and 6.10 contain melodies that were generated using random order Markov chains, meaning the order to determine intervals between notes and the order to determine rhythm were selected at random from the range 1 through 5. Something of interest that occurs in Figure 6.9 is that after the initial four notes, which were provided by the author, the rest of the melody is almost in C

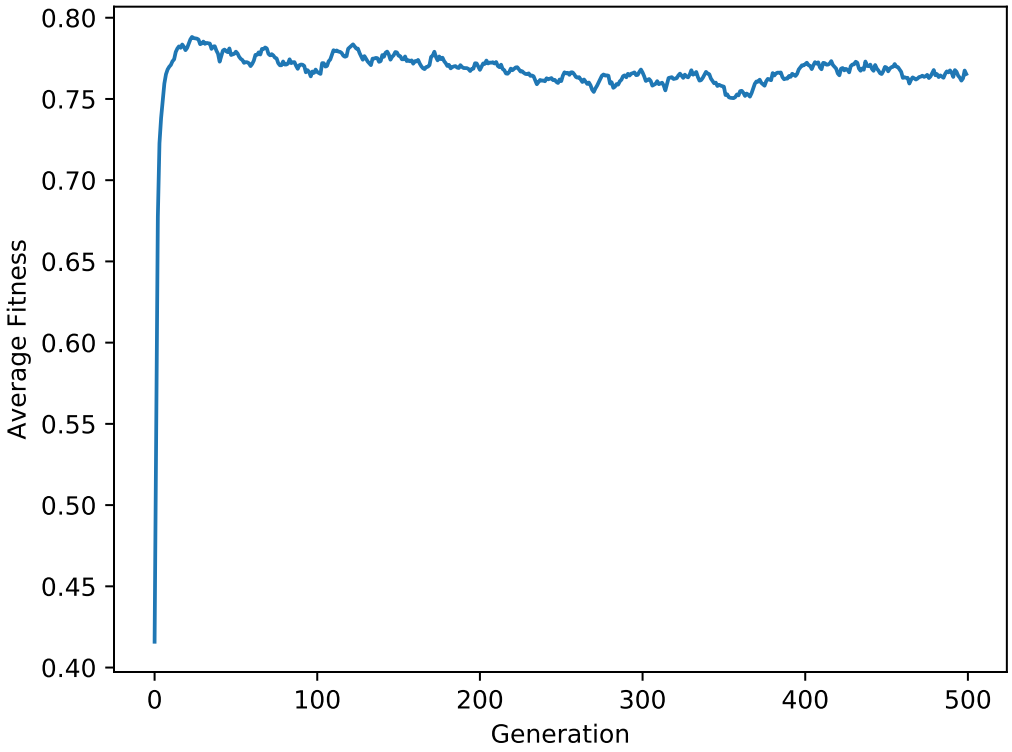

Figure 6.5: Fitness over 500 generations with an initial population generated using Markov chains.

Minor. In general we expect most of the pieces produced to tend toward C Major or A Minor, as those are the keys the Markov model was trained on. Even though the melody shifted from the expected key, it manages to maintain coherence because the music the Markov chain was trained with stay in one key, and so too does the produced melody.

The melodies is Figures 6.11 and 6.12 were generated using the genetic algorithm with the surrogate fitness function trained using the chorales from music21's built in corpus of Bach music and an initial population of randomly generated melodies. Of particular interest in Figure 6.11 is the set of arpeggiations in the third and fourth measures. Without explicitly telling the algorithm anything about arpeggios or repeating ideas in slightly differing ways, it picked up that this makes for good (Bach-like) music.

An application of this project is to provide inspiration for the composer. See Figure 6.13 for an example where the author used the melody from Figure 6.11 and added a bass line and more satisfying ending. The generated melodies could also be chained together to create longer compositions, or they could have their rhythms increased by some multiple to have more than a snippet of music.

