Journal of Humanistic Mathematics

Volume 11 | Issue 2 July 2021

Markov Chains for Computer Music Generation

Ilana Shapiro
Pomona College

Mark Huber
Claremont McKenna College

Markov Chains for Computer Music Generation

Ilana Shapiro

Pomona College, California, USA
issa2018@mymail.pomona.edu

Mark Huber

Department of Mathematical Sciences, Claremont McKenna College, California, USA
mhuber@cmc.edu

Abstract

Random generation of music goes back at least to the 1700s with the introduction of Musical Dice Games. More recently, Markov chain models have been used as a way of extracting information from a piece of music and generating new music. We explain this approach and give Python code for using it to first draw out a model of the music and then create new music with that model.

Keywords: randomized algorithms, musical dice games, music composition, Markov chains

1. Introduction

Randomness has long been used in the generation of music. One of the first methods for randomized music composition, called *Musikalisches Würfelspiel* (*Musical Dice Games*), arose in the 18$^{\text{th}}$ century. These games were based off the observation that in any piece of music, individual notes of music are combined into *measures* (or *bars*), each of which has a fixed length. They work by deciding what an entire bar will sound like at once.

The initial musical dice game was created in 1757 by Johann Philipp Kirnberger, who published a method [2] for composing a *polonaise* in *minuet and trio* form. This is an example of a musical form called *ternary* because it consists of three parts. The first and third parts are the same eight bars,

called the *minuet*. The middle part is called the *trio*. A simple way to represent this structure is to write ABA, where section A is the minuet and section B is the trio. Each section is eight bars long.

So to create such a musical piece, it was necessary to write down the minuet part (section A) and the trio part (section B). Rather than generate one section at a time, in Kirnberger's game the first measures of both section A and B were generated, then the second measures, and so on until all eight measures were complete.

For a particular measure, the procedure for generating the corresponding minuet measure and the trio measure worked as follows. One would roll two fair six-sided dice, and label the results X_1 and X_2. X_1 was then used in a look-up table to determine the content of that measure of the minuet, and X_2 was used in a different look-up table to determine the content of that measure of the trio. Figure 1 shows a table from a 1767 edition [3] of Kirnberger's work. "Premiere partie" indicates the minuet, and "seconde partie" indicates the trio.

For instance, if bar 4 of the minuet were under construction, and the first roll were a 4, then based on Kirnberger's encoding, bar 4 of the minuet (section A) would use piece 74. Similarly, if the second roll were a 2, then bar 4 of the trio (i.e., section B) would use piece 39.

Table pour un Menuet avec un Dé.

	Premiere Partie.							Seconde Partie.					
	1	2	3	4	5	6		1	2	3	4	5	6
1 Jet	23	63	79	13	43	32	1 Jet	33	55	4	95	38	44
2 - -	77	54	75	57	7	47	2 - -	60	46	12	78	93	76
3 - -	62	2	42	64	86	84	3 - -	21	88	94	80	15	34
4 - -	70	53	5	74	31	20	4 - -	14	39	9	30	92	19
5 - -	29	41	50	11	18	22	5 - -	45	65	25	1	28	17
6 - -	83	37	69	3	89	49	6 - -	68	6	35	51	61	10
7 - -	59	71	52	67	87	56	7 - -	26	91	66	82	72	27
8 - -	36	90	8	73	58	48	8 - -	40	81	24	16	85	96

Figure 1: Table for using die rolls to construct bars of a minuet and trio. Scan from https://imslp.org/wiki/File:PMLP243537-kirnberger_allezeit_fertiger_usw.pdf.

Each roll of the dice for each bar of the result determines a different piece. There are sixteen distinct bars in total in sections A and B, and each bar has six possibilities (since one die is rolled per bar); hence, this game can theoretically produce $6^{16} = 2.82110991 \times 10^{12}$ different musical compositions.

However, these dice games are greatly restricted in that they rely on a composer that has already created the possible bars to be put together. In other words, the player is merely piecing together already composed music in new ways.

That leaves open the following question: how does one randomly create the individual notes that comprise the bars?

One such approach is to model music using Markov chains, which opens doors to computationally composing arbitrarily long and fully-fledged compositions.

1.1. Using Markov chains

In the musical dice game, the bar choices were independent. However, this is a bad idea for note generation. If the notes are changing too rapidly, and each note is independent of the preceding note, the result is more likely to be cacophony than music.

A solution comes with the use of *Markov chains*. A Markov chain is a sequence of random variables $X_1, X_2, X_3, \ldots$ such that the distribution of X_{t+1} conditioned on $X_1, \ldots, X_t$ only depends on X_t, and not on the values of $X_1, \ldots, X_{t-1}$. Markov chains were introduced in 1906 by Andrey Markov [5] as a way of understanding which letters follow others in a typical text.

In this paper, Markov chains are used to determine the sequence of notes (both in pitch and duration). The distribution of the type of the next note will only depend on the current note, and not on any of the notes that came before.

The first use of Markov chains to compose music came in 1957, when the ILLIAC I computer was used to compose the *Illiac Suite* by Hiller and Isaacson [7]. Since then, Markov chains have been a simple tool for automatically generating a new piece of music. The Markov chains employed by Hiller and Isaacson dealt purely with horizontal melody; in this paper, we endeavor to incorporate harmony and rhythm as well.

In the past sixty years since Hiller and Isaacson, Markov chains have become increasingly popular as a means of music generation. Ramanto and Maulidevi (2017) [6] employed Markov chains for *procedural content generation* of music, in which music is randomly generated in response to a user-inputted mood. Rather than generating music in the style of an existing piece as this paper seeks to, they sought to generate music in the style of a certain mood. Linskens (2014) [4] also employed Markov chains for algorithmic music *improvisation*, rather than composition. The Markov chains were trained on an existing piece, like they are in this paper, but then the algorithm was given a certain amount of freedom to vary between the notes of a designated chord or even an unspecified pitch lying somewhere in the bounds of a chord in order to achieve the improvisation quality. This paper does not explore improvisation, though this is certainly an interesting avenue.

