Composerator

CS51 Final Project Specification

Garrett Parrish, Cameron Clarke, Billy Cox

Project Goals:

Composerator is meant to be an algorithmic music composition application. It will take, as input, a string of notes ~~(either manually entered, deduced from an audio file, or~~ read from a MIDI file and translate it into an output melody that is similar to the input. One could input any number of data streams and have them converted to a single output: for example you could input a Bach sonata melody and a Coltrane solo and see what the combination sounds like. The main algorithm that will facilitate this is the use of Markov Chains to probabilistically determine the output based on the pattern of input notes. Composerator will consist of three main parts: input encoding and preparing for analysis, the actual Markov Chain analysis that will analyze the song, and the decoding of the previous part into a single data stream to be played.

Language:

We will implement this project in Java and will use the Eclipse programming environment. We have already installed the git extension for Eclipse and have begun development.

Version Control

We have set up a git repo using one of our team members' github accounts and have tested committing, pulling, and pushing to and from repository from all of our team members' computers.

Part 1: Input

Before discussing how input will be processed, it's crucial to understand the main classes of this application.

Note: Notes are what make up the majority of the input and output music. Notes consist of three instance variables: pitch object, a duration, and a volume.

Pitch: Pitches contain a ~~frequency~~, octave, class ("C", "A"), and two exclusive booleans (flat and sharp).

We chose to include a MIDI ID instead of a frequency. A MIDI ID is an integer in the set {0, 1, … ,127} that represents one of the notes C0 through G10, without enharmonic repetitions. A full list of the MIDI ID numbers can be found both in the comments of Pitch.java and in the midi documentation at http://www.midimountain.com/midi/midi_note_numbers.html

Volume: A volume consists of a double representing the volume.

Duration: Duration consists of a double representing the time.

Rest: A rest is a special subclass of note with a default value of pitch, zero volume, and a indeterminate duration, and a MIDI ID of 128 (the default representation in MIDI).

Function 1: Convert an audio file to three *chainable* objects, one of type Note, one of type Duration, and one of type Volume. To do this, we will iterate over the song at a fixed interval, and calculate the FFT (Fast-Fourier Transform) of that interface. From there, will will scan the FFT output for the highest-powered frequency, by look at which frequency has the highest amplitude. From there, the pitch class and octave can be deduced by choosing the note class and octave closest to the previously determined highest-power frequency from the FFT output. Then, we can produce a note object with the pitch, duration, and volume as generated by this algorithm.

Function 1 Final Version

We ended up using Musical Instrument Digital Interface (MIDI) files as our input instead of raw audio files. For the purposes of our project, MIDI files have a few key advantages over raw audio. Firstly, much of the information we wished to deduce from audio files (pitch, volume, duration, etc) can be more readily deduced from the data contained within a MIDI file than a pure audio file. Additionally, many standard music notation applications allow MIDI files to be imported and viewed as notated sheet music. Thus we can view both input and output files in sheet music form. Finally, our application is designed to work best with monophonic melodies; these are easy to both extract and display in MIDI files.

Function 2: Effectively, at this point our input will look something like this (a list of note objects):
[(A, 3, 0.3), (Bb, 2.3, 0.4), … (G#, .4, 1.0)], which represents an A being played for 3 seconds at 0.3 volume followed by a Bb for 2.3 at 0.4 volume etc.

This implementation remained relatively unchanged, except for the fact that the first field in each entry (A, Bb, G#) contains not a pitch name but a Note object.

Chainable Interface:
Then, from this we will form three objects of the *chainable* interface corresponding to the 1st, 2nd, and 3rd entries of each 'note'. The Chainable abstract class is for Pitch, Volume, and Duration values to live each within one object of type Chainable.

Instead of implementing Chainable as an interface, we chose to implement it as an abstract class which Pitch, Volume, Duration, and Note extend. These first three types make up a Note object, which are themselves also of type Chainable.

From there, we can proceed to part 2 to actual probabilistically determine the output 'song'.

Part 2: Markov Chains

Song: A song consists of three chainable objects: a Note, Duration, and Volume Chain (as created by part 1 of implementation).

