The Chinese University of Hong Kong
Department of Computer Science and Engineering

Final Year Project Report
2016-2017 Spring - KY1602

Automatic piano reduction (backend)
Chord identification

Students: Yip Wai Man (1155047486)
Wong Cheuk Bun (1155048570)

Supervisors: Professor Yip Yuk Lap, Kevin
Professor Lucas Wong

Declaration

This is the final report of a collaborative project. The following content will be similar to the report content of all contributors in this project and parts of content may reuse materials in the half term report of the same project.

Abstract

This is the final report of an ongoing project on the implementation of piano reduction by machine learning. The report covers mainly on the design and implementation of algorithm in chord identification and the music theory behind. Experiments are conducted in order to test the accuracy of the program.

1. Introduction

1.1. Motivation

There are many music masterpieces all over the world where provide a lot of recreation and spiritual comfort to human beings. However, it is not always that one can find all parts or instruments that the composer used originally, especially when it comes to orchestral arrangement, where dozens of uncommon instruments used in a music piece.

Therefore, reduction on complicated music piece is an important topic. Reduction is a process that transforms complicated score into simpler score. Yet, reduction requires a lot of analysis in the original score and different musicians may have different considerations that leads to different results in reduction. Thus, reduction is usually done by human calculation. However, as machine learning becomes more and more common and apply to all aspects in life, reduction using computation algorithms by learning musicians' reduction pattern becomes realistic, which is less time consuming.

In the project, the main focus would be on piano reduction, where an orchestral score would be reduced to piano score. Such a reduction is a common process in music as complicated pieces can be mimicked by one man, which is very common when it comes to performing concerto, which is a music piece usually composed of one solo instrument and an orchestra (to be mimicked by piano). As a result, piano reduction is a common process to be done.

All in all, should reduction be able to processed by computer algorithm, many human force can be saved and users may have different set of reduction in much shorter time.

1.2. Background

Music reduction is an important yet difficult process. It must be done by musicians or people with full musical knowledge. The reduced works can also be distinct and unique as different musicians or people may have their own styles when reducing music works. As a result, a satisfying music reduction for complex music scores still currently requires knowledgeable musicians to do it.

In this project, we hope to build an computer application to do the complex piano reduction automatically by using the technology of machine learning. The application will build and train neural networks from the piano reduction works that was done by musicians, and aim at generating a satisfying automatic piano reduction for any input music scores.

This project has been launched for a few years. By the passion and hard works of our supervisors, Professor Kevin Yip and Professor Lucas Wong, and some former Computer Science and Engineering students in CUHK, lots of milestones have been done successfully. However, the reduction algorithm is said to be having a large room of improvement by implementing advanced music rules, for instances, chord.

In the future, the reduction algorithm is hoped to be brought to perfection by inserting new reduction rules. Moreover, to bring the computer application to client-side, the project will also develop more on the user interface part.

1.3. Objective

As mentioned in the previous sections, the project attempts to implement new features and provide a graphical user interface to enhance user-friendliness. The project then thus divided into two main part: backend parts for new features and front end part for user interface. This report will focus on the backend parts by analyzing the chords of the piece.

The backend project would look into chord patterns in music and run analysis in order to discover more efficient rules in machine learning of reduction process.

In this semester, the project would focus on designing algorithm to identify chords in a given piece of MusicXML file.

2. Music Theory

In this section, basic music theory that helps reader to understand the logic of the algorithm design would be discussed.

2.1. Basic Music Element

2.1.1. Music Score

Music score is handwritten, printed or digitalized form of music notation that users can indicate pitches, rhythm and other basic elements to allow them to play on instruments.

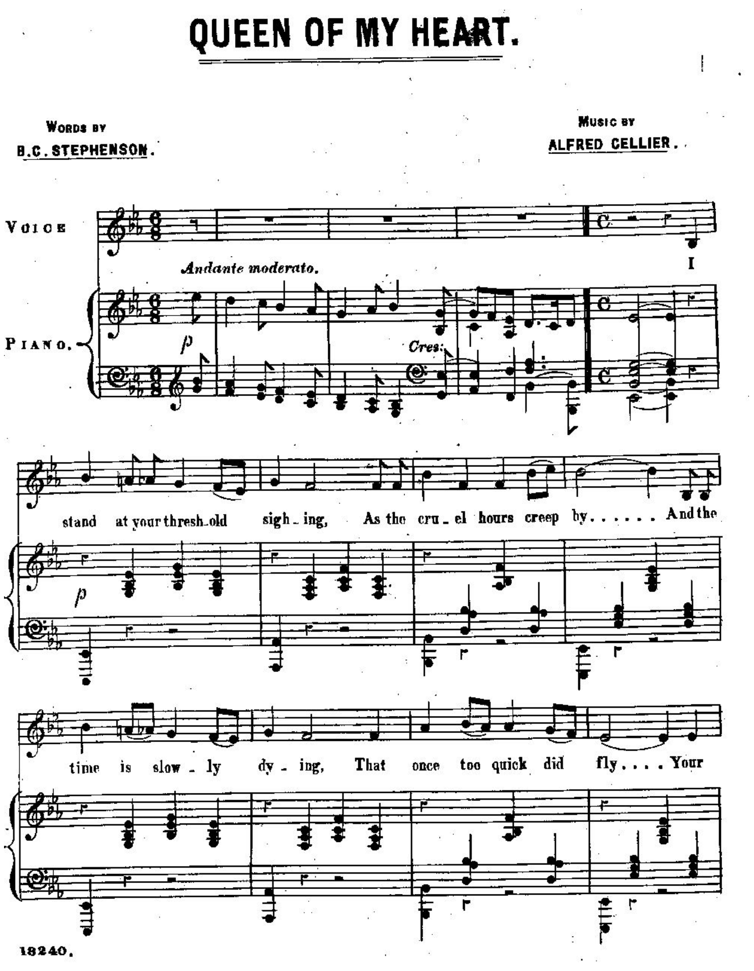

QUEEN OF MY HEART.
WORDS BY
B.C. STEPHENSON.
MUSIC BY
ALFRED CELLIER.
VOICE
PIANO.
Andante moderato.
Cres:
I
stand at your thresh-old sigh-ing, As the cru-el hours creep by...... And the
time is slow-ly dy-ing, That once too quick did fly.... Your
18240.

Figure 2.1.1.1 (https://en.wikipedia.org/wiki/Sheet_music#/media/File:QoMH.png)

2.1.2. Pitch

Pitch denotes the frequency of a music note. Pitches in music are mapped to distinct frequency and given different note names for simplicity.

2.1.2.1. Note and Sharp, Flat

In music, {C, D, E, F, G, A, B} are used to represent pitches while an octave number in range [-1, 9] may be added behind to distinct two pitches having the same note name. For instance, C4(~261Hz) is the 'middle C' on the music keyboard and C5(~523Hz) is another note named C.

Besides the seven note name used, a total of 5 symbols are used to alter the note by 'semitones'. A semitone is the smallest unit of interval in music having frequency $f_n = \sqrt[12]{2} f_0$, where f_n is the frequency of original note altered higher by one semitone and f_0 is the original frequency. The five symbols used are: 𝄪 ♯ ♮ ♭ 𝄫 (double sharp, sharp, natural, flat, double flat) where note is altered by two semitones higher, altered by one semitone higher, restored to original position, altered by one semitone lower and altered by two semitones lower.

2.1.2.2. Interval

Interval is the word used in music to describe the different between notes.

Usually, the interval of two notes is one whole tone, which is composed of two 'semitones'. However, there are two exceptions. The two set of notes {E, F} and {B, C} have interval of one semitone instead of two.

Musicians had named common used intervals with different names and using these intervals would be in great help while developing chord patterns. Please refer to the intervals at Appendix A.

2.1.2.3. Enharmonic notes

As notes can be altered higher or lower by at most 2 semitones, pitches can be expressed by different notes. For instance, C flat and B are sharing the same pitch, and they are called enharmonic notes.

In an octave, there are only 12 different pitches but yet total of 35 (=7 notes * 5 altering) note names. As a result, every pitches except one has 3 enharmonic notes. Knowing the occurrence of enharmonic notes will be a great help in building the data structure.

2.2. Beat

Beat in music is the basic unit of time, the pulse, of the mensural level. Understanding the beat of music would be a key to understand not only the time attribute of the piece but also the rhythm pattern and chord pattern.

2.2.1. Time Signature

Time signature is the starting point of the discussion about beat and time which is marked on the start of every score (Figure 2.2.1.1). It is formed by two numbers -- the upper one denotes the number of beats and the below one define the unit of each beat. With the information of time signature, the length of each measure is specified.

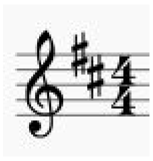


Figure 2.2.1.1

2.2.2. On-Beat and Off-Beat

Musicians generally emphasis beat by assigning important note on the instant of the beat (known as on-beat). Therefore, important elements such as chord (see 2.4) change are often on-beat while less important parts would be off-beat (contrary to on-beat).

2.3. Key

Among the 12 pitches within an octave, not all of them would be used in a music pieces. Therefore, a key specific a set of notes that follow specific pattern that would be used commonly in a piece. However, even notes that not specify in a key may occasionally appear in a piece.

In traditional western music, mainly two types of key, major key and minor key, are mainly used and they follow very similar pattern, including a key signature and 7 notes in the key.

2.3.1. Key Signature

Key signature are shown at the very beginning of each line and every time before a new key is applied to the piece. The key signature act as an important indicator of both key using and notes to be altered. Consider the key signature of A major/ f# minor (Figure 2.2.1.1), there are total 3 sharps (F#, C# and G#), indicating that every F, C and G note should be altered higher by one semitone throughout the parts using this key signature.

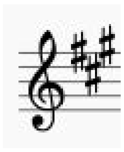


Figure 2.3.1.1

2.3.2. Major, Minor

Major and minor are the most common key types used in western traditional music. Both key types contain 7 notes in an octave but having different intervals between notes.

For each key signature, it can be matched to one major and one minor, where the interval between the tonic note, the first note, of the key is a minor 3rd (Please refer to Appendix A for the meaning of minor 3rd).

To simplify the discussion, C major and c minor would be focus in the remaining of the section.

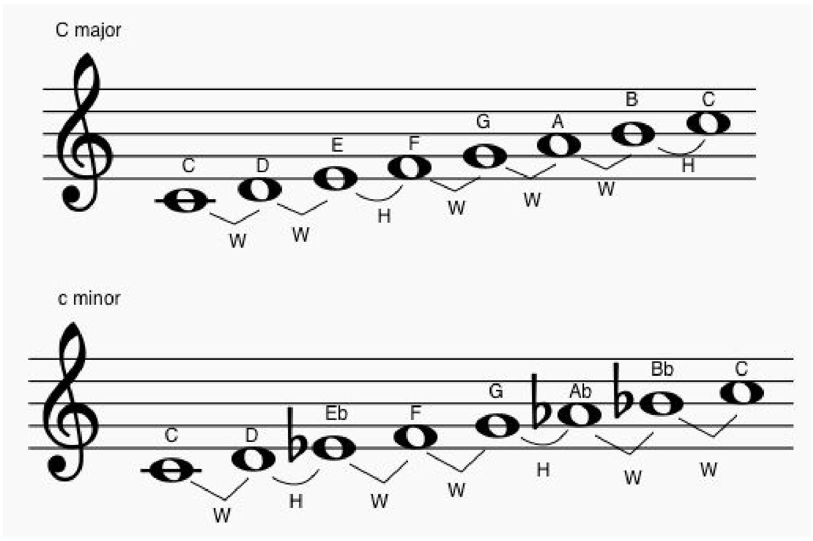


Figure 2.3.2.1 Scales of C major and c minor

Figure 2.3.2.1 shows the scales, which ordered the notes included in the specific key in increasing pitch. The ‘W’ and ‘H’ indicates the intervals between the two notes where ‘W’ for whole tone (2 semitones) and ‘H’ for half tone (1 semitones). In any scale of major and minor, there exists 5 whole tones and 2 half tones with position same as denoted in Figure 2.3.2.1.