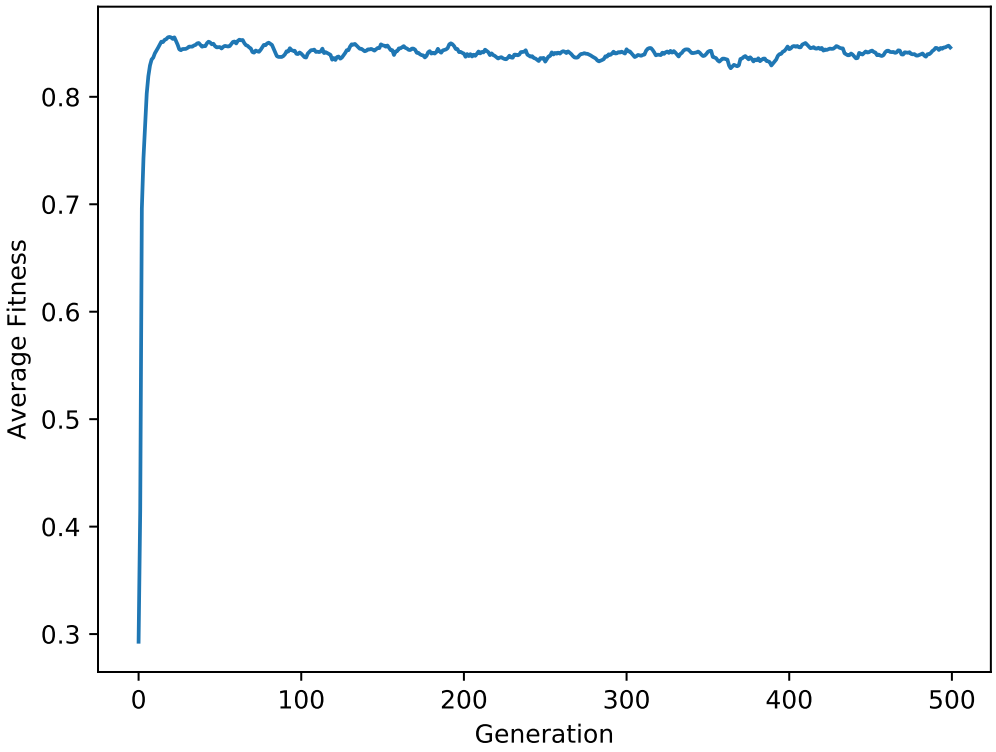

Figure 6.6: Fitness over 500 generations with a random initial population.

The ambitious musician might use these melodies to practice their sight reading; it is extremely unlikely that a melody produced by either the Markov chains or genetic algorithm has even been written before, let alone seen by a particular musician. Therefore, these melodies could provide a great opportunity to practice reading music seen for the first time.

6.2 Future Work

The work in this project could be expanded by writing code to generate counterpoint or harmonizations of melodies that are created through this project. One possible method to do this might involve simply using music21's ability to create a *Roman numeral analysis* (RNA) of a piece to determine which chords to use. Then, the software could fill in the bass, tenor, and alto parts with notes that fit the RNA. This would likely create very simple, possibly boring, harmonies that may not follow all the voice leading rules of Western music. But it would create *a* complete harmonization.

Many authors have written about the topic of automatic harmonization. Interestingly, much