Each note, duration, and volume chain implements the chainable interface which has methods and instance variables to apply broadly to any type of musical field that would pertain to note (pitch, duration, volume, and any other that may be added later). ~~Before each chain is processed into a Markov Chain and corresponding matrix, it executes the 'minimize' method to intelligently determine the range, increment, and order of the Markov matrix based on the input. For example, if the input only used notes from A3-G5, then it would only create those rows/columns for the matrix as opposed to generating a lot of rows that would otherwise be empty and make computation much slower.~~

The algorithm pertaining to the markov chain works as follows. Consider the following input note chain: [A, B, C, D, B, A, C, F, G, D, C, A]

The following matrix would be generate the following counts (left) for each current/follow pair and therefor the following predictive probabilities (right)

Following

Current

	A	B	C	D	E	F	G
A		1	1				
B	1			1			
C	1	1		1			
D			1				1
E							
F			1				
G						1	

Following

Current

	A	B	C	D	E	F	G
A		0.5	0.5				
B	0.5			0.5			
C	0.33	0.33		0.33			
D			0.5				0.5
E							
F			1				
G						1	

So, when going back to generate the new melody, when we have a C, there is a 1/3 chance it will follow an A,B, or D, and will generate that note according to those probabilities (part 3). Right now, these examples only represent a fraction of the notes available, but you can imagine how it would scale with the size of the matrix. Additionally, this is only a Markov Matrix up to 1st order (meaning that we only look at one element previously. But, if we look at say the previous three elements, the entries in each of the rows/columns would be of the form: AAA, AAB, AAC and the matrix would be a power of 3 larger. This is a trade off between quality/creativity in the music and size/time of computation.

So, in order to implement these computations, we are creating several classes and interfaces to represent the rows, columns, matrices, operations, and methods of traversing the matrices.

Chain Interface: The chain interface is different from the Chainable interface in that it actually corresponds to the matrices that perform the Markov operations. See source code for implementation.

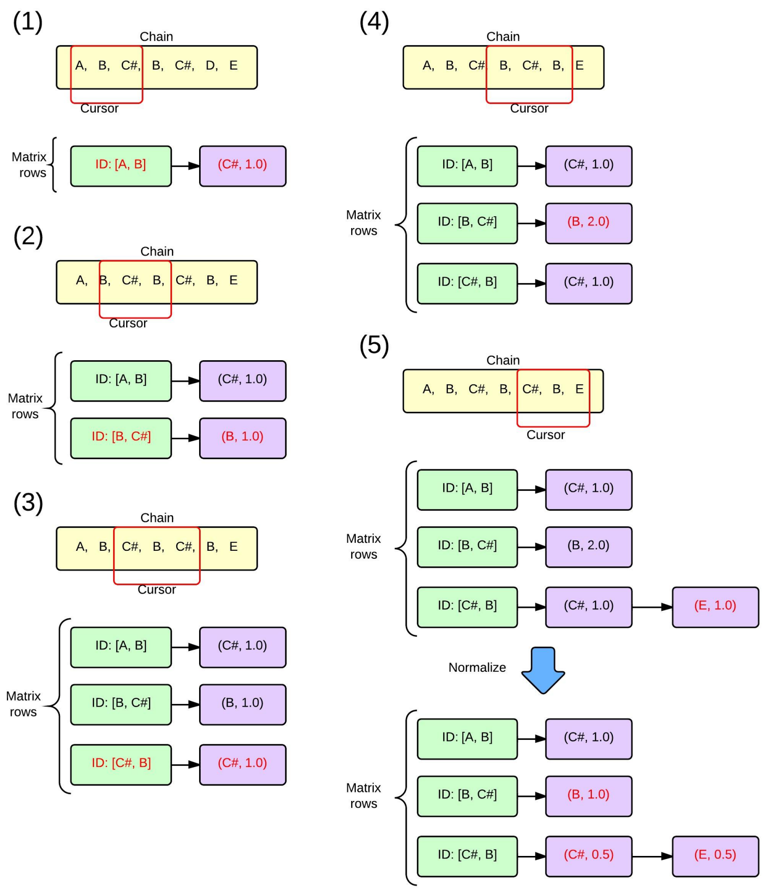
MarkovRow: Models a row in the matrix. Contains an array of objects of type MarkovEntry, an id of type

Cursor, and a length. It has methods to update a row, to normalize the row (divide each entry by the sum of its entries), a boolean to tell whether or not the row has been normalized, and a method to compare rows to see if they are the same. Also, it has attributes and methods to prepare a row for composition and to select a random note with the same id as the row given the probabilities specified in the row.