The scales of major and minor provide the fundamental of chords formation which will be further discussed in section 2.3.

2.4. Chord

Two or more notes sounded simultaneously are known as a chord. (Ottó , 1991) Chords are formed by adding two or more distinct frequency to produce harmonic effects, which vivid the expressiveness of music. Therefore, understanding the chords of piece is essential to analysis a music score and undergo reduction process.

In chords, notes may often given other names in while discussing chords. For instance, the fundamental note that chords are built on would name as Root, while other notes would name as second, third, so on and so forth depending the note difference from the root. If C major chords are being studied, the second would be D, the third would be E.

2.4.1. Common Chord Type

Chords can be categorized into different type defined by the intervals of notes within the chord. Common chord types are:

Major chord: root, M3 (note from root) and P5 (note from root)

Minor chord: root, m3, P5

diminish chord: root, m3, d5

-7 chord: adding m7 or M7 to the original chord (depends on the type of original chord)

Aside from the above chords, there are a few specific chords that have stricter rules. All chords that could be generated in major and minor are summarized into a table at Appendix B.

2.4.2. Romanization

If chords are to be discussed in the context of a key, chords can be represented in roman number. Romanization of chords enhance the readability for the discussion of 'chord progression' . Chord progression is a topic on relation of chords and gives meaning to chords or harmonic line.

Chords are romanized based on the position of the root in the scale. If the root is in the 3rd position of the scale, roman number III would be used to describe the chord.

2.4.3. Inversion

Chords may come not in ordering, i.e. having root note as base (root position of chord). If such a case happens, chord is said to be inversed. An inverted chord may serve as another function despite of the function of the original chord. Therefore, identifying an inversion and separating it from the original chord is important in studying chord progression.

For inversion chords, musicians may use figured bass to denote the difference.

For chords having 3 notes, if the third note is ordered in the base (first inversion), chords are added with an arabic number 6 behind the chord while if the fifth note is ordered in the base (second inversion), arabic number 64 is added behind the chord.

For chords having 4 notes, first inversion would add 65 behind the chord name, second inversion adding 43, while if the seventh note is the base (third inversion), 42 would be added.

2.4.4. Harmonic Note and Non-Harmonic Note

Chords are formed by several distinct frequencies that produce harmonic effect. These notes are referred to harmonic notes. Although the present of non-harmonic note need to be resolved, which referred to the process that non-harmonic notes pass to harmonic notes to release the tension built, musicians often include some notes that are not part of the chord -- non-harmonic note to build and embellish the melody (Joutsenvirta and Perkiömäki). Moreover, non-harmonic notes may used as tonic effect to establish a tension back to the desired note ("Pedal Point").

The present of non-harmonic note harden the question to identify the suitable chord as these notes may mess up with other harmonic notes to produce another chords. Therefore, knowing the characteristics of non-harmonic notes is essential to eliminate them from the score for chord recognition. In this chacter, different types of non-harmonic notes will be discussed.

2.4.4.1. Pedal Point

Pedal point is a sustained note in music having different from the other form of non-harmonic notes in terms of function and means to resolve. Unlike the rest of the non-harmonic notes, pedal points have longer duration. They usually last for a significant time; in most of the time, pedal points can last for bars and cross through several chords.

Pedal point is generally resolved by forming a chord with pedal point as a harmonic note whilst other type usually resolved by passing to a harmonic note.

2.4.4.2. Passing Note

Passing notes are notes that bridge two harmonic notes. It is used generally in falling or rising melody which take place on off-beat. Passing note is resolved by passing to a harmonic note. In Figure 2.4.4.2.1, the F note is the passing note which bridge the E note and the G note within the C chord.

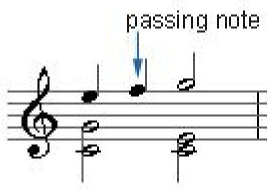


Figure 2.4.4.2.1

2.4.4.3. Neighbouring Note

Neighbouring note is identical to passing note except for the resolving method. Neighbouring note take a stepwise from the note before and return to the origin note.

In Figure 2.4.4.3.1, the F note is the neighbouring note where stepwise from E and return to E.

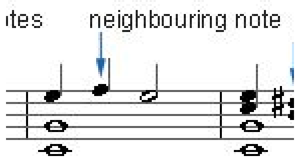


Figure 2.4.4.3.1

2.4.4.4. Anticipation

Anticipation is a non-harmonic note preparing for the coming chord which usually happens on off-beat. Harmony note of the next chord is played before the chord change and is resolved when the chord changed. Anticipation create tendency for the next chord.

In Figure 2.4.4.4.1, the sixteenth C prepares the chord change from Bm to C.

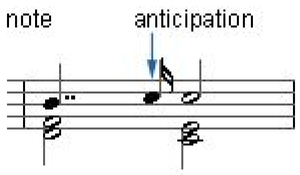


Figure 2.4.4.4.1

2.4.4.5. Suspension

Suspension is the opposite of anticipation. By suspending one of the previous chord's harmonic note and resolves by take a stepwise down to the new chord's harmonic note.

In Figure 2.4.4.5.1, the D note is resolved to C note of the C chord. Unlike other non-harmonic note types discussed, suspension usually place on on beat and therefore is accented.

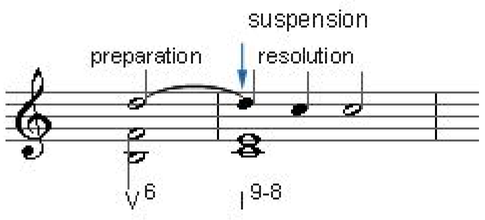


Figure 2.4.4.5.1

2.4.4.6. Appoggiatura

The last commonly used non-harmony note is appoggiatura. It is also put on on beat and accented. There are two types of appoggiatura. The first type is alike suspension but lack of the preparation process (to hold a note from the previous chord) like the D half note in Figure 2.4.4.5.1.

In Figure 2.4.4.6.1, the D fourth note is a non-harmonic note from the previous G chord. However, it is not prepared in the previous chord.

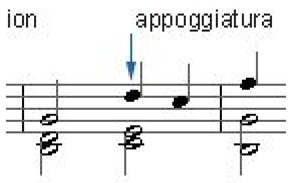


Figure 2.4.4.6.1

The other type of Appoggiatura is in form of grace note (a small note before the big note). These grace note usually share part of the time the big note and is accented.

In Figure 2.4.4.6.2, the small G note is an appoggiatura and is resolved to the F note.

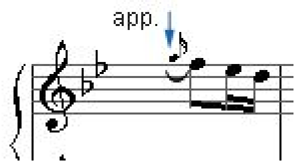


Figure 2.4.4.6.2

All in all, non-harmonic note on one hand increase the expressiveness of the music, it will also increase the difficulties in choosing the correct interval for chord analysis as it may happens both on beat and off beat.

2.5. Progression

Although chords are concerning the harmonic effect of instances, chords are inter-related. Different chords are succeeded to each other to build tension and resolve the tension. However, not every chords can be progressed from any chords. Some of the regulations in chord progression would be introduced.

2.5.1. Chord Function

In western tonal music, chords are given different functions depending the relation between tonic (the first note of the key) and other notes constructing the chords. These function define the purpose of chords -- either establishing or contradicting the tonality (the sense of the key) ("Chord Progression").

Under the study of chord function (also referred to diatonic function), chords are mainly classified into three types: tonic (T), subdominant (S) and dominant (D). The names are given by the note that act as the main function. Thus, in tonic chord, the most important note is the tonic (the first note in the key); subdominant (the 4th note) for subdominant chord and dominant (the 5th note) for dominant chord.

Tonic chord among all chords is the most stable chord and can express the tonality of the piece most. Therefore, pieces usually start and end at tonic chord. Moreover, tension is released by resolving to tonic chord in most of the case.

Dominant chord, on the other hand, is the most unstable chord due to the leading tone (the 7th note) which is the most unstable note and has a tendency to the tonic. Accordingly, dominant chords are usually the highest tension and resolve to tonic to release. However, dominant chords may also resolve to tonic chords through subdominant chords.

Subdominant chord mostly act as transition chord which connect dominant chords and tonic chords.

Understanding chord function support the design of chord recognition algorithm; for instance, it could be a hint to eliminate chords that not suitable in the sense of harmonic effect.

3. Technical Support

In this section, numbers of important technology and library which were used in this project will be introduced.

3.1. MusicXML

3.1.1. Introduction

Extensible Markup Language(XML) is a restricted form of SGML, the Standard Generalized Markup Languages. XML documents contains many entities, which are the storage units of parsed or unparsed data, and XML provides constraints on storage layout and logical structure.

MusicXML is a standard open format for sheet music file in XML form. MusicXML was designed to share sheet music between software and the files are readable and usable by a wide range of music score software, for instance, Finale and Musescore.

We are going to focus on the MusicXML sheet music file throughout our project due to the fact that it is widely used in digital music applications and by internet users. There are also libraries for reading and editing MusicXML files for different programming languages, for example, Python and Javascript. This is also one of the reason why Python was chosen for implementing the reduction algorithm. We will talk more about Python at below.

3.1.2. Overview

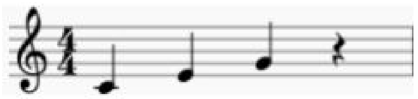
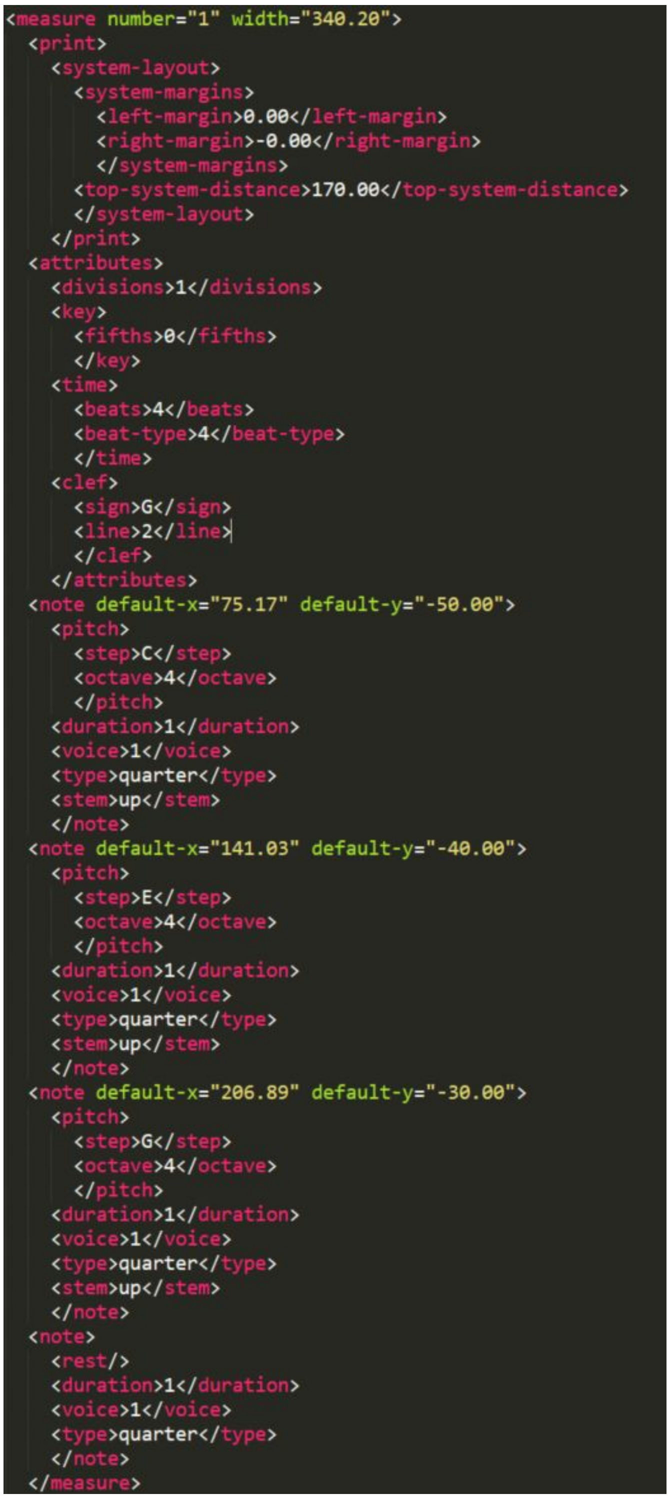


