**Programming Four-Part Harmony Composition**

**CSC 492 (Kurfess - F ‘19)**

**Final Report**

*Keenan Icarangal*

*kicarang@calpoly.edu*

**Background & Overview**

The topics tackled in this project include music composition (specifically choral music) and music generation with code (outputting a MIDI file that may be used in composition software). With modern composition software, the user must manually input notes one by one, constantly accounting for voicing and composition rules, in order to compose a full arrangement. This project aims to streamline the process of writing choral music. My goal is to be able to write full choral pieces with four-part harmony (soprano, alto, tenor, bass) -- taking in the melody line (soprano part) and lead sheet (chord progression) -- having the program dictate the notes for the three other parts, accounting for the aforementioned composition rules. This program will be helpful to those who want to compose a choral piece within a small timeframe and composers who want a basis to work from and modify to their liking.

Composers usually use composition software such as MuseScore and Sibelius to bring their musical concepts to life. Current music programs such as Magenta rely on machine learning to help users gain perspective on the creative process and help develop their projects further. Studies have been done by three individuals from Ionian University in Greece, called Automatic Melodic Harmonization, where they examine different machine learning approaches and algorithms to help generate harmonization. There is also software called Harmony Builder which helps users arrange and compose music easier with the aid of artificial intelligence. These are two similar projects undertaken by other college students in the past:

* Auto-Harmonization plugin for Finale 2010 <https://www.youtube.com/watch?v=mGWMCdmO9aM&app=desktop>
* Harmonization that takes in WAV file melody and outputs harmony

<https://www.youtube.com/watch?v=2MbfRtTNZlQ&app=desktop>

This program does not use machine learning and instead will have composition rules to work off of.

My system will help those who want to arrange choral music by cutting down the time it takes to compose a full piece and simplifying the composition process. The program -- written in Java -- will be able to take in a melody line and chord progression and systematically compose a complete choral piece with four-part harmony. The user will input a MIDI file containing the melody (this can be created with music composition software) and the chord progression associated with the piece (the user must know this before using the program). The program will then output a MIDI file of the complete piece.

The goal of this quarter was to take the simple harmonizer prototype developed during Spring ‘19 (which could take in a melody and chord progression of a song and generate a four-part harmony following certain guidelines) and add features to enhance the user’s experience; essentially version 2.0. The new features included a better command-line interface, input of the arrangement’s melody line using a MIDI file, input of the arrangement’s chord progression using a text file, and the output of the final arrangement generated by the harmonizer with a MIDI file. In comparison, the simple harmonizer only took in command-line input for all necessary user input. This document goes over the complete harmonizer prototype developed over the Fall ‘19 quarter in detail.

**Program**

**I/O**

There are 2 Java classes responsible for taking in user input and outputting the newly generated MIDI file. Both utilize the JFugue library to aide both the parsing and generation of MIDI files.

The Driver class is responsible for taking in user input and parsing it in order to pass the necessary information to the Harmonizer class. Driver takes in the following inputs: arrangement’s name, melody file name, chord progression file name, and key signature. The arrangement’s name is used as the output MIDI file’s name (ex. ‘Twinkle Twinkle Little Star’ to ‘TwinkleTwinkleLittleStar.mid’). The melody file name is used to open the melody MIDI file. The melody file is then passed through certain JFugue parse classes in order to obtain the melody line and time signature for the arrangement. The chord progression file name is used to open the chord progression text file. The chord progression file is then parsed by the Driver class in order to create Chord objects. The key signature is passed to the Harmonizer to generate correct chords and notes for the final arrangement. The reason behind having the user manually input the key signature and not the time signature stems from caveats found when using the JFugue library for MIDI parsing. When parsing the melody file, getting the arrangement’s time signature is reliable. However, getting the arrangement’s key signature is not (ex. a melody file with the key of G was the key of F according to the JFugue parser). Other caveats include the octave of notes being one octave too high (ex. a C4 note being parsed as C5), note values being floats (ex. an eighth note represented as ‘0.11822916666666666A80D0’), and rests being in between adjacent notes (ex. two adjacent C4 dotted quarter notes being represented as C5/0.35572916666666665A80D0 R/0.019270833333333348 C5/0.35572916666666665A80D0).

The Generator class is responsible for taking the generated four-part harmony from the Harmonizer class and generating/outputting the final MIDI file for the user. This class utilizes the JFugue Pattern class to simplify the process of taking in each part and “stitching” them together, passing the created Pattern object to the MIDI file writer.

**Harmonizer Classes**

The harmonizer utilizes 5 Java classes that all work in tandem to generate the four-part harmony.

The ChoirNote and Note classes store the data regarding different types of musical notes. Note is the superclass of ChoirNote and contains the musical note’s name as a String. ChoirNote in addition to the note’s name contains the musical note’s octave and duration. Notes are used in the Chord class to indicate the chord’s different notes (root, third, fifth, bass note) while ChoirNotes are used in the Harmonizer class to represent the different parts (soprano, alto, tenor, bass) as ArrayLists.