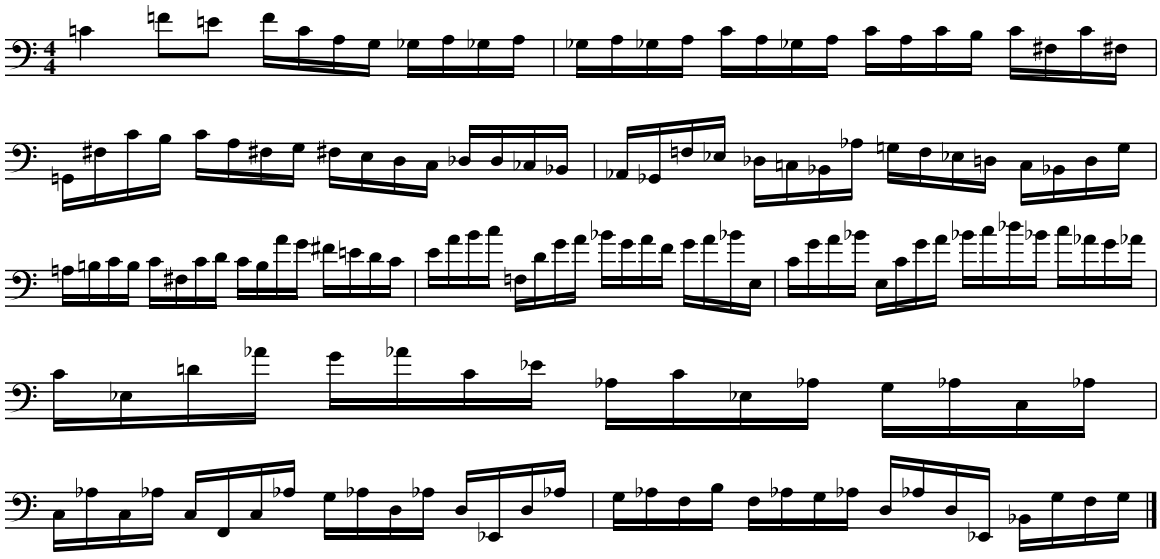

Figure 6.7: A melody generated by Markov chains.

of the research on this topic focuses on using genetic algorithms to accomplish the task. William Schottstaedt [30] defines a genetic algorithm to generate multi-voice music that follows the rules of the Common Practice era. Somnuk Phon-Amnuaisuk and Geraint A. Wiggins [28] use genetic algorithms to harmonize preexisting soprano parts. Andres Acevedo [1] uses genetic algorithms to write counterpoint in a fugue setting. A fugue is a musical form in which an initial theme is introduced in one part, that is imitated in other parts and repeated throughout the piece. There are many other examples of using genetic algorithms to solve harmonization and counterpoint; these are just a small sample of the approaches explored.

Approaches other than using genetic algorithms also exist for creating pleasing counterpoint. For example, David Cope discusses a rule-based approach to writing counterpoint [10]. In this method, a set of rules about how to compose is provided to the program, rather than the program learning from the music on which it is based. In another approach, Kamil Adiloglu and Ferda Alpaslan [2] use a feed forward neural network to accept a single voice melody and produce a second voice to complement the first one.

Along with adding counterpoint or harmony generation, one might take different approaches to produce melodies. For example, Chun-Chi Chen and Risto Miikkulainen [8] combine neural networks and genetic algorithms by using an evolutionary algorithm to find neural networks that produce good melodies.