Figure 3.1.2.1



```
<measure number="1" width="340.20">
  <print>
    <system-layout>
      <system-margins>
        <left-margin>0.00</left-margin>
        <right-margin>-0.00</right-margin>
        </system-margins>
      <top-system-distance>170.00</top-system-distance>
      </system-layout>
    </print>
  <attributes>
    <divisions>1</divisions>
    <key>
      <fifths>0</fifths>
      </key>
    <time>
      <beats>4</beats>
      <beat-type>4</beat-type>
      </time>
    <clef>
      <sign>G</sign>
      <line>2</line>
      </clef>
    </attributes>
  <note default-x="75.17" default-y="-50.00">
    <pitch>
      <step>C</step>
      <octave>4</octave>
      </pitch>
    <duration>1</duration>
    <voice>1</voice>
    <type>quarter</type>
    <stem>up</stem>
    </note>
  <note default-x="141.03" default-y="-40.00">
    <pitch>
      <step>E</step>
      <octave>4</octave>
      </pitch>
    <duration>1</duration>
    <voice>1</voice>
    <type>quarter</type>
    <stem>up</stem>
    </note>
  <note default-x="206.89" default-y="-30.00">
    <pitch>
      <step>G</step>
      <octave>4</octave>
      </pitch>
    <duration>1</duration>
    <voice>1</voice>
    <type>quarter</type>
    <stem>up</stem>
    </note>
  <note>
    <rest/>
    <duration>1</duration>
    <voice>1</voice>
    <type>quarter</type>
    </note>
  </measure>
```

Figure 3.1.2.2

From above, an example of a measure consists of 3 notes and 1 rest is shown. A MusicXML file exported from the above measure is shown in Figure 3.1.2.2, we can find that the XML file includes all of the important data inside a measure.

Inside the “attributes” entity, we can see that the “division”, “key signature”, time signature” and “clef” are clearly stored.

Afterwards, we can find that there are 4 note entities. Each of them consist of the information of pitches, durations and types for different notes.

3.2. Music21

3.2.1. Introduction

Music21 is a computer-aided musicology toolkit library written in Python. It is a project created by a team in M.I.T., which was lead by Michael Cuthbert, Christopher Ariza, Benjamin Hogue, Josiah Wolf Oberholtzer.

Music21 is useful for creating music scores, editing musical element objects in a programmatic way. It also supports numbers of sheet music file type including MusicXML, which is our main focus file type of our project.

By using Music21 library, we can create or read music scores and musical elements more easily as the library treats music elements in an Object-Oriented Programming way. Specific musical elements can be spotted and altered rapidly and conveniently in such programming way, which will be a great help on extracting, generating, editing and analyzing music scores.

After parsing a input sheet music file, all elements can easily be accessed by a few lines of Python code.

3.2.2. Overview

An example of accessing the musical elements using Python and Music21 is captured in the following.

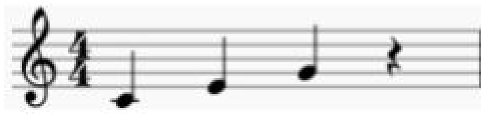
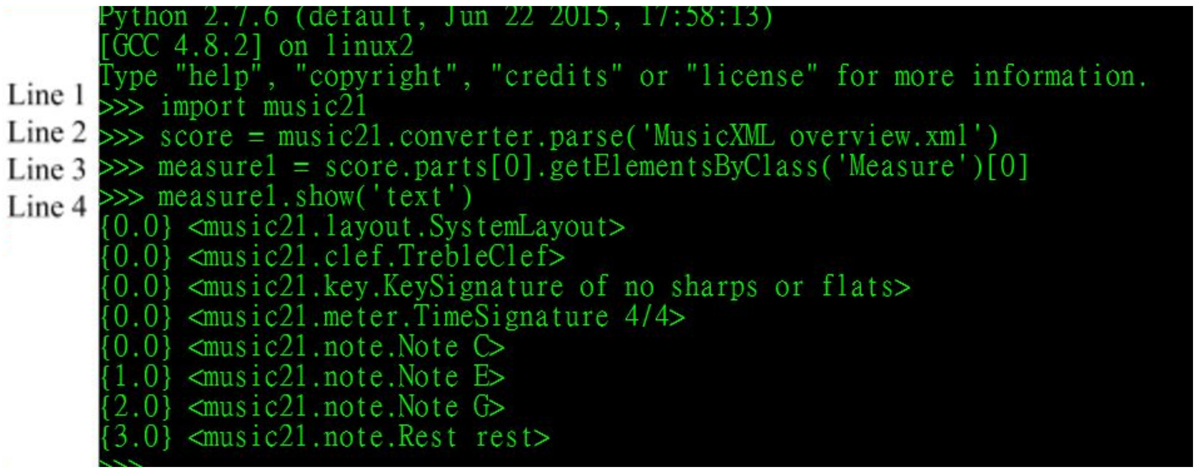


Figure 3.1.2.1

Recalling the sample measure in section 3.1.2, we can see that this is a measure with a treble clef, 4/4-time signature, 3 notes and 1 rest note included. Below are the simple basic procedures to access musical elements using Music21.



```
       Python 2.7.6 (default, Jun 22 2015, 17:58:13)
       [GCC 4.8.2] on linux2
       Type "help", "copyright", "credits" or "license" for more information.
Line 1 >>> import music21
Line 2 >>> score = music21.converter.parse('MusicXML overview.xml')
Line 3 >>> measure1 = score.parts[0].getElementsByClass('Measure')[0]
Line 4 >>> measure1.show('text')
       {0.0} <music21.layout.SystemLayout>
       {0.0} <music21.clef.TrebleClef>
       {0.0} <music21.key.KeySignature of no sharps or flats>
       {0.0} <music21.meter.TimeSignature 4/4>
       {0.0} <music21.note.Note C>
       {1.0} <music21.note.Note E>
       {2.0} <music21.note.Note G>
       {3.0} <music21.note.Rest rest>
       >>>
```

Figure 3.2.2.1

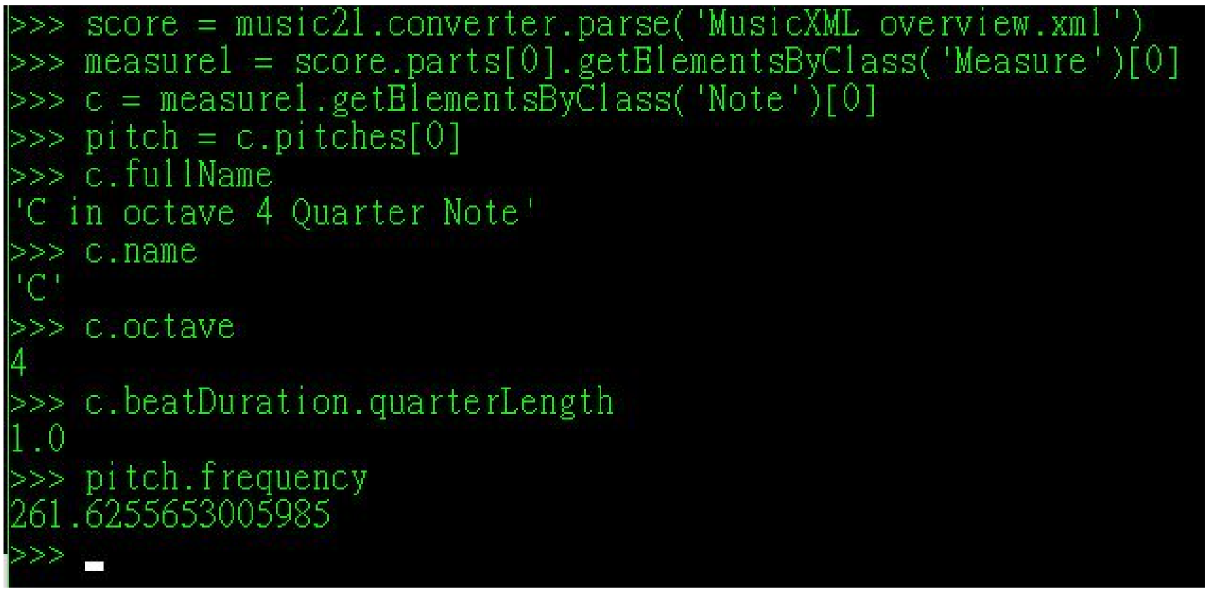
Firstly, the Music21 library must be imported in Python to access the functions in the library.

Then, in line 2, the sheet music file in MusicXML format was parsed and return as an object called “score”.

In line 3 and 4, the first measure was returned as object and showed in text form in the command line window.

In the text output of the measure object, a list of musical elements was shown in different type of Music21 library objects. On the left there were also offset of these objects, which are important when analyzing music scores.

After getting access to a specific measure of a music score, notes or other musical elements can also be accessed.



```
>>> score = music21.converter.parse('MusicXML overview.xml')
>>> measure1 = score.parts[0].getElementsByClass('Measure')[0]
>>> c = measure1.getElementsByClass('Note')[0]
>>> pitch = c.pitches[0]
>>> c.fullName
'C in octave 4 Quarter Note'
>>> c.name
'C'
>>> c.octave
4
>>> c.beatDuration.quarterLength
1.0
>>> pitch.frequency
261.6255653005985
>>> _
```

Figure 3.2.2.2

When exploring inside a Music21 note object, we can get basic information of a note by reading the object’s instance variables, such as note name, octave, beat length, frequency etc. Note name, duration and frequency are essential in the chord identification algorithm, and Music21 library provides a great solution in accessing these data in a convenient way. We will go through the usage of these elements in identifying chords later in this report.

3.3. Python

Python, which was created in the early 1990s by Guido van Rossum, is a programming language developed based on a language called ABC. Python can run in both interpreted and compiled environments. There are no type declarations for variables, parameters, methods or functions. These flexibilities make Python codes shorter and cleaner, which is convenient for rapid prototyping. Python, as a result, is now a popular programming language.

In our project, Python was chosen as the programming language of the chord identification algorithm due to a few reasons. One of them is that Music21, which was introduced in the previous section, is used by the former CUHK students for developing the piano reduction algorithm. In order to integrate the chord identification algorithm into the reduction program eventually, Python would be the only choice. Moreover, Music21 also provides convenient functions for us to analyze musical elements, which is essential in chord identification.

3.4. Previous Work

As mentioned above, the ultimate goal in this project is to develop new rules in an automatic piano reduction algorithm, which is developed previously by former CUHK students. In this section, we will briefly introduce the developed reduction algorithm.

The existing reduction algorithm takes 3 types of input, target, sample input and sample output. Sample input and sample output can be lists of MusicXML files to provide multiple samples, which are used for training the neural network in the reduction algorithm. After analyzing the sample input and output files, a well-trained neural network will be generated and will be used for creating a reduced music scores from the target input.

The reduction algorithm was developed with various musical rules as input of the generated neural network. Our goal currently is to develop a new musical rule, chord identification, to enhance the accuracy and the performance of the reduction algorithm.

With the aid of the previously developed program, we can reference methods and approaches of accessing musical elements. Moreover, in the near future of next semester, we hope that we can try inserting the chord identification rule into the reduction algorithm to test the accuracy and efficiency of the chord recognition approaches.

4. Design and Implementation

4.1. First Term Work and Modification

In first term, a class ChordRecognizer was designed and implemented to identify chords. Inside, many pre-calculated dictionaries were used to convert different music notations or languages into numbers that allow further calculation. For further details, please refer to the first term report. However, in the second term, the ChordRecognizer class and implementation method was replaced by a whole new structure, which will be discussed in the following sections.