MarkovMatrix: Markov matrix represents the Markov Chain matrix for one of the Chainable objects. It consists of a list of objects of type MarkovRow, an order, and methods to aid in composition.

Cursor: The cursor is the object used to actually traverse the matrix and iterate through the Markov Rows and Columns. This object will store an array of (order + 1) elements of a given chainable list by traversing this list in order, looking at (order + 1) elements at a time. The first (order) elements of the cursor determine the row in our MarkovMatrix that we want to update. The (order + 1)st element determines the column in that row that we want to update. Given these coordinates, we increment the entry of the MarkovMatrix by 1.

The following graphic gives an overview of what a MarkovMatrix is and how it is constructed from the other Markov classes and Cursor. This example shown deals specifically with Pitch objects but can be generalized to duration and Volume objects as well:



We start in step (1) with a chain of Pitches, represented by the yellow Chain box. We have a Cursor object, represented by a red box, surrounding the first three notes in the chain (A, B, C#). Tis cursor has length 3, so we know we are dealing with a second order Markov chain. In each case, the first two elements in the Cursor determine the ID of an entry into the matrix and the last element determines how to handle making or updating an entry into the matrix. Initially our MarkovMatrix contains no MarkovRow objects, and is thus empty. Since this is the case, we create a new MarkovRow with ID = [A, B] and the entry [C#, 1]. This entry represents that there has been one occurrence of the Pitch C# following the sequence [A, B] of Pitch objects.

The cursor continues to advance through the Chain in step (2). We see that the sequence of Note objects [B, C#] has not appeared yet, so we add a row to the matrix with ID = [B, C#] and whose only entry so far is [B, 1.0]. This process continues analogously in step (3).

In step (4) we see that the Cursor comes upon an ID sequence (B, C#) that has already been encountered. Thus we navigate to row [B, C] in our matrix. Also, we see that the Note B has already appeared following this ID sequence previously in the song, so instead of creating a new entry in row [B, C] we increment the existing entry B to (B, 2.0). In step 5 we encounter a similar situation where we see an ID that has already been encountered. However, the note following this sequence is new, so we create a new entry [E, 1.0].

After the matrix has been created, we must normalize each row to determine the probabilities of each note following a given ID. To do this, we divide each entry in the row by the sum of the entries in the row. This process results in the matrix that appears below the blue arrow in the diagram, which contains all of the IDs encountered in the song and the probability of a given note following a given ID. It should be noted that each row does not necessarily include every possible note, nor do we have rows for every possible sequence of Notes as an ID. Were we to have this, we would most likely end up with a very large sparce matrix (one where most entries in the matrix are 0). In our implementation, the exclusion of a row or an entry in a row signifies that a given ID or note following an ID never occurs in the song. This choice cuts down drastically on the space complexity of our matrix and the time complexity of entry lookup.

Part 3: Output

This part of the program is where we translate the Markov matrices into the final output chains and finally an output MIDI or audio track. We will be using Java's standard audio library to actually output the audio. So, now that we have the Markov Matrices already generated, we will then, using the cursor, traverse back through the matrix in reverse or to probabilistically generate the new melody. For example, if we start with the same note as the input, then we would find that row's pitch in the matrix and probabilistically determine the following pitch and would repeat the same process for duration and volume, gradually building up the entire melody as we go through. The ouput, again would be in the form of a 'song' which is an amalgamation of three chainable objects. Then, we will call the 'play' method of the song to actually output it.

Code:

To check out our source code so far (is well documented and may make all of this clearer), please pull the public repo from github: git@github.com:fawrkes/Composerator.git

Demo Video:

Check out our demo video: https://vimeo.com/93806950