Others, such as Yanchenko and Mukherjee (2018) [8] have used more complex statistical models such as *time series models*, which are variations on *Hidden Markov Models* (HMMs). With the Hidden Markov Model, instead of generating a sequence of states, each state omits an *observable*, and the states themselves are hidden. The idea here is to use techniques such as dynamic programming to backtrack from the generated observables in order to determine the optimal sequence of hidden states that generated these observables. Kathiresan (2015) [1] also employs HMMs to generate music against a variety of constraints with the goal of making it sound as subjectively "musical" as possible. This paper does not delve into HMMs, as the aim is to experiment with the musical capabilities of simple Markov chains, but this may certainly be an interesting avenue for future exploration.

The rest of the paper is organized as follows. In the next section (§2), we describe the terminology of Markov chains in more detail. The subsequent sections show how to estimate the parameters of the chain (§3-§4), and then finally a new piece of music is built from an existing piece using these estimates (§5). §6 contains the results of our work and §7 concludes this paper.

2. Theoretical Foundations of Markov Chains

Consider a sequence of random variables $X_1, X_2, X_3, \ldots$. Such a set of random variables $\{X_i\}$ forms a *stochastic process*. The index i in X_i is often called the *time*. For a fixed time i, X_i is called the *state* of the chain.

A Markov chain is a stochastic process such that for all i, it holds that

$$[X_i \mid X_1, \ldots, X_{i-1}] \sim [X_i \mid X_{i-1}].$$

Here the $\sim$ symbol means that the left hand side and the right hand side have the same distribution. In words, this says that the distribution of the i^{th} state in the sequence, given the values of all the states that came before, will depend purely on the previous state. The most common type of Markov chain is also *time homogeneous*, which means that for all i it holds that $[X_i \mid X_{i-1}] \sim [X_2 \mid X_1]$. In other words, the random way in which the state evolves does not change as the time changes.

A Markov chain is also called a *memoryless process*, since the next state depends purely on the current state, and not on the memory of the notes that came before. In order to describe a time homegeneous Markov chain, it is necessary to know what values the random variables X_i can take on, and what the probabilities are for moving to the next state.

2.1. Representing Markov chains

Markov chains can be represented in a variety of ways. One helpful way is to represent them graphically with a *directed graph*. This is a collection of nodes and edges, in which each edge has a direction from one node to another. Each node represents a state in the Markov chain, and each edge has a probability associated with it that represents the probability the source node will transition to the destination node (the source and destination node can be the same, which means the process remains in the same state). All the probabilities associated with the edges extending from a node must sum to one (if any edges are omitted, it is assumed that they represent a transition with probability zero).

An example of a graphical representation of a simple Markov chain is shown in Figure 2. The states $\{\text{Sunny}, \text{Windy}, \text{Rainy}\}$ represent weather, and the probabilities of moving between the three states are above the arrows.

An alternate way of encoding the Markov chain is with the *transition matrix*, where the $(a, b)^{th}$ entry of the matrix is the probability of moving from state a to state b.

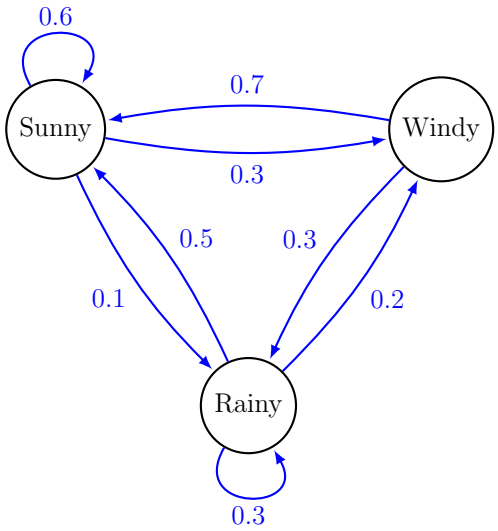

Figure 2: Markov Chain graph example.

The transition matrix for the Markov chain from Figure 2 is as follows.

$$\begin{array}{c} \\ \text{Sunny} \\ \text{Windy} \\ \text{Rainy} \end{array} \begin{array}{c} \begin{array}{ccc} \text{Sunny} & \text{Windy} & \text{Rainy} \end{array} \\ \begin{pmatrix} 0.6 & 0.3 & 0.1 \\ 0.7 & 0 & 0.3 \\ 0.5 & 0.2 & 0.3 \end{pmatrix} \end{array}$$

Note that all rows in the transition matrix (or equivalently the probabilities on all outgoing edges) must sum to 1.

3. Using Markov Chains to Generate Music

Now consider a given piece of music, which we will call the *training data*. This data will be used to estimate the probabilities for our Markov chain. This chain can then be used to generate music in the same style as the original piece.

In order to generate music, we want the nodes in the Markov chain, or the set of states, to represent *sound objects*. These are entities that represent a single note or chord and contain information about its pitch(es), octave(s), and duration. Thus, each node will contain data about the single note name or collection of notes in the chord, using note names A through G; the accidental

(sharp, flat, or natural) for each note, represented by #, b, or no symbol, respectively; the octave for each note, represented by an integer from 0-8; and the duration of the sound object, denoted by a whole note, half note, quarter note, or some shorter value.

One additional special case will be accounted for: the rest, where no sound is played. A rest will be indicated by R. Rests also have a duration.

The set of states will be determined by parsing the piece of music. This process will be discussed shortly, in §4.

We can define our transition matrix by determining the probability for each pair of sound objects s_1 and s_2, i.e., the chance of moving from s_1 to s_2 in the chain. In addition, we would also like to define an initial probability vector I. This vector gives us the probability that for each state s_i, the initial state in the chain X_1 will be equal to s_i.

For example, consider the Markov chain represented graphically in Figure 3 that consists of only three sound objects. Note that ♩ represents a quarter note, and ♪ represents an eighth note.

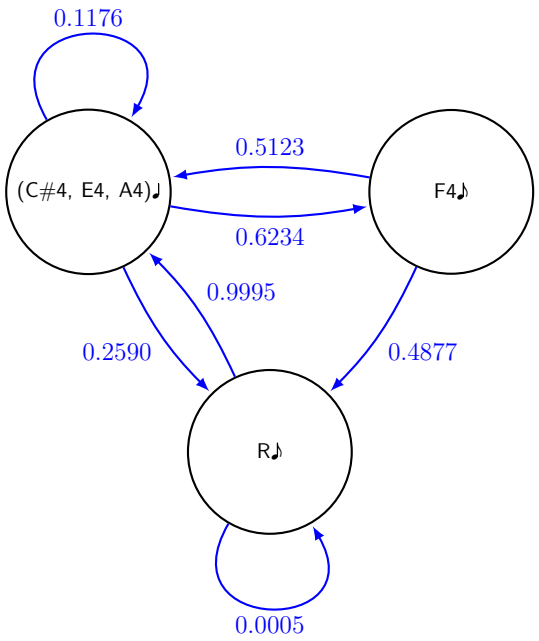

Figure 3: Musical Markov Chain example using graphical representation.