In second term, the whole program has been modularized in order to be plugged to the reduction algorithm. Moreover, throughout the original main algorithm in first term, extracted score data and recognized results are simply stored in lists and dictionaries. In order to ease the implementation of the whole chord identification algorithm and future development, major structural objects in main identification algorithm has been fully modularized. In the following section, the new main module, ChordIdentifier, and the newly designed classes will be introduced.

4.2. Main Module Flow

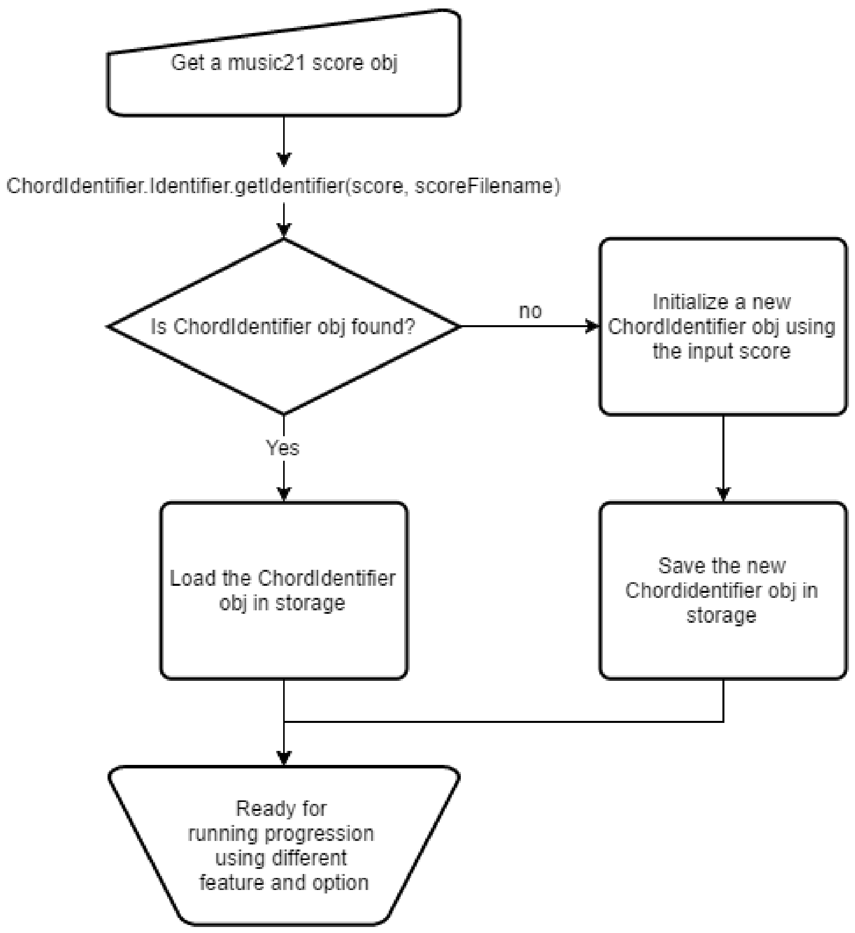

Get a music21 score obj
ChordIdentifier.Identifier.getIdentifier(score, scoreFilename)
Is ChordIdentifier obj found?
no
Initialize a new ChordIdentifier obj using the input score
Yes
Load the ChordIdentifier obj in storage
Save the new Chordidentifier obj in storage
Ready for running progression using different feature and option

Figure 4.2.1 Main Module Flow

The main flow of the ChordIdentifier module is as above. When a music21 score objects is input to the Identifier, which is the main class of ChordIdentifier module, a checking on the score filename will be applied to find a saved ChordIdentifier file in the storage folder. If there exist a saved file, the Identifier class will return a loaded Identifier object. Otherwise, a newly initialized Identifier object will be saved and returned.

An Identifier object stored results on basic chord analysis algorithm, which is all chords that could be the correct answer for every ChordInterval. After that, by running the progression algorithm in different modes and using different features, the module is hoped to be able to find out a correct chord progression path from all of the possibilities.

4.3. ChordIdentifier Storage File

It is found that reading scores and running basic chord analysis is time consuming when initializing an Identifier object, which details will be introduced later in section 4.4. Due to the fact that customized chord progression analysis will not be applied before the score reading and chord analysis process, a storing function is implemented in order to reduce the time cost when reloading a same music score.

The ChordIdentifier object will be serialized using python native library, cPickle, and stored under ./Storage/ folder named with the input score filename and a file extension of ".ChordIdentifier" for the purpose of restoring.

4.4. Identifier Class

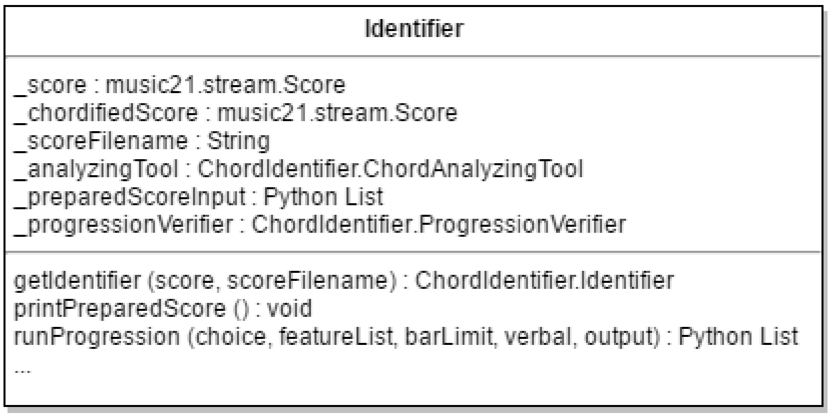
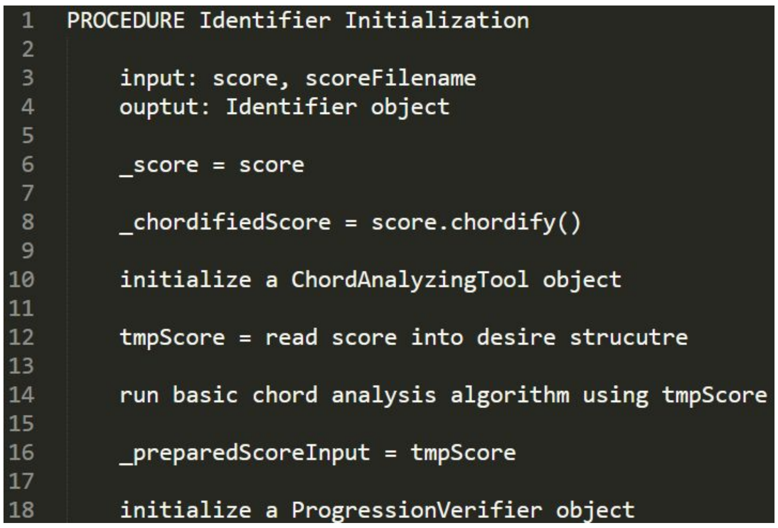

```
Identifier
_score : music21.stream.Score
_chordifiedScore : music21.stream.Score
_scoreFilename : String
_analyzingTool : ChordIdentifier.ChordAnalyzingTool
_preparedScoreInput : Python List
_progressionVerifier : ChordIdentifier.ProgressionVerifier
getIdentifier (score, scoreFilename) : ChordIdentifier.Identifier
printPreparedScore () : void
runProgression (choice, featureList, barLimit, verbal, output) : Python List
...
```

Figure 4.4.1 Identifier UML

Identifier class is the main class of the ChordIdentifier module. The class is responsible for storing score information, score objects, result of basic analysis, and some initialized objects for later analysis. For instance, ProgressionVerifier object will be initialized in order to get ready for running progression. It is important to initialize the verifier as an object and reuse it for multiple progression analysis as dynamic programming algorithm is applied inside the progression algorithm, which will be discussed later in section 4.9.

Besides, a "chordified" score will also be stored in the object. Chordify() is a function provided in music21, which combines all simultaneous music notes into Chord objects and return as a new Score object. However, it is found that the provided function is very time consuming, therefore the storage function in section 4.3 was introduced and implemented. Storing the "chordified" score can prevent multiple calling for the Chordify() function and result in faster chord analysis.

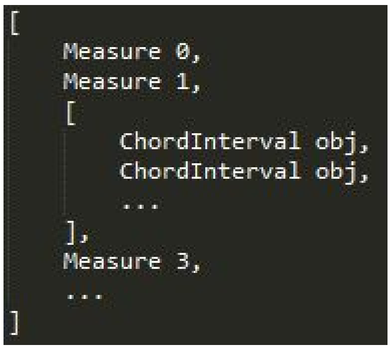
4.4.1 Initialization Flow



```
1   PROCEDURE Identifier Initialization
2
3       input: score, scoreFilename
4       ouptut: Identifier object
5
6       _score = score
7
8       _chordifiedScore = score.chordify()
9
10      initialize a ChordAnalyzingTool object
11
12      tmpScore = read score into desire strucutre
13
14      run basic chord analysis algorithm using tmpScore
15
16      _preparedScoreInput = tmpScore
17
18      initialize a ProgressionVerifier object
```

Figure 4.4.1.1 Identifier Initialization Pseudocode

In line 12, the desire structure of the score is a nested list of ChordIdentifier.ChordInterval object. It will be a list representing measures and inside each measure, there will be a list of ChordInterval object.



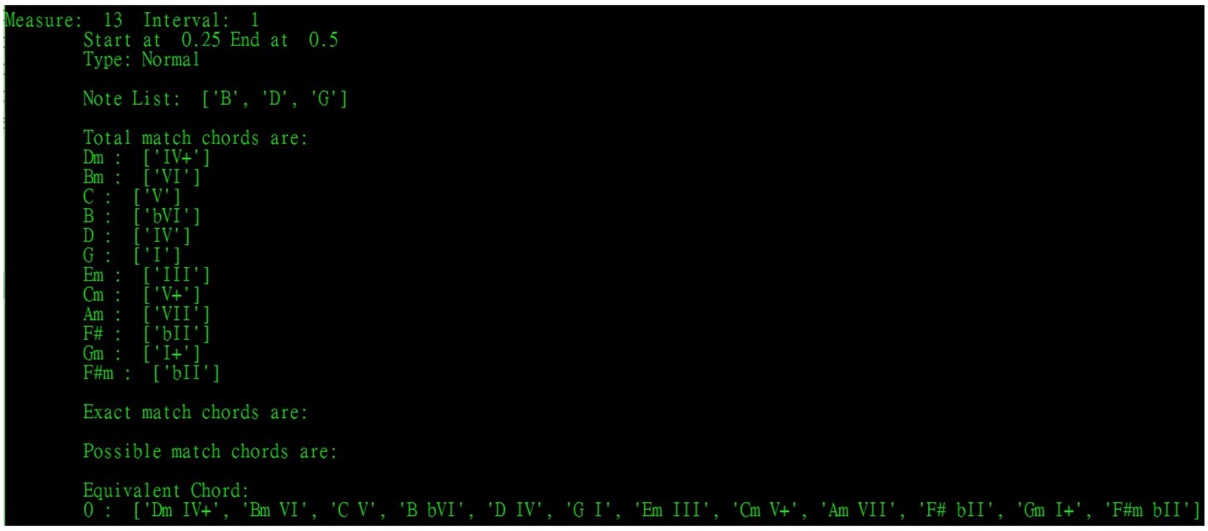
```
[
    Measure 0,
    Measure 1,
    [
        ChordInterval obj,
        ChordInterval obj,
        ...
    ],
    Measure 3,
    ...
]
```

In line 14 and 16, the basic chord analysis algorithm will call the chord analyzing functions for each ChordInterval objects in the list and store the result back inside the ChordInterval object. The nested list of ChordInterval objects will finally be well prepared and stored in _preparedScoreInput.

The chord analyzing functions are provided in the ChordAnalyzingTool class and ChordInterval class, which will be discuss later in section 4.5.1 and 4.6.3 respectively.

4.4.2 printPreparedScore

This is a function for printing the __preparedScoreInput list to stdout.