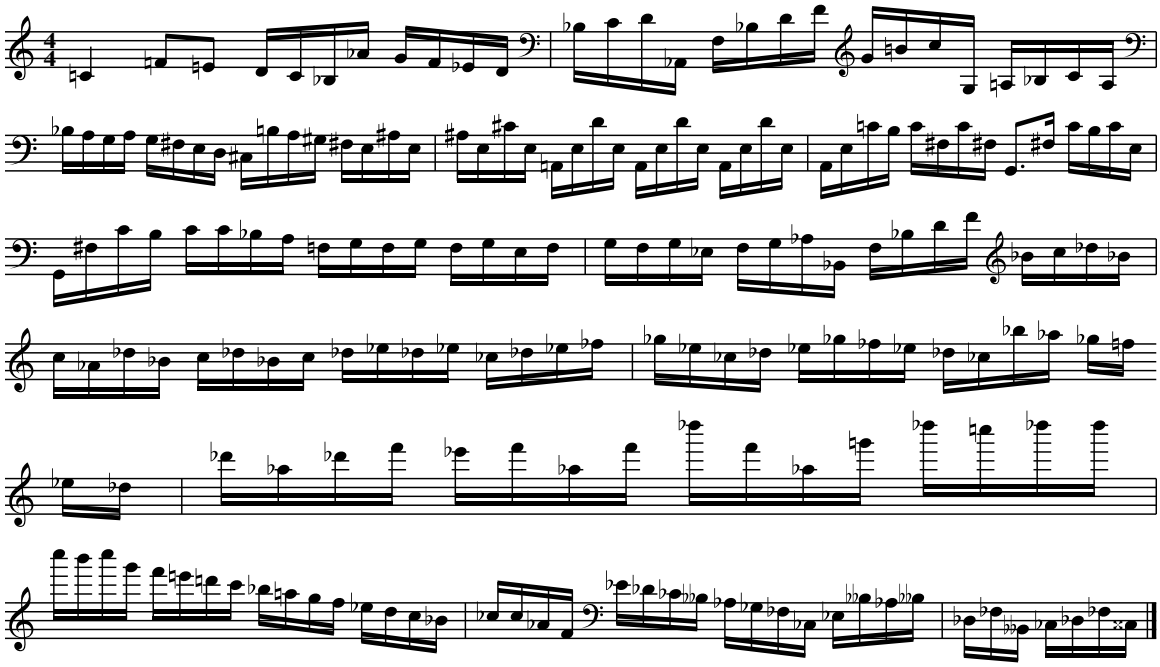

Figure 6.8: A melody generated by Markov chains.

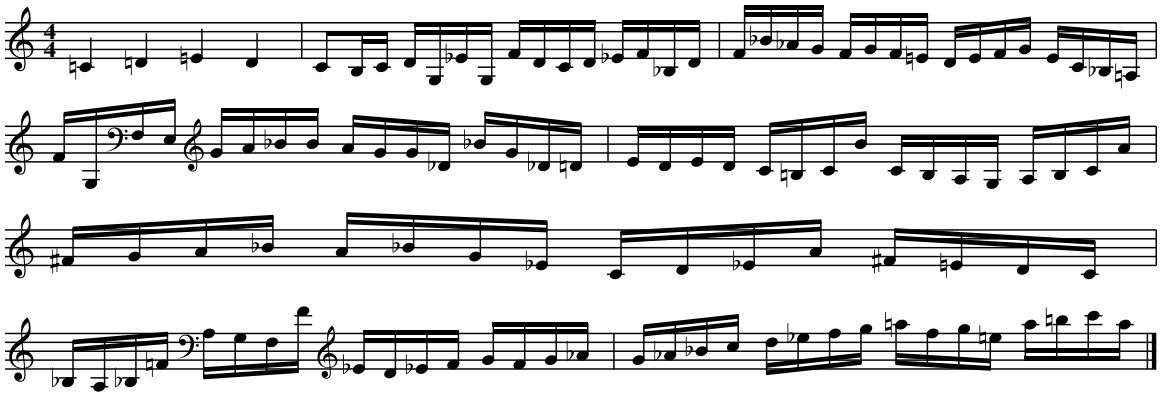

Figure 6.9: A melody generated by random order Markov chains.