The set of states in the example is $S = \{$(C#4, E4, A4)♩, F4♪, R♩$\}$. The first state is a chord that consists of three notes — C#, E4, and A4 — and lasts for a duration of one quarter note. The second state is a single note — F4 — with a duration of one eighth note, and the final state is a rest with a duration of one quarter note.

The transition matrix M representation of this chain is shown in Figure 4.

State	(C#4, E4, A4)♩	F4♪	R♩
(C#4, E4, A4)♩	0.1176	0.6234	0.2590
F4♪	0.5123	0.0000	0.4877
R♩	0.9995	0.0000	0.0005

Figure 4: Musical Markov Chain example using a transition matrix.

In order to generate a new piece of music, it is necessary to choose a sound object to start with in our generation. We could simply pick the first sound object in the training data, or we could create an *initial probability vector* that tells us the probability that each sound object is encountered. We will choose to do the latter and determine our initial probability vector by finding how many total sound objects there are in the piece (including repetitions) and the number of times each individual sound object appears.

To see how this works, suppose that in the training data, (C#4, E4, A4)♩ appears twice, F4♪ appears three times, and R♩ appears once. Then the resulting initial probability vector I is shown in Figure 5.

(C#4, E4, A4)♩	F4♪	R♩
0.3333	0.5000	0.1667

Figure 5: Musical Markov Chain Example

We now have the tools to generate music from the Markov chain in the same style as the training data.

4. Parsing the Training Data

Now we move from theory to practice: using a computer, how can we find the probabilities for our Markov chain, and then simulate a musical score using this chain? The Python language will be used for this exploration.

Throughout this section, please refer to the file `parse_musicxml.py` in Appendix A.1 for the full Python code. Alternatively, the code as well as the examples are available at the following link:

https://github.com/ilanashapiro/Markov-Model-Music-Generation

There are additional examples included in the GitHub repository; the ones discussed in this paper are in the files named "`Cantabile_flute_excerpt`" and "`Cantabile_piano_excerpt`" (`.musicxml` extension for the original version and `.mid` extension for the generated version).

The training data (the input musical piece) is given in a symbolic form called *MusicXML*. It is a file format that encodes musical notation based on *extensible markup language* (hence the xml ending of MusicXML).

We will use Python's ElementTree library to parse the MusicXML file and the NumPy library to build and manipulate matrices in a class called `Parser`. This class will be used later in the runner file `generate.py` to parse the input MusicXML files. `Parser`'s constructor initializes some important information, such as filename, transition matrix, initial probability vector, and states (the sound objects we will obtain from the input piece).

We initially obtain the data that allows us to build our transition matrix. All sound objects (whether they are chords or individual notes) are extracted sequentially from the MusicXML file and stored in an ordered dictionary alongside the number of times each one appears in the piece. A sound object is uniquely identified by its note(s), accidental, octave, and duration. At this time, we also simultaneously save each sound object in an ordered list (the set of states) in the order it appears. Note that this ordered list, as it represents a set, does not contain repetitions. This process ensures that the sound object dictionary and the list of states contain the same data in the same order, which will allow us to successfully create our transition matrix.

From the dictionary, the transition matrix is created using NumPy. If the length of the list of states (i.e. the number of sound objects) is n, then the transition matrix has dimensions $n \times n$. Both the row and column order correspond to the order of the state dictionary and the list of states. The transition matrix is built as follows:

1. Using NumPy, the matrix is initialized to the known dimensions $n \times n$. Next, the matrix is built row by row.

2. Each entry i, j in the matrix is initialized to represent the number of times the i^{th} sound object transitions to the j^{th} sound object in the list of states. At this point, the matrix is symmetric.
3. Once all n^2 entries have been initialized, NumPy is used to divide all the elements in each row by the row sum.
4. Finally, for each row, each entry is replaced with the sum of all the previous entries using NumPy's `cumsum` function. This means that the first element in each row will retain the value from the previous step, and all subsequent values will be sequentially greater. Note that because of what we did in the previous step, by applying `cumsum` we ensure that the final value in each row is now 1.

Imagine the i^{th} row representing a line, and each i, j entry representing a segment on that line. The i, j entry that corresponds to the longest line segment is the entry corresponding to some sound object (i.e. state) j that sound object i has the highest probability of transitioning to. This process to transform the data into the line analogy is also known as *inverse transform sampling*.

The initial probability vector is built in a similar way. This vector has dimensions $1 \times n$, since we simply want to know the probability that each of n sound objects is chosen at random. We therefore build the initial probability vector as follows:

1. Using NumPy, a matrix is initialized to the known size of $1 \times n$ (i.e. one row of length n).
2. The i^{th} entry in the matrix (i.e. the initial probability vector) is initialized to represent the number of times the i^{th} note in the list of states appears in the piece.
3. Once all n entries have been initialized, NumPy is used to divide all the elements in each row by the row sum.
4. Finally, NumPy's `cumsum` function is used to replace each entry in the single row of this matrix with the sum of all the previous entries. This means that the first element will retain the value from the previous step, and all subsequent values will be sequentially greater. Note that because of what we did in the previous step, by applying `cumsum` we ensure that the final (i.e. n^{th}) value is now 1.

The line segment analogy applies here exactly the same way as before.

A final thing to note is that the last note in the piece is assigned a transition to a quarter rest, and a transition is then added from the quarter rest to the first note in the piece. This ensures that the Markov model contains no *absorbing* states, or states that once entered, cannot be exited.

5. Generating New Music

[Throughout this section, please refer to the file `generate.py` in Appendix A.2.]

Now that we have a working parser that initializes all the elements we need for our Markov chain, we are ready to generate new music in the style of the training data.

We now a create file called `generate.py` and import our parser file (`parse_musicxml.py`). We can instantiate the Parser class to create Parser objects (i.e., create Markov chains) for however many songs we want, so long as we have the corresponding MusicXML files. In the code attached here, four parsers are created in a list. This allows us to loop through the Parser objects in the list and generate music for the Markov chain that each represents.