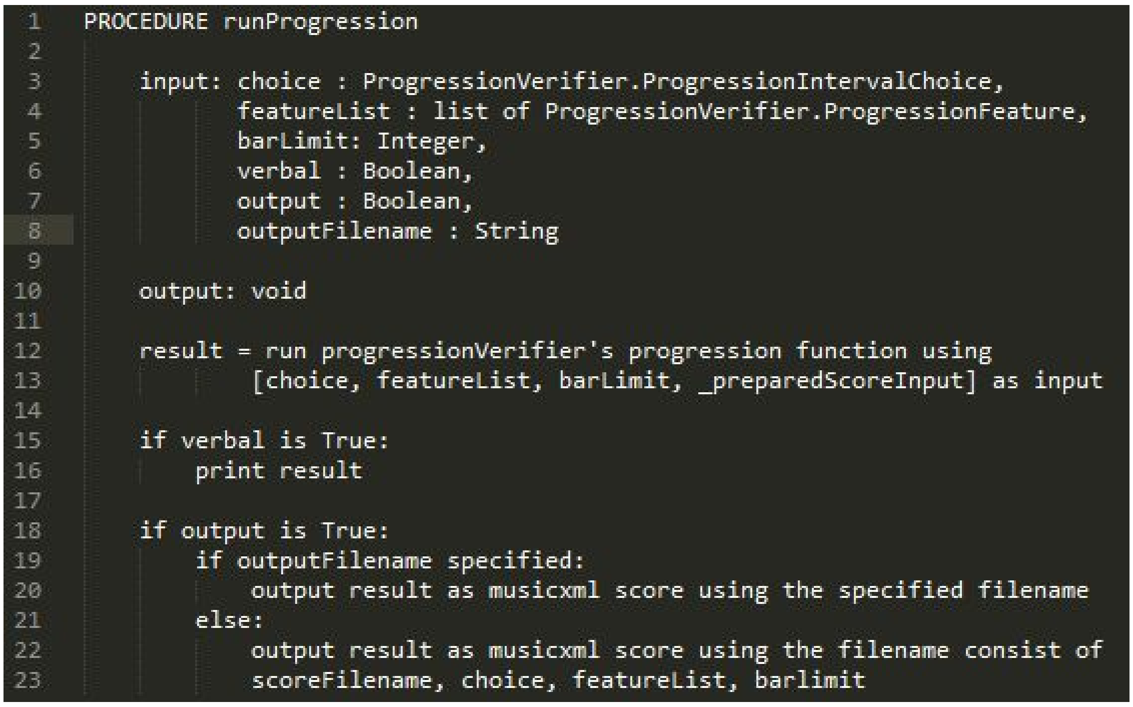
```
Measure:  13  Interval:  1
        Start at  0.25 End at  0.5
        Type: Normal

        Note List:  ['B', 'D', 'G']

        Total match chords are:
        Dm :  ['IV+']
        Bm :  ['VI']
        C :  ['V']
        B :  ['bVI']
        D :  ['IV']
        G :  ['I']
        Em :  ['III']
        Cm :  ['V+']
        Am :  ['VII']
        F# :  ['bII']
        Gm :  ['I+']
        F#m :  ['bII']

        Exact match chords are:

        Possible match chords are:

        Equivalent Chord:
        0 :  ['Dm IV+', 'Bm VI', 'C V', 'B bVI', 'D IV', 'G I', 'Em III', 'Cm V+', 'Am VII', 'F# bII', 'Gm I+', 'F#m bII']
```

Figure 4.4.2.1 Sample of printPreparedScore()

The above image shows one of the sample output of the function. The printing for each ChordInterval object include measure number, interval number, list of notes and some basic analysis result.

4.4.3 runProgression

This is a function for driving the ProgressionVerifier to run a progression analysis.



```
1   PROCEDURE runProgression
2
3       input: choice : ProgressionVerifier.ProgressionIntervalChoice,
4              featureList : list of ProgressionVerifier.ProgressionFeature,
5              barLimit: Integer,
6              verbal : Boolean,
7              output : Boolean,
8              outputFilename : String
9
10      output: void
11
12      result = run progressionVerifier's progression function using
13              [choice, featureList, barLimit, _preparedScoreInput] as input
14
15      if verbal is True:
16          print result
17
18      if output is True:
19          if outputFilename specified:
20              output result as musicxml score using the specified filename
21          else:
22              output result as musicxml score using the filename consist of
23              scoreFilename, choice, featureList, barlimit
```

Figure 4.4.3.1 runProgression Pseudocode

In line 12 and 13, the progression function of ProgressionVerifier object will be further introduced in the section 4.9.

4.5. ChordAnalyzingTool Class

In first term, when main driving program is running, numbers of ChordRecognizer object are first initialized by different given tonics. It is found that numbers of important calculated information in different ChordRecognizer objects, such as ChordTypeDictionary and RomanDictionary, is useful and reusable in the afterward process, such as equivalent chord calculation. In order to simplify the procedure of access in the afterward process, a new class ChordAnalyzingTool is designed for storing all of the above basic informational dictionaries and act as a substitution of the original ChordRecognizer class.

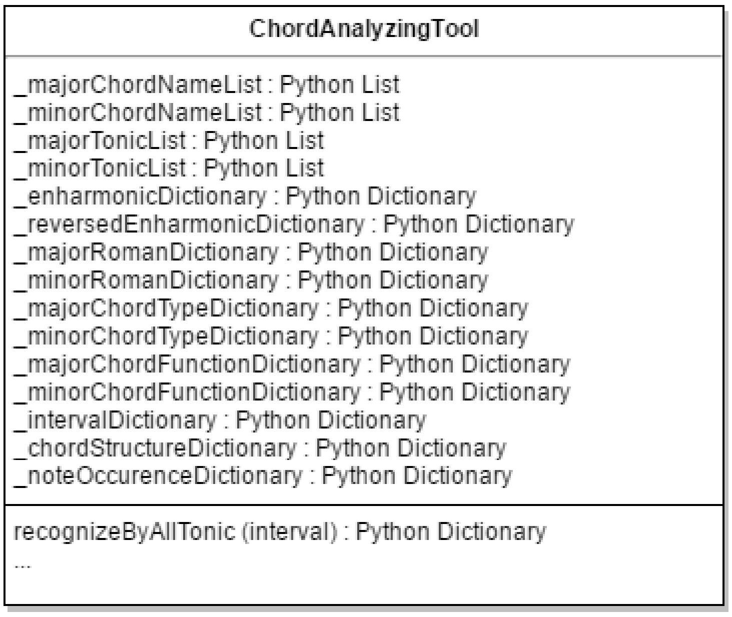
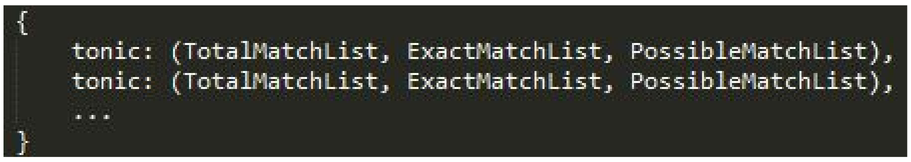

ChordAnalyzingTool
_majorChordNameList : Python List
_minorChordNameList : Python List
_majorTonicList : Python List
_minorTonicList : Python List
_enharmonicDictionary : Python Dictionary
_reversedEnharmonicDictionary : Python Dictionary
_majorRomanDictionary : Python Dictionary
_minorRomanDictionary : Python Dictionary
_majorChordTypeDictionary : Python Dictionary
_minorChordTypeDictionary : Python Dictionary
_majorChordFunctionDictionary : Python Dictionary
_minorChordFunctionDictionary : Python Dictionary
_intervalDictionary : Python Dictionary
_chordStructureDictionary : Python Dictionary
_noteOccurenceDictionary : Python Dictionary
recognizeByAllTonic (interval) : Python Dictionary
...

Figure 4.5.1 ChordAnalyzingTool UML

The ChordAnalyzingTool is initialized with numbers of hard-coded and calculated python dictionary base on musical theories. They are extremely useful in running further analysis.

4.5.1. recognizeByAllTonic()

This function is implemented to improve and replace the old ChordRecognizer in first term. It adopted the newly implemented ChordInterval object and make use of the newly designed dictionaries in the ChordAnalyzingTool. The improved algorithm limits the result in several ways which enhance the accuracy. Besides, the function also return a new structure of results, which is a Python Dictionary as following,



```
{
    tonic: (TotalMatchList, ExactMatchList, PossibleMatchList),
    tonic: (TotalMatchList, ExactMatchList, PossibleMatchList),
    ...
}
```

The result dictionary uses tonic as keys and tuple consists of three match lists as values. Each match list consist of a list of tuple, which is named “Match Tuple”.

4.5.1.1. Algorithm Flow

The algorithm is much alike the previous implemented one in semester 1 except for the following modifications:

1. Redefined possible match and exact match while adding a new match type: Total match
2. Constrainted chords must have its root in order to be recognized
3. Major key chords and minor key chords are seperated for simplier progression hangle

4.5.1.2. Match Type

As mentioned in the previous section, match type are redefined. The new definition and implementation method would be covered in this section.

Total Match:	All notes in the specific interval is used to build the chords and all notes in the chord is present in interval
Exact Match:	All notes in the chord is present in the notes from the specific interval but there is/are note(s) that is/are not matter with the construction of the chord
Possible Match:	Notes in the specific interval can partially construct the chord by missing one note from the chord

Nevertheless, there are several limitations in order to eradicate some impossible chord, for instance, if a I+ chord is found total match in Cm, I chord would never be possible match although C, G can form I chord in Cm partially. The rules are as below:

1. If total match is found, and the total match chord contains 3 notes, neglect all results in exact match and possible match; if total match chord contains 4 notes, only neglect all results in possible match but reserves results in exact match.
2. If exact match is found, neglect results in possible match having the same note as root note as the exact match chord.
3. In possible match, for all 7th chord (please refer to the chord type in Appendix B), neglect result if the missing note is the 7th note.

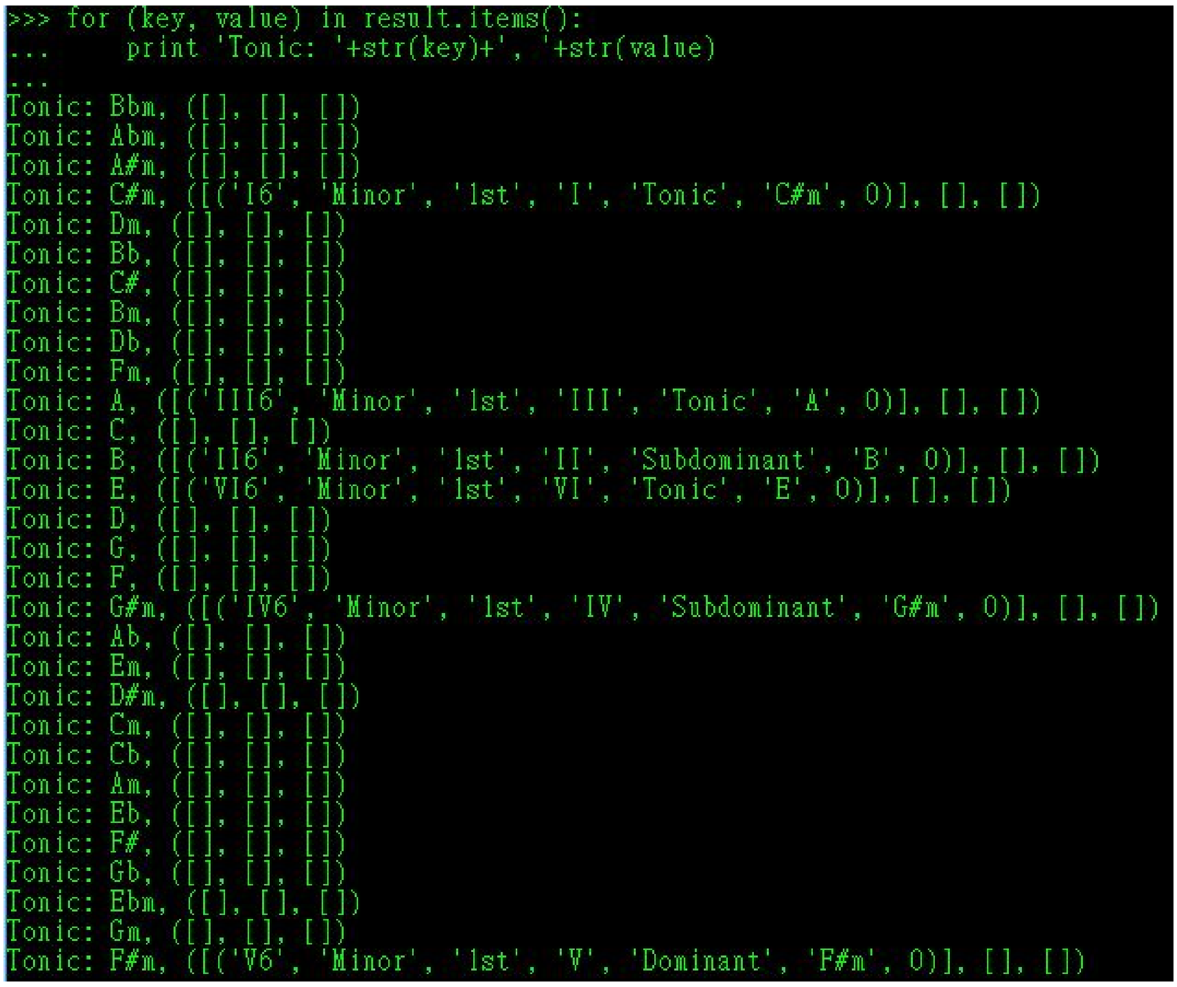
Under these constraints, the program could ensure minimized the number of chords recognized while taking all possibilities into account.