The Chord class stores data regarding each chord in the song’s chord progression. A Chord contains the chord’s name, type, notes, duration, inversion, and key (used for generating the correct notes). Every time a new Chord object is created, its constructor generates its type and all the individual notes that comprise it based on its name (ie. Am, C, D#dim/C#). In the Harmonizer class, Chord objects are used in an ArrayList to represent the song’s chord progression.

The Harmonizer uses all the previously mentioned classes to generate all four parts. It contains all the song’s necessary information for the generating the harmonies including its key, chord progression (as an ArrayList of Chord objects), and melody (ArrayList of ChoirNote objects). The four parts are then generated using backend functions (described in the **Harmonizer Functions** section) and stored as four different ArrayLists (of ChoirNote objects) with each representing a certain vocal part (soprano, alto, tenor, bass).

**Harmonizer Functions**

The Harmonizer does the main share of work in this program’s routine due to it housing the backend functions that generate the four parts. These functions include the following: genSop, genBassHomophonic, genMiddleVoicesHomophonic, genHomophonic, and enforceRulesHomophonic.

The private function genSop() simply sets the soprano ArrayList of ChoirNote objects equal to the inputted melody that was passed into the Harmonizer’s constructor.

The private function genBassHomophonic() indexes through the chord progression ArrayList finding each chord’s bass note and adding those notes to the bass ArrayList, taking into account the bass vocal range.

The private function genMiddleVoicesHomophonic() indexes through the chord progression ArrayList and, for each chord, finds the missing chord notes (using the function getMissingNotes) not already sung by the bass or soprano part. If the chord has a certain inversion, it doubles the correct note (ex. root position triads double the root of the chord while first inversion triads double the soprano note). It then sets the alto and tenor parts’ notes accordingly, ensuring that every note in the chord is sung.

The private function enforceRulesHomophonic() calls two other functions: enforceOctaveRuleHomophonic() and enforceIntervalRuleHomophonic(). The function enforceOctaveRuleHomophonic() indexes through all four generated parts and ensures that the alto part is within one octave of the soprano and the tenor part is within one octave of the alto. If one part is not, the function either raises or lowers the note accordingly. The function enforeIntervalRuleHomophonic() indexes through all the four generated parts and checks if any parallel fifths or parallel octaves occur in the arrangement. If either is found, the program indicates to the user where it occurs and between which parts it occurs.

**Conclusion**

**Next Steps**

In terms of the quality of music outputted, rules such as vocal part ranges, spacing, and intervals are accounted for. Some rules not accounted for include tenor and alto having unreasonable interval jumps, passing tones (the 7th note in a major scale) not resolving into the tonic (the 1st note in a major scale), and parallel octaves and fifths.

I found implementing and enforcing these rules difficult. In particular, finding the ordering of which rule to enforce at any given time. The issue stemmed from one rule being enforced negating the other. Due to this, I decided to not enforce some rules. Instead, I indicated to the user the places where rules were broken (parallel 5ths, parallel octaves) and gave them the liberty to resolve the issue whichever way they wanted.

I want to ultimately continually work on this project and clean up some aspects of it. In the future, I plan to implement a better UI since many of my users will not be familiar with the command line interface. A simple drag and drop UI for files will be something to keep in mind.

**Feedback**

I demonstrated my program and its mechanics to my former choir director/music theory teacher (Keane Ishii) and he loved the premise. He compared the process to cooking in the kitchen where the program acts similar to having someone lay out all the pots, pans, and ingredients for the user to utilize. In terms of the musical rules, he said that most of them are arbitrary in modern music and “perfect” music rarely happens anymore.

I also spoke to Darynne Casim, one of the current choir coordinators of the group Ating Himig (Pilipino Cultural Exchange’s non-audition choir). He was impressed by the musical generation and told me that he would utilize it throughout the rest of the academic year. The idea of cutting down the time it took to arrange was appealing to him since composing with his fellow coordinator usually took around 3 hours to lay down a basis (simple voicing of the three parts with chord progression).

**Sources**

"3A Chord Voicing And Doubling". Legacy.Earlham.Edu, 2019,

http://legacy.earlham.edu/~tobeyfo/musictheory/Book2/FFH2\_CH3/3A\_ChordVoicingDoubling.html.

Koelle, David. “JFugue.” *JFugue*, 2017, http://www.jfugue.org/.

Makris, Dimos & Karydis, Ioannis & Sioutas, Spyros. (2016). Automatic Melodic

Harmonization: An overview, challenges and future directions. https://www.researchgate.net/publication/303818327\_Automatic\_Melodic\_Harmonization\_An\_overview\_challenges\_and\_future\_directions

Musilogic. (2018). HarmonyBuilder - The Intelligent Music Composition Software.

http://www.harmonybuilder.com/index.html#

Williams, Victoria. "The Rules Of Harmony". Mymusictheory.Com, 2019,

https://www.mymusictheory.com/learn-music-theory/for-students/grade-6/online-

ourse/175-the-rules-of-harmony