In order to generate music from a Markov chain, we start by using NumPy to generate a random number from 0 to 1 inclusive. This is known as a *standard uniform* random variable. Now consider the initial probability vector. To use our generated standard uniform to draw from this vector, think of the generated random number as being a point on the line segment that is the result of inverse transform sampling having been applied to this vector. This can be visualized in Figure 6.

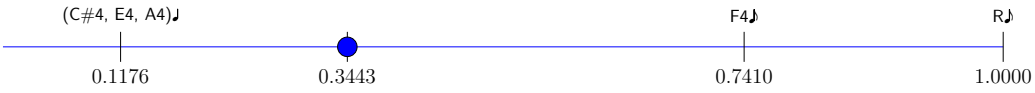

Figure 6: Inverse Transform Sampling example: the blue dot is a uniform draw from 0 to 1, the value of 0.3443 indicates that the draw is F4♩ since that is the next label to the right of the dot.

We will choose the next highest state (i.e., sound object) compared to the randomly chosen point we generated. In this example, our randomly generated point would give us the sound object C3♪. This allows us to choose the initial state of the Markov model.

We then generate a sequence of states (i.e., our generated music) from the model starting at this initial state. We follow the same method as above for choosing the next state to transition to, except we now use the transition matrix instead of the initial probability vector. The length of the sequence is determined by the user's input. In the code in the Appendix, the length chosen is 100 notes.

After generating the sequence of sound objects, the sequence is written out to a MIDI file, which is then loaded into the symbolic music software MuseScore for viewing and playing.

6. Results of the Music Generation

The generated music in this paper results from Markov chains trained on excerpts from Shapiro's composition *Cantabile* for flute and piano. In order to obtain these excerpts, the flute and piano parts were separated, and a short passage was taken from each in order to demonstrate a monophonic example (i.e., the solo flute part), and an example with harmony/chords (i.e., the piano part). The results of the music generated from these parts using their respective Markov chains are as follows:

Figure 7 below contains the original flute part (that is, the training data), and Figure 8 on the next page contains the generated flute part.

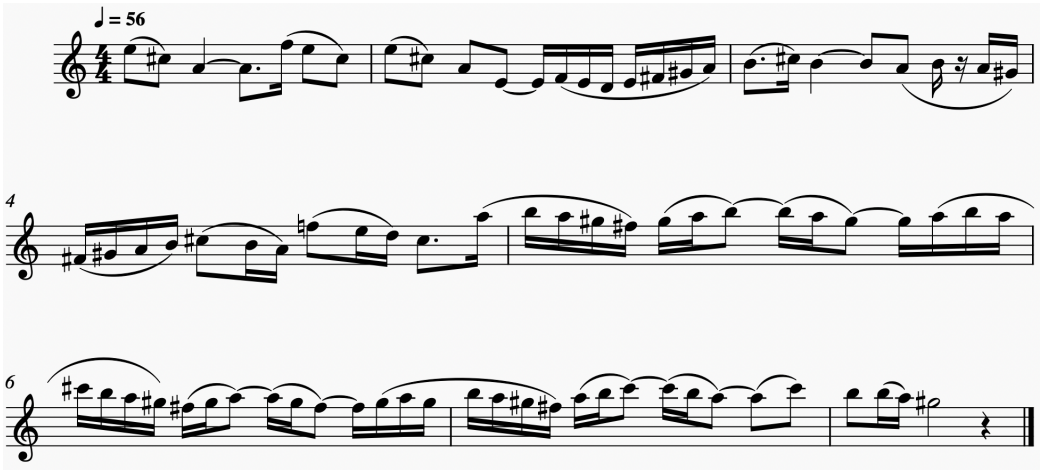

Figure 7: Original flute part [excerpt from *Cantabile* by Ilana Shapiro]

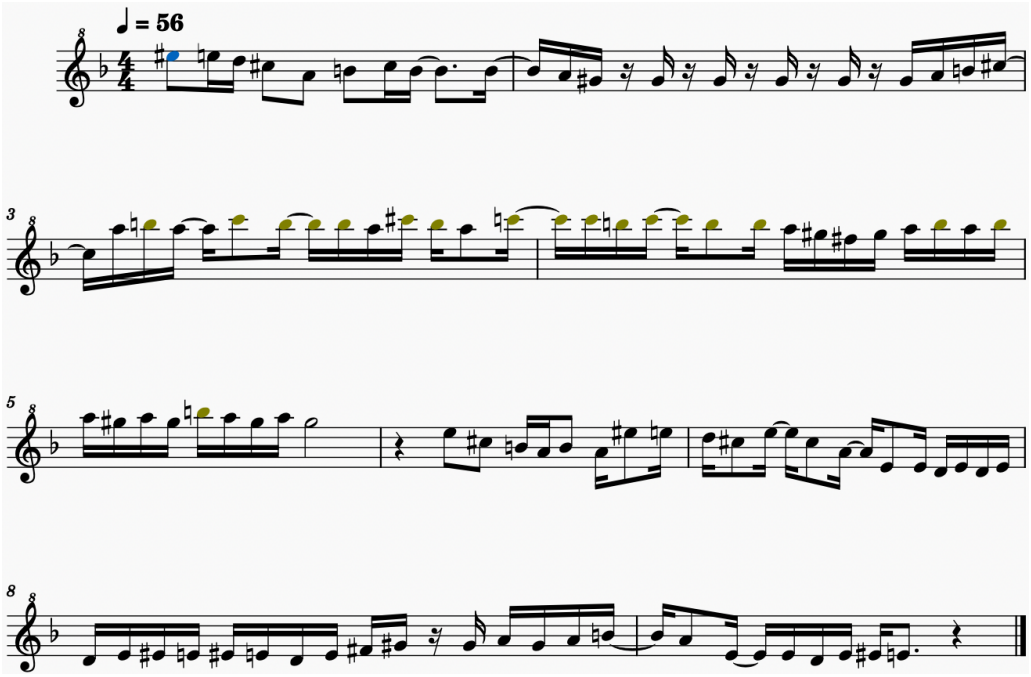

Figure 8: Generated flute part

Notice that the generated flute part does not have the same number of measures as the original flute part. When running the program, the user must specify how many measures the generated part will be. This number does not have to match that of the training data, since the training data is only used to create the Markov chain. Once this is complete, arbitrarily long pieces can be generated from the chain.