4.5.1.3. Match Tuple

Each Match Tuple stores important information of a possible chord result, which is produced by the basic chord analysis algorithm. The Match Tuple are all in the following structure,

```
(chord name, chordType, inversion, roman, chord function, tonic, group no)
```

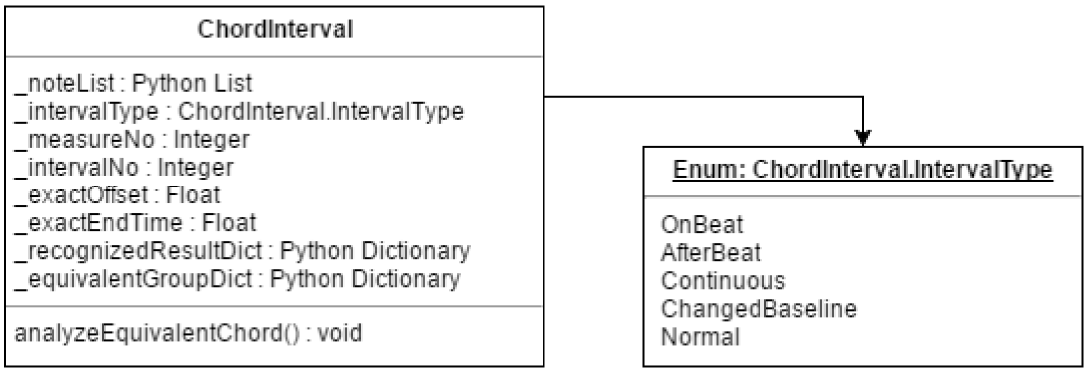
and widely used in the later on analysis. The following image will show a sample result structure.



```
>>> for (key, value) in result.items():
...     print 'Tonic: '+str(key)+', '+str(value)
...
Tonic: Bbm, ([], [], [])
Tonic: Abm, ([], [], [])
Tonic: A#m, ([], [], [])
Tonic: C#m, ([('I6', 'Minor', '1st', 'I', 'Tonic', 'C#m', 0)], [], [])
Tonic: Dm, ([], [], [])
Tonic: Bb, ([], [], [])
Tonic: C#, ([], [], [])
Tonic: Bm, ([], [], [])
Tonic: Db, ([], [], [])
Tonic: Fm, ([], [], [])
Tonic: A, ([('III6', 'Minor', '1st', 'III', 'Tonic', 'A', 0)], [], [])
Tonic: C, ([], [], [])
Tonic: B, ([('II6', 'Minor', '1st', 'II', 'Subdominant', 'B', 0)], [], [])
Tonic: E, ([('VI6', 'Minor', '1st', 'VI', 'Tonic', 'E', 0)], [], [])
Tonic: D, ([], [], [])
Tonic: G, ([], [], [])
Tonic: F, ([], [], [])
Tonic: G#m, ([('IV6', 'Minor', '1st', 'IV', 'Subdominant', 'G#m', 0)], [], [])
Tonic: Ab, ([], [], [])
Tonic: Em, ([], [], [])
Tonic: D#m, ([], [], [])
Tonic: Cm, ([], [], [])
Tonic: Cb, ([], [], [])
Tonic: Am, ([], [], [])
Tonic: Eb, ([], [], [])
Tonic: F#, ([], [], [])
Tonic: Gb, ([], [], [])
Tonic: Ebm, ([], [], [])
Tonic: Gm, ([], [], [])
Tonic: F#m, ([('V6', 'Minor', '1st', 'V', 'Dominant', 'F#m', 0)], [], [])
```

Figure 4.5.1.4.1 Sample of Match Tuple

4.6. ChordInterval Class


ChordInterval
_noteList : Python List
_intervalType : ChordInterval.IntervalType
_measureNo : Integer
_intervalNo : Integer
_exactOffset : Float
_exactEndTime : Float
_recognizedResultDict : Python Dictionary
_equivalentGroupDict : Python Dictionary
analyzeEquivalentChord() : void
Enum: ChordInterval.IntervalType
OnBeat
AfterBeat
Continuous
ChangedBaseline
Normal

Figuire 4.6.1 ChordInterval UML

ChordInterval is an important class designed to modularize the divided music score within a specific time interval. When initializing a Identifier object with an input music score, music score is divided using Chordify() function. The program will then initialize numbers of ChordInterval object and construct the desired structure mentioned in section 4.4.1. Inside every ChordInterval object, numbers of attribute will be stored, including:

the measure number;
the interval number;
consisting note list, which is a list of ChordNote objects and will be discussed in section 4.7;
interval type, which will be discussed below;
offset and end time, which represents the time interval in the original music score;
and the results of the basic chord analysis algorithm, which are in the structure of Python Dictionary.

4.6.1. ChordInterval.IntervalType Enum

IntervalType is an designed Enum for grouping ChordInterval into different types. Due to the fact that python 2.7 does not have an Enum class, the IntervalType Enum is simulated by a class with a list of integer. A ChordInterval object can have multiple IntervalType after analyzing from the music score. Different interval types will be introduced below.

4.6.1.1. OnBeat

OnBeat interval type is assigned to a ChordInterval object when this ChordInterval is an onbeat interval.

4.6.1.2. AfterBeat

AfterBeat interval type is assigned when the ChordInterval is the first ChordInterval after an onbeat interval.

4.6.1.3. ChangedBaseline

ChangedBaseline interval type is assigned when the target ChordInterval has its base note, the note with the lowest frequency, changed comparing to the previous ChordInterval.

4.6.1.4. Continuous

Continuous interval type is assigned when the ChordInterval is made up of numbers of continuous single notes. When using the music21 Chordify() function, only the music notes with a same time interval would be grouped. When there is a time interval that only have one single note, the divided note will not be treated as a Chord due to the fact that a Chord should at least consists of two notes. As a result, all consecutive single notes will be grouped into a single interval and Continuous interval type is then introduced to represent it.

4.6.1.5. Normal

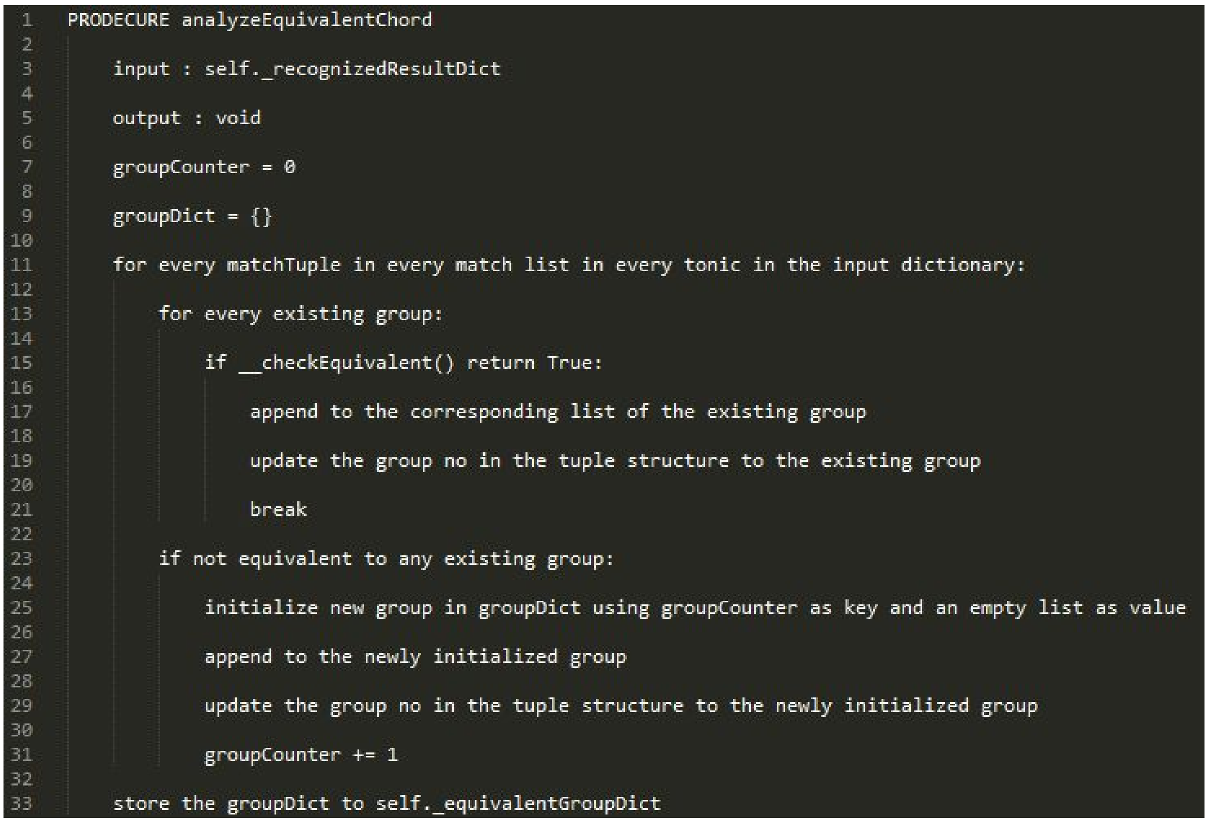
Finally, Normal interval type is assigned when the target ChordInterval is not recognized as the above interval types.

4.6.2. Result From Basic Chord Analysis Algorithm

There are two Python Dictionaries, recognizedResultDict and equivalentGroupDict, in each ChordInterval object responsible to store the basic chord analysis results. RecognizedResultDict is the returned dictionary from the function recognizeByAllTonic() from section 4.5.1; and for the equivalentGroupDict, it is from the function analyzeEquivalentChord(), which is a function inside ChordInterval Class and will be introduced below.

4.6.3. analyzeEquivalentChord()

Equivalent chords are chords that consists of same chord construction but in different tonic. Although these equivalent chords may have different chord function in their tonic, finding



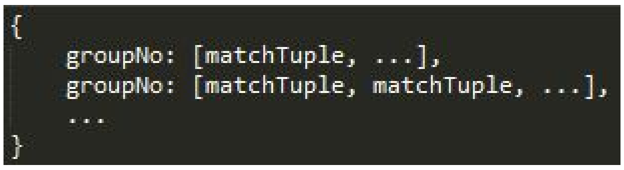
```
PRODECURE analyzeEquivalentChord

    input : self._recognizedResultDict

    output : void

    groupCounter = 0

    groupDict = {}

    for every matchTuple in every match list in every tonic in the input dictionary:

        for every existing group:

            if __checkEquivalent() return True:

                append to the corresponding list of the existing group

                update the group no in the tuple structure to the existing group

                break

        if not equivalent to any existing group:

            initialize new group in groupDict using groupCounter as key and an empty list as value

            append to the newly initialized group

            update the group no in the tuple structure to the newly initialized group

            groupCounter += 1

    store the groupDict to self._equivalentGroupDict
```

Figure 4.6.3.1 analyzeEquivalentChord Pseudocode

In line 3 and 33, as the analyzeEquivalentChord is an instance method, the instance variable _recognizedResultDict and _equivalentGroupDict can be accessed and altered inside the function.