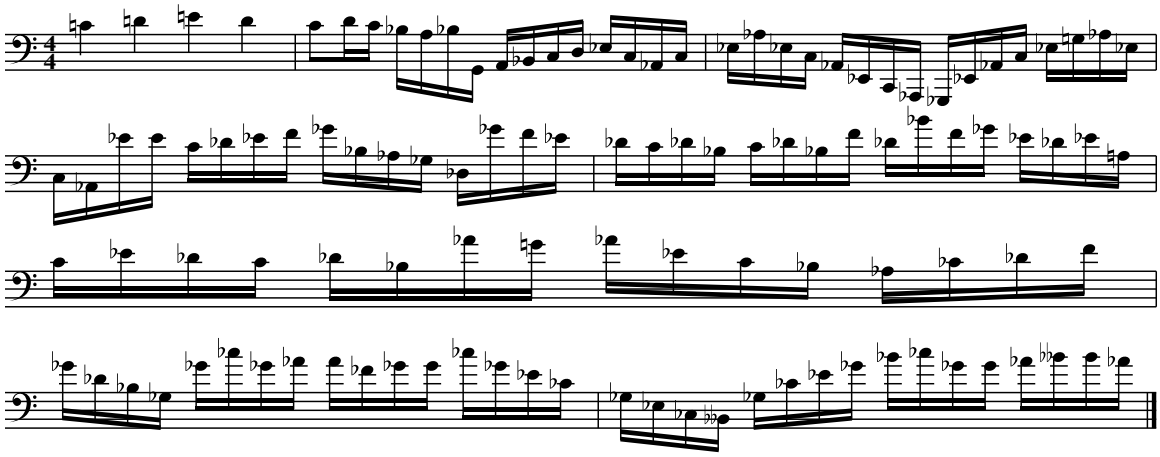

Figure 6.10: A melody generated by random order Markov chains.

Figure 6.11: A melody generated with the genetic algorithm.

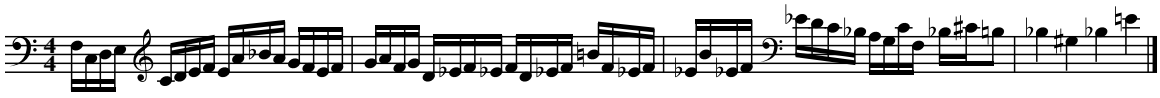

Figure 6.12: A melody generated with the genetic algorithm.

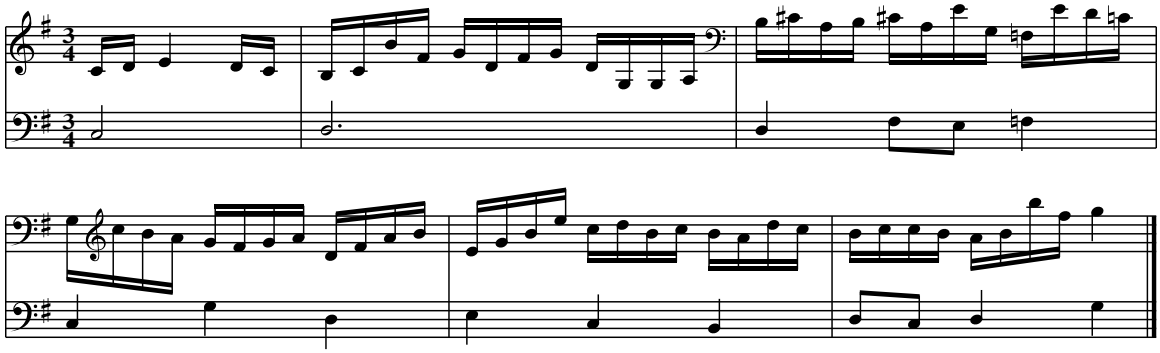

Figure 6.13: The melody in Figure 6.11 with a bass line written by the author.

References

1. Andres Garay Acevedo. Fugue composition with counterpoint melody generation using genetic algorithms. In *International Symposium on Computer Music Modeling and Retrieval*, pages 96–106. Springer, 2004. 36

2. Kamil Adiloglu and Ferda N. Alpaslan. A machine learning approach to two-voice counterpoint composition. *Knowledge-Based Systems*, 20(3):300–309, April 2007. ISSN 0950-7051. doi: 10.1016/j.knosys.2006.04.018. URL http://www.sciencedirect.com/science/article/pii/S0950705106001468. 36

3. Manuel Alfonesca, Manuel Cebrián, and Alfonso Ortega. A Fitness Function for Computer-Generated Music using Genetic Algorithms. *WSEAS Transactions on Information Science and Applications*, 3(3):518–525, 2006. URL https://repositorio.uam.es/bitstream/handle/10486/664728/fitness_alfonseca_WTISA_2006.pdf.