The generated flute part in Figure 8 contains marked similarities to the original. The rhythm in measure 1 of the generated part is quite similar to the rhythm in measure 4 of the original, and the harmony throughout centers around the key of A major, just like the original, even though the key signature indicates D minor. Additionally, notice the behavior of the C in measures 3-4: it is sometimes flat and sometimes sharp, a behavior picked up from measures 6-7 of the original score. Some notes, such as the first two eighth notes of measure 6 in the generated score, are a direct quote (in this case, of measure 1) from the original. The last beat of measure 2 in the generated score also appears to be an incomplete F# minor scale inspired by measure 4 of the original.

Figure 9 below contains the original piano part (i.e., the training data for piano) and Figure 10 on the next page contains the generated piano part. They also do not have the same number of measures, as was specified by the user before running the program.

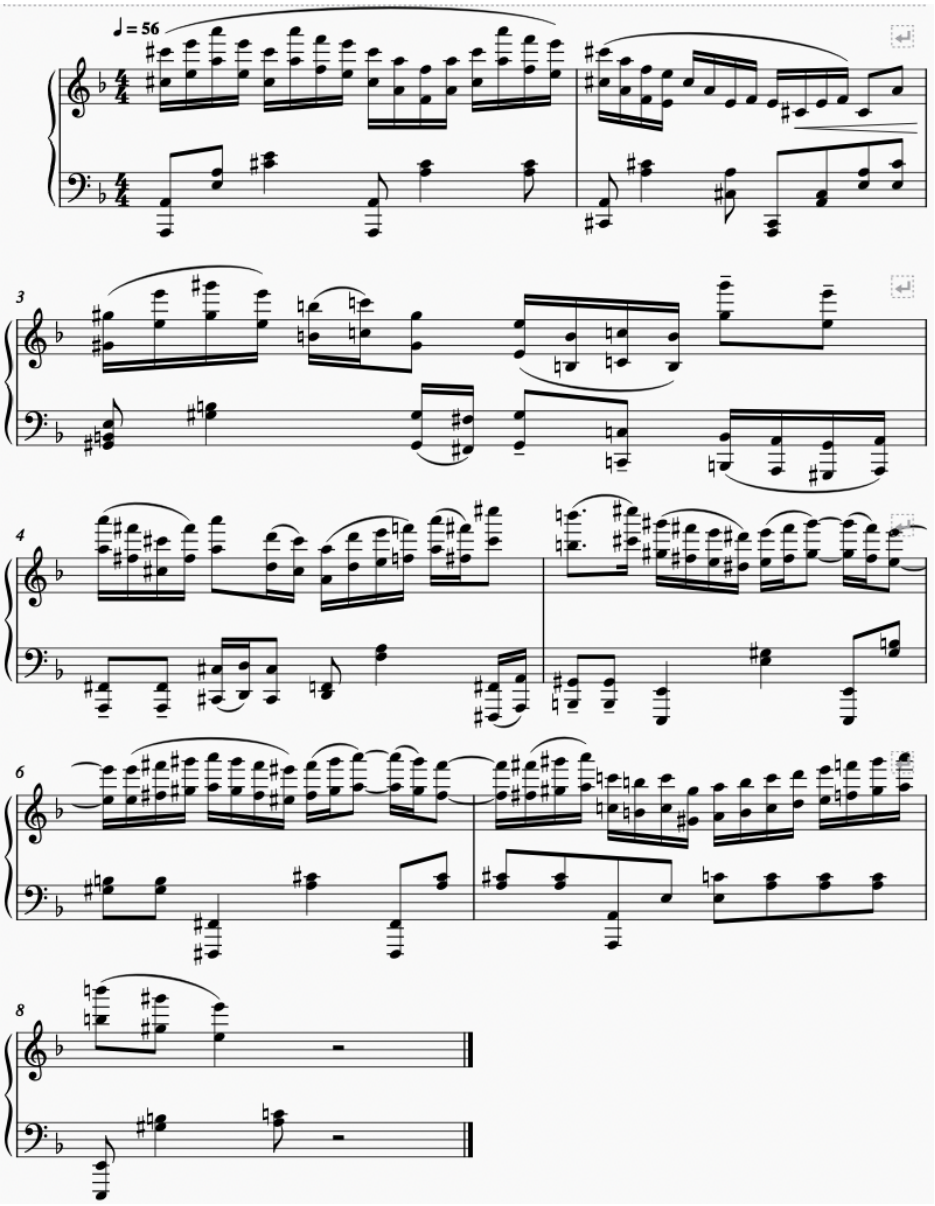

Figure 9: Original Piano Part [excerpt from *Cantabile* by Ilana Shapiro]