In line 15, the __checkEquivalent() function is implemented to examine the input match tuples and find out if they are equivalent by doing calculations using the music dictionaries in the ChordAnalyzingTool.

In line 9 and 33, the dictionary structure will be as following,



```
{
    groupNo: [matchTuple, ...],
    groupNo: [matchTuple, matchTuple, ...],
    ...
}
```

4.7. ChordNote Class

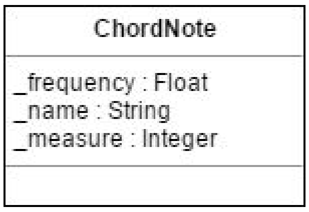


Figure 4.7.1 ChordNote UML

ChordNote is the most basic object in the ChordIdentifier module. Useful attributes are extracted from music21 note objects and stored inside the ChordNote object. The frequency attribute is useful when calculating the baseline change. However the class does not include any instance or class functions.

4.8. ProgressionBank Class

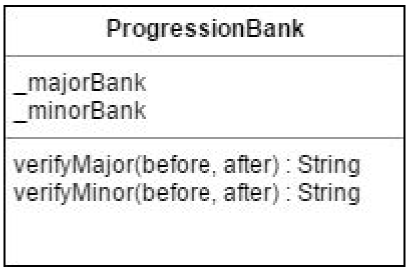

ProgressionBank
_majorBank
_minorBank
verifyMajor(before, after) : String
verifyMinor(before, after) : String

Figure 4.8.1 ProgressionBank UML

ProgressionBank Class modularize a bank object that stores two progression banks and provide function to verify whether the input chord name pair is a valid chord progression.

When initializing a ProgressionBank object, the program will load the progression table files from storage in order to get reading for verifying.

There are two progression tables used in the ChordIdentifier module, Major Progression Table and Minor Progression Table. The two tables are saved in CSV file format and are provided by the project supervisor, Professor Lucas Wong. The tables provide answers on whether a chord can be progressed to another chord in music sense.

4.9. ProgressionVerifier Class

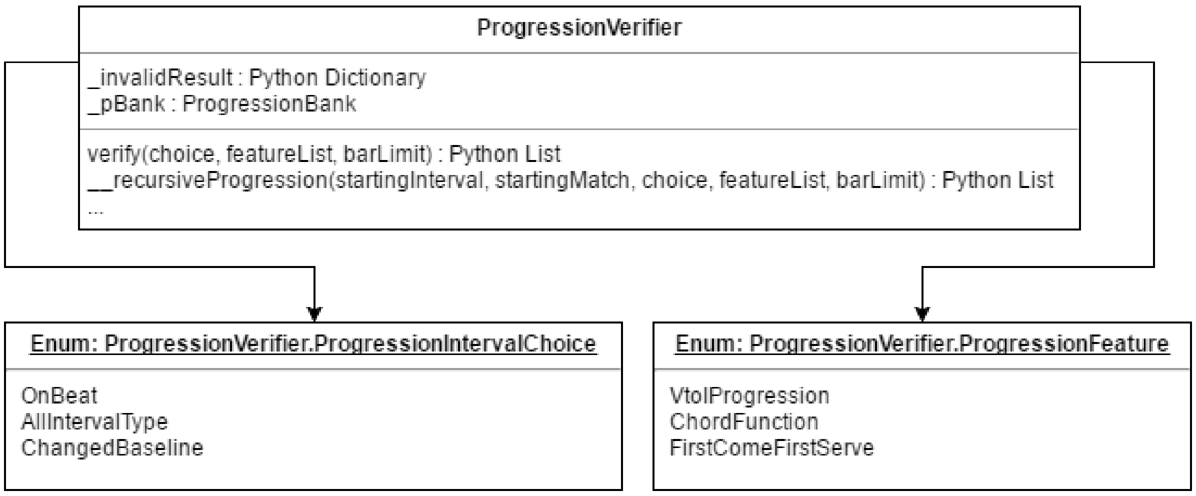

ProgressionVerifier
_invalidResult : Python Dictionary
_pBank : ProgressionBank
verify(choice, featureList, barLimit) : Python List
__recursiveProgression(startingInterval, startingMatch, choice, featureList, barLimit) : Python List
...
Enum: ProgressionVerifier.ProgressionIntervalChoice
OnBeat
AllIntervalType
ChangedBaseline
Enum: ProgressionVerifier.ProgressionFeature
VtoIProgression
ChordFunction
FirstComeFirstServe

Figure 4.9.1 ProgressionVerifier UML

The ProgressionVerifier Class is the most important Class in the ChordIdentifier module. Under the ProgressionVerifier Class, there are two simulated Enum named ProgressionIntervalChoice and ProgressionFeature. These are the optional modes and features that user can choose when running the progression algorithm. Besides, there is also a dictionary named _invalidResult, which is used to store the invalid results throughout the progression algorithm. Finally, the Class also provide the verify function, which is separated as verify(), the driving function; and __recursiveProgression(), the recursive function.

4.9.1. ProgressionVerifier.ProgressionIntervalChoice Enum

The ProgressionIntervalChoice Enum specifies the target ChordInterval type when running the progression algorithm. Users can specify their target interval type by inputting the corresponding ProgressionIntervalChoice when calling the function verify().

4.9.1.1. OnBeat

The OnBeat choice specifies that the progression algorithm should only examine progression on ChordInterval with OnBeat interval type.

4.9.1.2. AllIntervalType

AllIntervalType choice specifies that the progression algorithm should examine progression on ChordInterval with OnBeat, AfterBeat, ChangedBaseline and Normal interval type.

4.9.1.3. ChangedBaseline

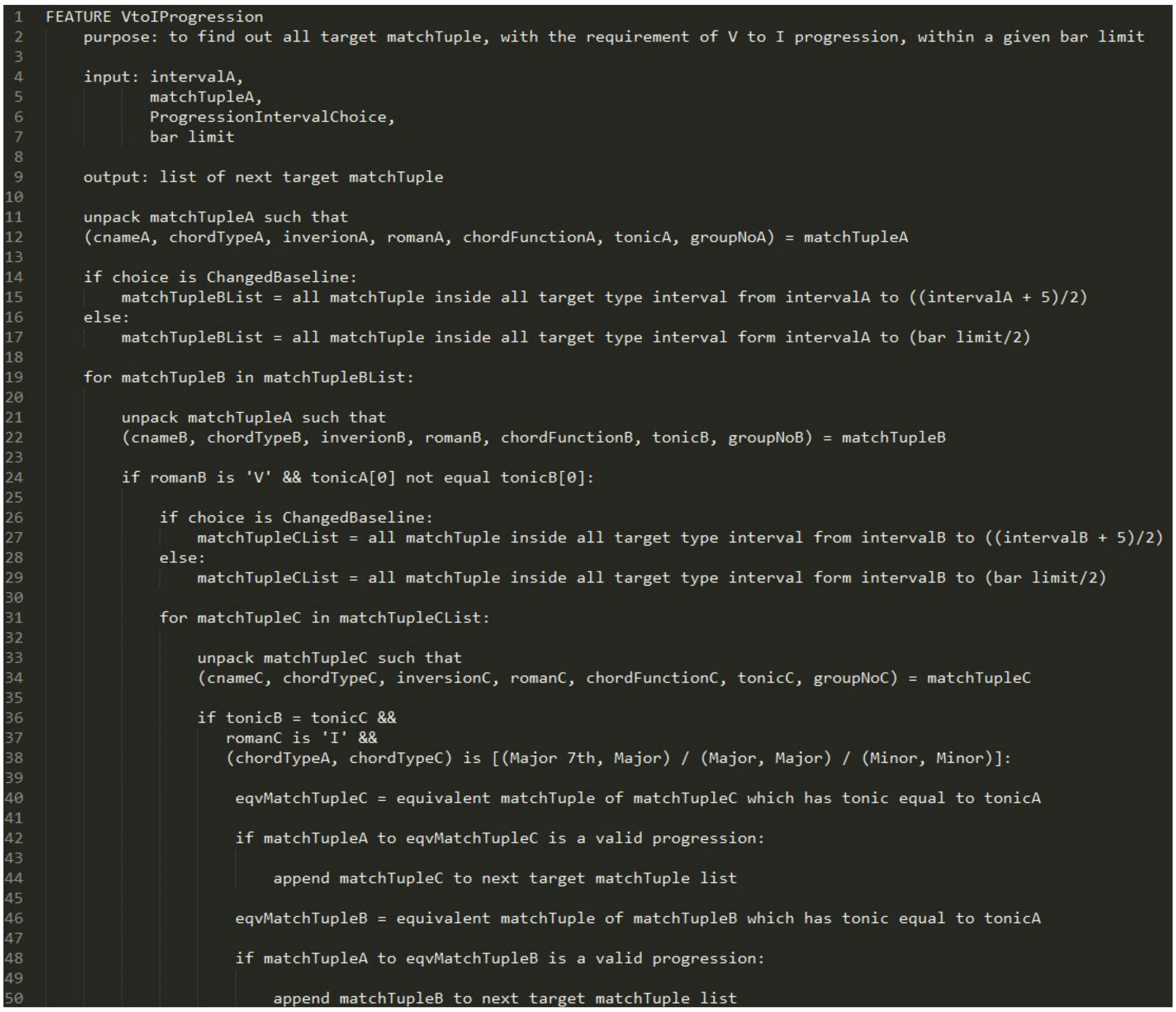
The ChangedBaseline choice specifies that the progression algorithm should only examine progression on ChordInterval with ChangedBaseline interval type.

4.9.2. ProgressionVerifier.ProgressionFeature Enum

The ProgressionFeature Enum specifies the features that the progression algorithm should use.

4.9.2.1. VtoIProgression

VtoIProgression has the highest priority in all three features when multiple features are selected. Moreover, VtoIProgression is the only feature that has the possibility to have progression with tonic change.



```
FEATURE VtoIProgression
    purpose: to find out all target matchTuple, with the requirement of V to I progression, within a given bar limit

    input: intervalA,
           matchTupleA,
           ProgressionIntervalChoice,
           bar limit

    output: list of next target matchTuple

    unpack matchTupleA such that
    (cnameA, chordTypeA, inverionA, romanA, chordFunctionA, tonicA, groupNoA) = matchTupleA

    if choice is ChangedBaseline:
        matchTupleBList = all matchTuple inside all target type interval from intervalA to ((intervalA + 5)/2)
    else:
        matchTupleBList = all matchTuple inside all target type interval form intervalA to (bar limit/2)

    for matchTupleB in matchTupleBList:

        unpack matchTupleA such that
        (cnameB, chordTypeB, inverionB, romanB, chordFunctionB, tonicB, groupNoB) = matchTupleB

        if romanB is 'V' && tonicA[0] not equal tonicB[0]:

            if choice is ChangedBaseline:
                matchTupleCList = all matchTuple inside all target type interval from intervalB to ((intervalB + 5)/2)
            else:
                matchTupleCList = all matchTuple inside all target type interval form intervalB to (bar limit/2)

            for matchTupleC in matchTupleCList:

                unpack matchTupleC such that
                (cnameC, chordTypeC, inversionC, romanC, chordFunctionC, tonicC, groupNoC) = matchTupleC

                if tonicB = tonicC &&
                   romanC is 'I' &&
                   (chordTypeA, chordTypeC) is [(Major 7th, Major) / (Major, Major) / (Minor, Minor)]:

                    eqvMatchTupleC = equivalent matchTuple of matchTupleC which has tonic equal to tonicA

                    if matchTupleA to eqvMatchTupleC is a valid progression:

                        append matchTupleC to next target matchTuple list

                    eqvMatchTupleB = equivalent matchTuple of matchTupleB which has tonic equal to tonicA

                    if matchTupleA to eqvMatchTupleB is a valid progression:

                        append matchTupleB to next target matchTuple list
```

Figure 4.9.1.2.1 VtoIProgression feature Pseudocode

In line 4 and 5, for each run of the recursive function,, there will be an input Interval and MatchTuple, named intervalA and matchTupleA, representing the latest MatchTuple in the current chord progression.