4. Adam Alpern. Techniques for Algorithmic Composition of Music. *Hampshire College*, 1995. URL https://pdfs.semanticscholar.org/d701/dd6cdc82ed544422c553dab59426f759d558.pdf.

5. Roger Alsop. Exploring the Self Through Algorithmic Composition. *Leonardo Music Journal*, 9:89–94, 1999. ISSN 0961-1215. doi: 10.2307/1513482. URL http://www.jstor.org/stable/1513482.

6. Torsten Anders and Eduardo R. Miranda. Constraint Programming Systems for Modeling Music Theories and Composition. *ACM Comput. Surv.*, 43(4):30:1–30:38, October 2011. ISSN 0360-0300. doi: 10.1145/1978802.1978809. URL http://0-doi.acm.org.dewey2.library.denison.edu/10.1145/1978802.1978809.

7. Christopher Ariza. Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition. *Computer Music Journal*, 35(3):40–56, 2011. ISSN 0148-9267. doi: 10.2307/41241764. URL http://www.jstor.org/stable/41241764.

8. C.-CJ Chen and Risto Miikkulainen. Creating melodies with evolving recurrent neural networks. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3, pages 2241–2246. IEEE, 2001. 36

9. Florian Colombo, Samuel P. Muscinelli, Alexander Seeholzer, Johanni Brea, and Wulfram Gerstner. Algorithmic Composition of Melodies with Deep Recurrent Neural Networks. *arXiv:1606.07251 [cs, stat]*, June 2016. doi: 10.13140/RG.2.1.2436.5683. URL http://arxiv.org/abs/1606.07251. arXiv: 1606.07251.

10. David Cope. *Computers and Musical Style*. Number v. 6 in The Computer Music and Digital Audio Series. A-R Editions, Madison, Wis, 1991. ISBN 978-0-89579-256-3. 36

11. David Cope. *Experiments in Musical Intelligence*. Number v. 12 in The Computer Music and Digital Audio Series. A-R Editions, Madison, Wis, 1996. ISBN 978-0-89579-314-0.

12. David Cope. *The Algorithmic Composer*. Number v. 16 in The Computer Music and Digital Audio Series. A-R Editions, Madison, Wis, 2000. ISBN 978-0-89579-454-3.

13. David Cope. *Virtual Music: Computer Synthesis of Musical Style*. MIT Press, Cambridge, Mass, 2001. ISBN 978-0-262-03283-4.

14. Alan R.R. de Freitas and Frederico Gadelha Guimarães. Originality and Diversity in the Artificial Evolution of Melodies. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 419–426, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0557-0. doi: 10.1145/2001576.2001634. URL http://0-doi.acm.org.dewey2.library.denison.edu/10.1145/2001576.2001634. 18

15. Kemal Ebcioğlu. An Expert System for Harmonizing Four-Part Chorales. *Computer Music Journal*, 12(3):43–51, 1988. ISSN 0148-9267. doi: 10.2307/3680335. URL http://www.jstor.org/stable/3680335.

16. Douglas Eck and Jüsrgen Schmidhuber. Learning the Long-Term Structure of the Blues. In *Artificial Neural Networks – ICANN 2002*, Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. ISBN 978-3-540-44074-1 978-3-540-46084-8. doi: 10.1007/3-540-46084-5_47. URL https://link.springer.com/chapter/10.1007/3-540-46084-5_47.

17. David Fernández and Francisco Vico. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013. URL http://www.jair.org/media/3908/live-3908-7454-jair.pdf.

18. Iain Foxwell and Don Knox. Composing with Algorithms: Two Novel Generative Composition Tools. In *Proceedings of the 7th Audio Mostly Conference: A Conference on Interaction with Sound*, AM '12, pages 76–81, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1569-2. doi: 10.1145/2371456.2371468. URL http://0-doi.acm.org.dewey2.library.denison.edu/10.1145/2371456.2371468.