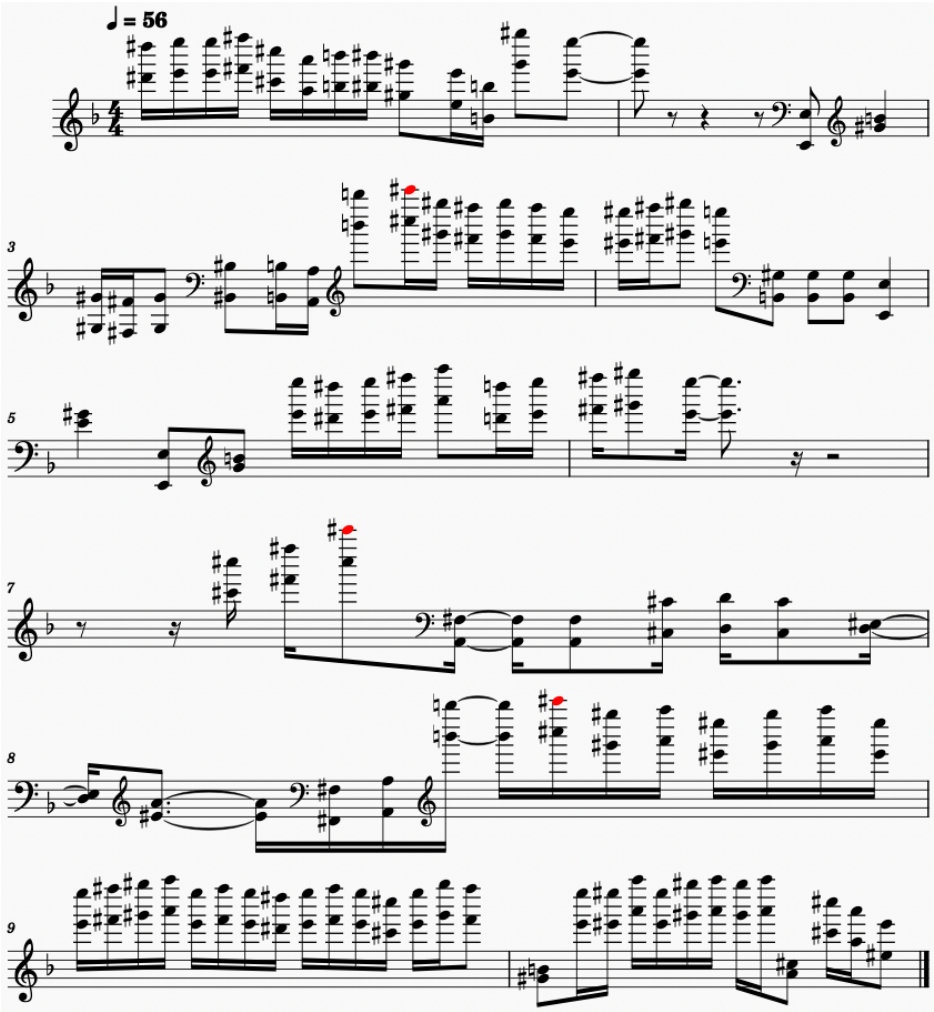

Figure 10: Generated Piano Part

(Note that because of the way the MIDI file was generated, the generated piano part gets compressed into a single staff. This is not a result of the Markov chain, it is simply due to the MIDI formatting).

The generated piano part in Figure 10 demonstrates melodic, harmonic, and rhythmic qualities from the training piece. The parallel octaves from the original score are frequent throughout the generated part, and harmonic structures (like the augmented C chord (C-E-G#) in the final measure and the A major scale in measure 9) have made their way through as well.

In addition, notice the rhythmic similarity of the scores, particularly the common patterns of sixteenth notes tied over into the next beat and the pattern of four sixteenths, one eighth and two sixteenths, and two eighths that appears in both measure 3 of the original and measure 1 of the generated score.

7. Conclusions

Using a simple Markov chain, music can be successfully generated in the style of the training piece. Rhythm, octave, pitch, and accidentals are accounted for. However, there are limitations to the current setup as well as many other avenues to be explored. Currently, the parser does not handle pieces with multiple voices within a single part, or a piece with multiple instruments considered simultaneously, due to difficulty parsing the data from the current musicXML format. In addition, dynamics are not taken into account. Other statistical models, such as the Hidden Markov Model (HMM) mentioned earlier in the paper, may provide interesting avenues of exploration. Using HMMs and dynamic programming, we could, for instance, generate observable notes/chords, and use dynamic programming to uncover the optimal sequence of rhythms, or perhaps dynamics (whatever we choose to be the hidden states) based on the observables. It may also be an interesting avenue to explore the power of simple as well as hidden Markov models in creating less tonal music, and even jazz. It is evident that statistical modeling opens a multitude of creative avenues for computer music generation.

References

[1] Thayabaran Kathiresan, *Automatic Melody Generation*, PhD thesis, KTH Royal Institute of Technology School of Electrical Engineering, June 2015.

[2] Johann Philipp Kirnberger, *Der allezeit fertige Polonaisen- und Menuettencomponist*, Werner Icking, 1757.

[3] Johann Philipp Kirnberger, *Der allezeit fertige Polonaisen- und Menuettencomponist*, George Ludewig Winter, 1767.

[4] Erlijn J Linskens, *Music Improvisation using Markov Chains*, PhD thesis, Maastricht University, June 2014.

[5] Andrey Andreevich Markov, In Yu. V. Linnik, editor, *Selected Works*, Classics of Science, Academy of Sciences of the USSR, 1951.

[6] Adhika Sigit Ramanto and Nur Ulfa Maulidevi, "Markov chain based procedural music generator with user chosen mood compatibility", *International Journal of Asia Digital Art & Design*, Volume 21 Issue 1 (March 2017), pages 19–24.

[7] Örjan Sandred, Mikael Laurson, and Mika Kuuskankare, "Revisiting the illiac suite—a rule based approach to stochastic processes", *Sonic Ideas/Ideas Sonicas*, Volume **2** (2009), pages 42–46.

[8] Anna K Yanchenko and Sayan Mukherjee, *Classical Music Composition Using State Space Model*, PhD thesis, Duke University, September 2018.

A. Appendix

The following code can also be accessed at
https://github.com/ilanashapiro/Markov-Model-Music-Generation

A.1. `parse_MusicXML.py`

```
import xml.etree.ElementTree as ET
import collections
import numpy as np

class Parser:
  def __init__(self, filename):
    self.filename = filename
    self.root = ET.parse(filename).getroot()

    self.initial_transition_dict = collections.
       OrderedDict()
    self.normalized_initial_probability_vector =
       None

    self.transition_probability_dict = collections.
       OrderedDict()
```

```
14      self.normalized_transition_probability_matrix =
            None
15
16      self.states = []
17
18      self.smallest_note_value = None
19      self.tempo = None
20
21      self.order_of_sharps = ['F', 'C', 'G', 'D', 'A',
            'E', 'B']
22      self.key_sig_dict = {'C':'', 'D':'', 'E':'', 'F'
            :'', 'G':'', 'A':'', 'B':''}
23      self.parse()
24
25    def parse(self):
26      prev_note = None # the prev note (it may be part
            of a chord). but this variable itself NEVER
            stores a chord
27      sound_object_to_insert = None # either note or
            chord
28      prev_sound_object = None # either note or chord
29      in_chord = False
30      note = None
31      chord = None
32      prev_duration = None
33      first_sound_object = None
34
35      direction_blocks = self.root.find('part').find('
            measure').findall('direction')
36      for direction_block in direction_blocks:
37        if self.tempo is None and direction_block.find
              ('sound') is not None and 'tempo' in
              direction_block.find('sound').attrib:
38          self.tempo = int(direction_block.find('sound
                ').attrib['tempo'])
39      self.instrument = self.root.find('part-list').
            find('score-part').find('part-name').text
40      if self.instrument == 'Piano':
```

```
41          self.instrument = 'Acoustic Grand Piano'
42        self.name = self.root.find('credit').find('
              credit-words').text
43
44        for i, part in enumerate(self.root.findall('part
              ')):
45          for j, measure in enumerate(part.findall('
                measure')):
46            measure_accidentals = {}
47            self.set_key_sig_from_measure(measure)
48
49            for k, note_info in enumerate(measure.
                  findall('note')):
50              duration = note_info.find('type').text
51              note = None
52
53              if note_info.find('pitch') is not None:
54                value = note_info.find('pitch').find('
                      step').text if note_info.find('pitch'
                      ).find('step') is not None else ''
55                octave = note_info.find('pitch').find('
                      octave').text if note_info.find('
                      pitch').find('octave') is not None
                      else ''
56
57                accidental_info = note_info.find('
                      accidental').text if note_info.find('
                      accidental') is not None else None
58                accidental = '' if accidental_info is
                      None else ('#' if accidental_info ==
                      'sharp' else 'b' )
59                note_for_accidental = value+octave
60                if accidental_info == 'sharp':
61                  accidental = '#'
62                  measure_accidentals[
                        note_for_accidental] = '#'
63                elif accidental_info == 'flat':
64                  accidental = 'b'
```

```
65              measure_accidentals[
                    note_for_accidental] = 'b'
66            elif accidental_info == 'natural':
67              accidental = ''
68              measure_accidentals[
                    note_for_accidental] = 'n'
69            elif accidental_info is None and
                   note_for_accidental in
                   measure_accidentals:
70              accidental = '' if measure_accidentals
                    [note_for_accidental] == 'n' else
                    measure_accidentals[
                    note_for_accidental]
71            else:
72              accidental = self.key_sig_dict[value]
73
74            note = value + accidental + octave
75          elif note_info.find('chord') is None:
76            # means that note_info.find('rest') is
                 not None ----> so we are in a rest
77            note = 'R'
78
79          if note is not None:
80            is_last_iteration = i == len(self.root.
                 findall('part')) - 1 and j == len(
                 part.findall('measure')) - 1 and k ==
                  len(measure.findall('note')) - 1
81
82            if note_info.find('chord') is not None:
83              # currently, the duration is just
                   going  to be that of the last note
                   in the chord (can go back to change
                    this later...)
84              if in_chord:
85                chord.append(note)
86              else:
87                chord = [prev_note, note]
88                in_chord = True
```

```
89              else:
90                prev_sound_object =
                     sound_object_to_insert
91                if in_chord and note_info.find('chord'
                     ) is None:
92                  in_chord = False
93                  sound_object_to_insert = (tuple(
                       sorted(chord)), prev_duration)
94                else:
95                  sound_object_to_insert = (prev_note,
                       prev_duration)
96
97                self.handle_insertion(
                     prev_sound_object,
                     sound_object_to_insert)
98                if first_sound_object is None and
                     prev_sound_object is not None and
                     prev_sound_object[0] is not None:
99                  first_sound_object =
                       prev_sound_object
100               if is_last_iteration:
101                 # note that we're NOT in a chord (i.
                       e. last sound object is NOT a
                       chord)
102                 self.handle_insertion(
                       sound_object_to_insert, (note,
                       duration))
103
104           prev_note = note
105           prev_duration = duration
106
107     if in_chord:
108       # handle the case where the last sound object
             was a chord
109       final_chord = (tuple(sorted(chord)),
             prev_duration)
110       self.handle_insertion(sound_object_to_insert,
             (final_chord, prev_duration))
```

```
111     else:
112       # final sound object was NOT a chord, it was a
              note
113       # set the last note/chord to transition to a
             quarter rest, rather than nothing
114       # then add a transition from the rest back to
             the first sound object
115       # this ensure that everything has a transition
              defined for it
116       self.handle_insertion((note, duration), ('R',
             "quarter"))
117       self.handle_insertion(('R', "quarter"),
             first_sound_object)
118
119     self.build_matrices()
120
121   def set_key_sig_from_measure(self, measure_object)
        :
122     key_sig_value = measure_object.find('attributes'
        )
123     if key_sig_value is not None:
124       key_sig_value = key_sig_value.find('key')
125       if key_sig_value is not None:
126         key_sig_value = int(key_sig_value.find('
              fifths').text)
127
128         for note in self.key_sig_dict:
129           self.key_sig_dict[note] = ''
130         if key_sig_value == 0:
131           return
132         elif key_sig_value < 0:
133           for i in range(len(self.order_of_sharps) -
                 1, len(self.order_of_sharps)-
                key_sig_value, -1):
134             self.key_sig_dict[self.order_of_sharps[i
                  ]] = 'b'
135         else:
```

```
136             for i in range(key_sig_value, len(self.
                    order_of_sharps)):
137               self.key_sig_dict[self.order_of_sharps[i
                    ]] = '#'
138
139     def build_matrices(self):
140       self.
            build_normalized_transition_probability_matrix
            ()
141       self.build_normalized_initial_probability_vector
            ()
142
143     def build_normalized_initial_probability_vector(
          self):
144       self.normalized_initial_probability_vector = np.
            array(list(init_prob for init_prob in self.
            initial_transition_dict.values()))
145       # convert to probabilities
146       self.normalized_initial_probability_vector =
            self.normalized_initial_probability_vector/
            self.normalized_initial_probability_vector.
            sum(keepdims=True)
147       # multinomial dist
148       self.normalized_initial_probability_vector = np.
            cumsum(self.
            normalized_initial_probability_vector)
149
150     def build_normalized_transition_probability_matrix
          (self):
151       # initialize matrix to known size
152       list_dimension = len(self.states)
153       self.normalized_transition_probability_matrix =
            np.zeros((list_dimension,list_dimension),
            dtype=float)
154
155       for i, sound_object in enumerate(self.states):
156         for j, transition_sound_object in enumerate(
              self.states):
```

```
157         if transition_sound_object in self.
               transition_probability_dict[sound_object
               ]:
158           self.
                 normalized_transition_probability_matrix
                 [i][j] = self.
                 transition_probability_dict[
                 sound_object][transition_sound_object]
159     self.normalized_transition_probability_matrix =
          self.normalized_transition_probability_matrix
          /self.
          normalized_transition_probability_matrix.sum(
          axis=1,keepdims=True)
160     self.normalized_transition_probability_matrix =
          np.cumsum(self.
          normalized_transition_probability_matrix,axis
          =1)
161
162   def handle_insertion(self, prev_sound_object,
        sound_object_to_insert):
163     if sound_object_to_insert is not None and
          sound_object_to_insert[0] is not None:
164       if prev_sound_object is not None and
            prev_sound_object[0] is not None:
165         self.insert(self.transition_probability_dict
              , prev_sound_object,
              sound_object_to_insert)
166       if sound_object_to_insert not in self.states:
167         self.states.append(sound_object_to_insert)
168
169       if sound_object_to_insert in self.
            initial_transition_dict:
170         self.initial_transition_dict[
              sound_object_to_insert] = self.
              initial_transition_dict[
              sound_object_to_insert] + 1
171       else:
```

```
172         self.initial_transition_dict[
               sound_object_to_insert] = 1
173
174   def insert(self, dict, value1, value2):
175     if value1 in dict:
176       if value2 in dict[value1]:
177         dict[value1][value2] = dict[value1][value2]
               + 1
178       else:
179         dict[value1][value2] = 1
180     else:
181       dict[value1] = {}
182       dict[value1][value2] = 1
183
184   def print_dict(self, dict):
185     for key in dict:
186       print(key, ":", dict[key])
187
188   def rhythm_to_float(self, duration):
189     switcher = {
190       "whole": 4,
191       "half": 2,
192       "quarter": 1,
193       "eighth": 1/2,
194       "16th": 1/4,
195       "32nd": 1/8,
196       "64th": 1/16,
197       "128th": 1/32
198     }
199     return switcher.get(duration, None)
```