19. Sepp Hochreiter and Jürgen Schmidhuber. Long Short-term Memory. *Neural computation*, 9: 1735–80, December 1997. doi: 10.1162/neco.1997.9.8.1735.

20. Andrew Horner and David Goldberg. Genetic Algorithms and Computer-Assisted Music Composition. *Urbana*, 51:437–441, 1991. URL https://www.researchgate.net/profile/Andrew_Horner/publication/252520139_Genetic_Algorithms_and_Computer-Assisted_Music_Composition/links/571875be08aed43f63221228.pdf.

21. Cheng-Zhi Huang, Tim Cooijmans, Adam Roberts, Aaron Courville, and Douglas Eck. Counterpoint by Convolution. 2016. URL https://ismir2017.smcnus.org/wp-content/uploads/2017/10/187_Paper.pdf.

22. Jeremy Leach and John Fitch. Nature, Music, and Algorithmic Composition. *Computer Music Journal*, 19(2):23–33, 1995. ISSN 0148-9267. doi: 10.2307/3680598. URL http://www.jstor.org/stable/3680598.

23. Max V. Mathews and John R. Pierce, editors. *Current directions in computer music research*. Number 2 in System Development Foundation benchmark series. MIT Press, Cambridge, Mass, 1989. ISBN 978-0-262-13241-1.

24. Mike McKerns. better multiprocessing and multithreading in python, February 2018. URL https://github.com/uqfoundation/multiprocess. original-date: 2015-06-20T15:45:09Z.

25. Matt McVicar, Satoru Fukayama, and Masataka Goto. AutoGuitarTab: Computer-aided Composition of Rhythm and Lead Guitar Parts in the Tablature Space. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 23(7):1105–1117, July 2015. ISSN 2329-9290. doi: 10.1109/TASLP.2015.2419976. URL http://0-dx.doi.org.dewey2.library.denison.edu/10.1109/TASLP.2015.2419976. 18

26. Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. Springer Science & Business Media, August 2009. ISBN 978-3-211-75540-2. Google-Books-ID: jaowAtnXs-DQC. 5

27. George Papadopoulos and Geraint Wiggins. AI methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity*, pages 110–117. Edinburgh, UK, 1999. URL http://www.academia.edu/download/3433841/AI_Methods_for_Algorithmic_Composition_A_Survey_a_Critical_View_and_Future_Prospects.pdf. 18

28. Somnuk Phon-Amnuaisuk and Geraint Wiggins. The four-part harmonisation problem: a comparison between genetic algorithms and a rule-based system. In *Proceedings of the AISB'99 Symposium on Musical Creativity*, pages 28–34. AISB London, 1999. 36

29. Luke Prudente and Andrei Coronel. Towards Automated Counter-Melody Generation for Monophonic Melodies. pages 197–202. ACM Press, 2017. ISBN 978-1-4503-4828-7. doi: 10.1145/3036290.3036295. URL http://dl.acm.org/citation.cfm?doid=3036290.3036295.

30. William Schottstdaet. Automatic Counterpoint. In *Current Directions in Computer Music Research*, pages 199–214. The MIT Press, Cambridge, Massachusetts, 1989. 36

31. Peter M. Todd. A Connectionist Approach to Algorithmic Composition. *Computer Music Journal*, 13(4):27–43, 1989. ISSN 0148-9267. doi: 10.2307/3679551. URL http://www.jstor.org/stable/3679551. 19

32. Derek Wells and Hala ElAarag. A Novel Approach for Automated Music Composition Using Memetic Algorithms. In *Proceedings of the 49th Annual Southeast Regional Conference*, ACM-SE '11, pages 155–159, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0686-7. doi: 10.1145/2016039.2016083. URL http://0-doi.acm.org.dewey2.library.denison.edu/10.1145/2016039.2016083.