A.2. generate.py

```
1 import parse_MusicXML
2 import random
3 import numpy as np
4 import midi_numbers
```

```
5 from midiutil import MIDIFile
6 import sys
7 import re
8
9 # I did not write this function. Credit: Akavall on
     StackOverflow
10 # https://stackoverflow.com/questions/17118350/how-
     to-find-nearest-value-that-is-greater-in-numpy-
     array
11 def find_nearest_above(my_array, target):
12   diff = my_array - target
13   mask = np.ma.less(diff, 0)
14   # We need to mask the negative differences and
       zero
15   # since we are looking for values above
16   if np.all(mask):
17     return None # returns None if target is greater
         than any value
18   masked_diff = np.ma.masked_array(diff, mask)
19   return masked_diff.argmin()
20
21 def generate(seq_len, parser):
22   sequence = [None] * seq_len
23
24   # comment in for same start note as training data
25   note_prob = random.uniform(0, 1)
26   rhythm_prob = random.uniform(0, 1)
27   note_index = find_nearest_above(parser.
       normalized_initial_probability_vector,
       note_prob)
28   check_null_index(note_index, "ERROR getting note
       index in initial probability vector")
29   curr_index = 0
30
31   # comment in for seed
32   # sequence[0] = parser.states[0]
33   # note_index = 0
34   # curr_index = 1
```

```
35
36    while (curr_index < seq_len):
37      note_prob = random.uniform(0, 1)
38      rhythm_prob = random.uniform(0, 1)
39
40      note_index = find_nearest_above(parser.
          normalized_transition_probability_matrix[
          note_index], note_prob)
41      check_null_index(note_index, "ERROR getting note
            index in probability transition matrix")
42
43      sequence[curr_index] = parser.states[note_index]
44      curr_index += 1
45
46    return sequence
47
48  def check_null_index(index, error_message):
49    if(index == None):
50      print(error_message)
51      sys.exit(1)
52
53  def get_note_offset_midi_val(note):
54    switcher = {
55      "C": 0,
56      "C#": 1,
57      "Db": 1,
58      "D": 2,
59      "D#": 3,
60      "Eb": 3,
61      "E": 4,
62      "Fb": 4,
63      "E#": 5,
64      "F": 5,
65      "F#": 6,
66      "Gb": 6,
67      "G": 7,
68      "G#": 8,
69      "Ab": 8,
```

```
70      "A": 9,
71      "A#": 10,
72      "Bb": 10,
73      "B": 11,
74      "Cb": 11
75    }
76    return switcher.get(note, 0)
77
78  def get_pitch(note):
79    octave_info = re.findall('\d+', note)
80    if len(octave_info) > 0:
81      octave = int(octave_info[0])
82      note = ''.join([i for i in note if not i.isdigit
          ()])
83      base_octave_val = 12*octave + 24
84      note_val = base_octave_val +
          get_note_offset_midi_val(note)
85      return note_val
86    return None # this is a rest
87
88  if __name__ == "__main__":
89    parsers = [parse_MusicXML.Parser('
        Cantabile_flute_excerpt.musicxml'),
        parse_musicxml.Parser('Cantabile_piano_excerpt.
        musicxml')]
90
91    for parser in parsers:
92      sequence = generate(100, parser)
93      track  = 0
94      channel  = 0
95      time   = 0.0  # In beats
96      duration = 1.0   # In beats
97      tempo  = parser.tempo if parser.tempo is not
          None else 80  # In BPM
98      volume   = 100  # 0-127, as per the MIDI
          standard
99
```

```
100     output_midi = MIDIFile(1)  # One track, defaults
            to format 1 (tempo track is created
           automatically)
101     output_midi.addTempo(track, time, tempo)
102     output_midi.addProgramChange(track, channel,
           time, midi_numbers.instrument_to_program(
           parser.instrument))
103
104     time = 0.0
105     for sound_obj in sequence:
106       duration = float(parser.rhythm_to_float(
             sound_obj[1]))
107       sound_info = sound_obj[0]
108       if type(sound_info) is str:
109         pitch = get_pitch(sound_info)
110         if pitch is not None: # i.e. if this is not
               a rest
111           output_midi.addNote(track, channel, pitch,
                  time, duration, volume)
112       else: #  type(sound_info) is tuple
113         for note in sound_info:
114           pitch = get_pitch(note)
115           output_midi.addNote(track, channel, pitch,
                  time, duration, volume)
116       time += duration
117     with open(parser.filename + ".mid", "wb") as
           output_file:
118       output_midi.writeFile(output_file)
```