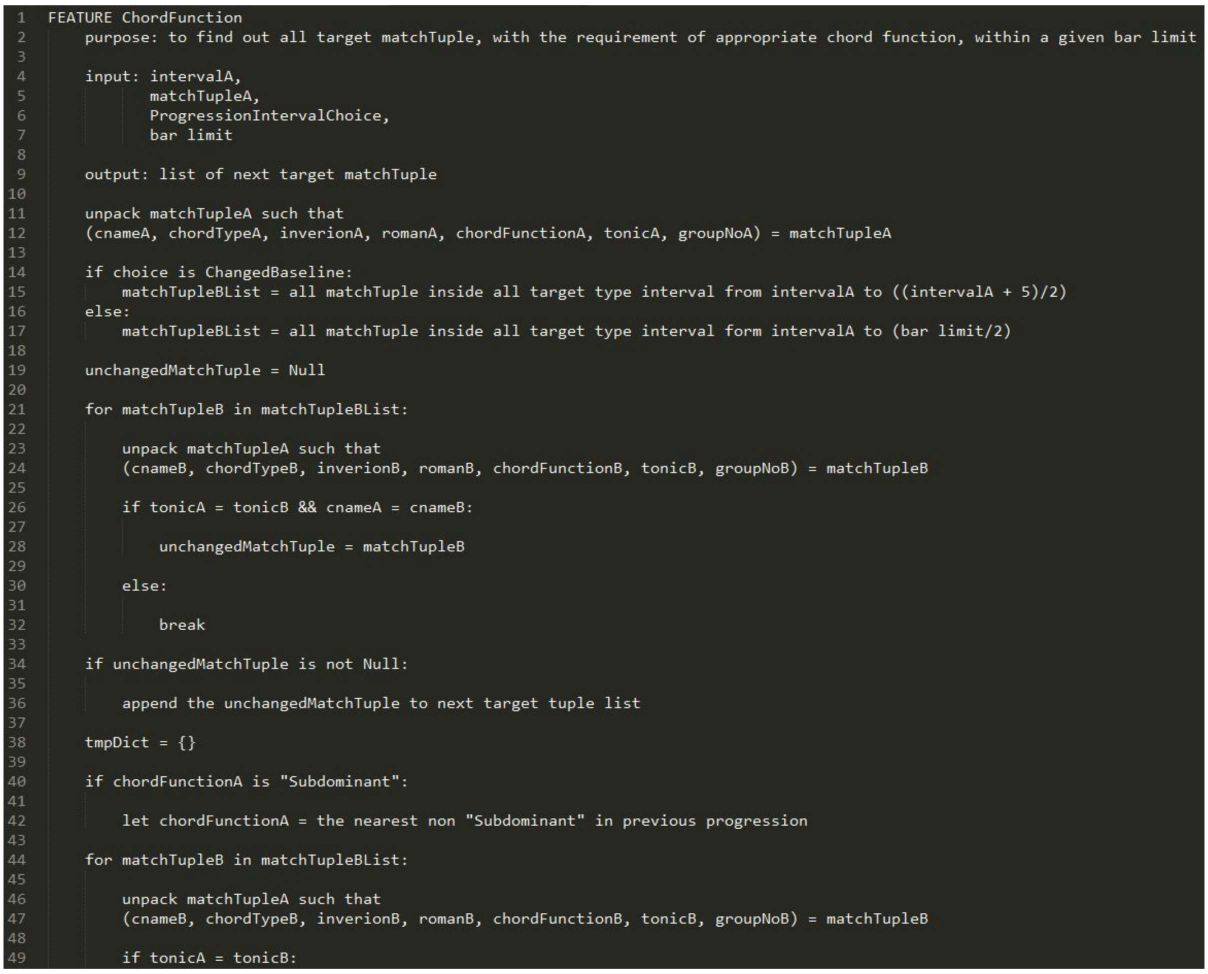
In line 24, the algorithm is trying to find a MatchTuple that have a different tonic with matchTupleA. Moreover, a tonic change of major and minor is also disallowed, for instance, if matchTupleA have tonic D, matchTupleB that have tonic Dm will not be selected even that they are in different tonic.

In line 40 to 44, using the eqvMatchTupleC that have a same tonic as matchTupleA ranked a higher priority, therefore the case will be handled first

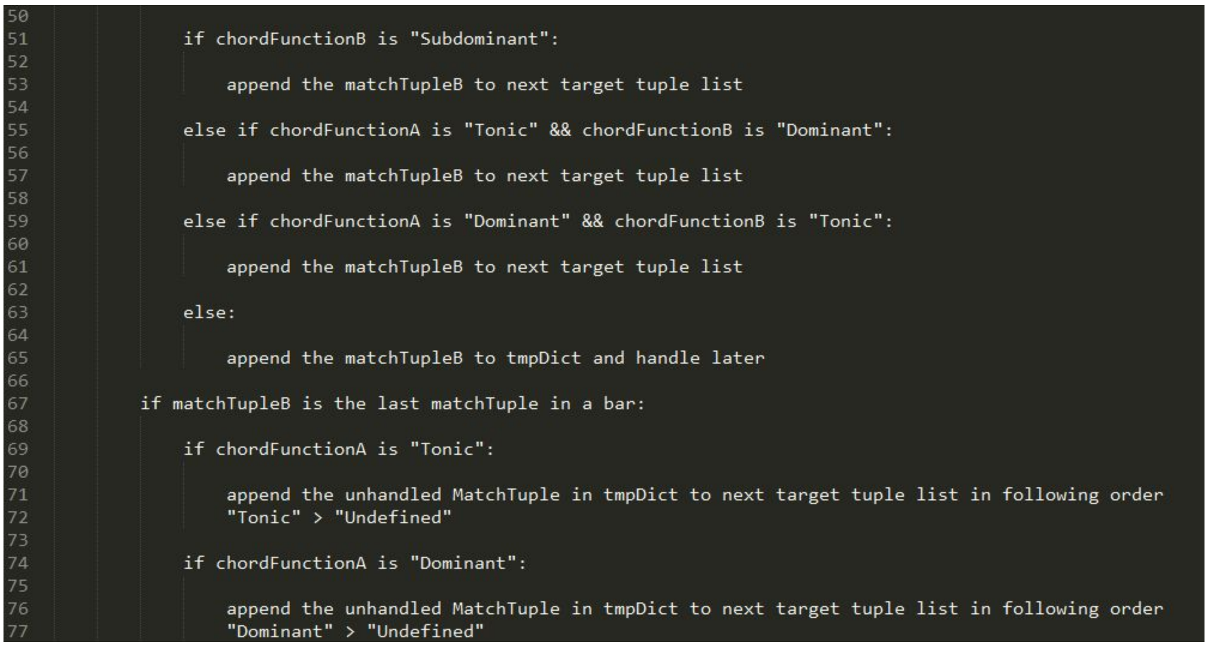
In line 46 to 50, although eqvMatchTupleB have a same tonic as matchTupleA, after matchTupleB progress to matchTupleC, the tonic will be changed and this is the only case that a change of tonic is allowed throughout the whole progression algorithm.

4.9.2.2. ChordFunction

ChordFunction is a feature that is mutually exclusive with the FirstComeFirstServe feature.



```
FEATURE ChordFunction
    purpose: to find out all target matchTuple, with the requirement of appropriate chord function, within a given bar limit

    input: intervalA,
           matchTupleA,
           ProgressionIntervalChoice,
           bar limit

    output: list of next target matchTuple

    unpack matchTupleA such that
    (cnameA, chordTypeA, inverionA, romanA, chordFunctionA, tonicA, groupNoA) = matchTupleA

    if choice is ChangedBaseline:
        matchTupleBList = all matchTuple inside all target type interval from intervalA to ((intervalA + 5)/2)
    else:
        matchTupleBList = all matchTuple inside all target type interval form intervalA to (bar limit/2)

    unchangedMatchTuple = Null

    for matchTupleB in matchTupleBList:

        unpack matchTupleA such that
        (cnameB, chordTypeB, inverionB, romanB, chordFunctionB, tonicB, groupNoB) = matchTupleB

        if tonicA = tonicB && cnameA = cnameB:

            unchangedMatchTuple = matchTupleB

        else:

            break

    if unchangedMatchTuple is not Null:

        append the unchangedMatchTuple to next target tuple list

    tmpDict = {}

    if chordFunctionA is "Subdominant":

        let chordFunctionA = the nearest non "Subdominant" in previous progression

    for matchTupleB in matchTupleBList:

        unpack matchTupleA such that
        (cnameB, chordTypeB, inverionB, romanB, chordFunctionB, tonicB, groupNoB) = matchTupleB

        if tonicA = tonicB:
```



```

            if chordFunctionB is "Subdominant":

                append the matchTupleB to next target tuple list

            else if chordFunctionA is "Tonic" && chordFunctionB is "Dominant":

                append the matchTupleB to next target tuple list

            else if chordFunctionA is "Dominant" && chordFunctionB is "Tonic":

                append the matchTupleB to next target tuple list

            else:

                append the matchTupleB to tmpDict and handle later

        if matchTupleB is the last matchTuple in a bar:

            if chordFunctionA is "Tonic":

                append the unhandled MatchTuple in tmpDict to next target tuple list in following order
                "Tonic" > "Undefined"

            if chordFunctionA is "Dominant":

                append the unhandled MatchTuple in tmpDict to next target tuple list in following order
                "Dominant" > "Undefined"
```

Figure 4.9.2.2.1 & 4.9.2.2.2 ChordFunction Pseudocode

In line 4 and 5, for each run of the recursive function,, there will be an input Interval and MatchTuple, named intervalA and matchTupleA, representing the latest MatchTuple in the current chord progression.

In line 19 to 36, the algorithm will first handle the consecutive progression path that the chord is remain unchanged with matchTupleA.

In line 51 to 65, the MatchTuple that match the requirements should have the priority according to their interval no, for the others, they will be handled at the end of each bar.

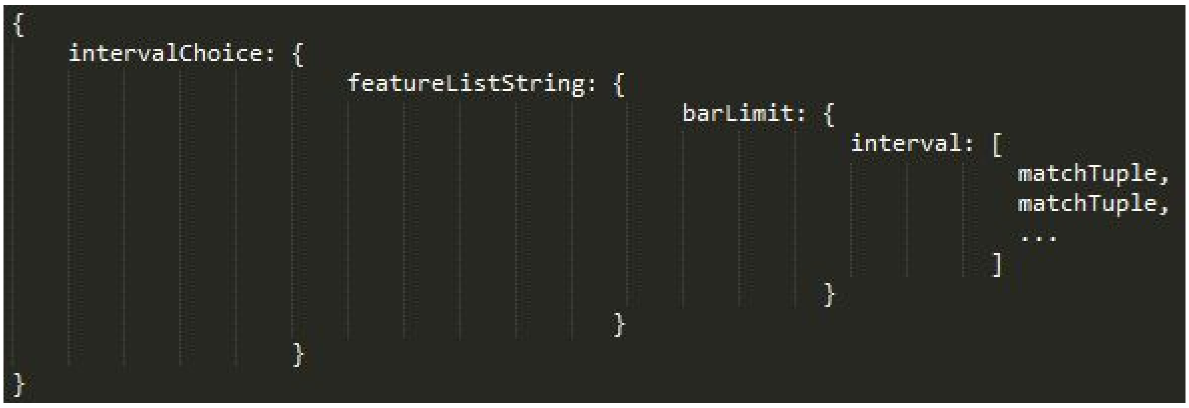
4.9.2.3. FirstComeFirstServe

FirstComeFirstServe is a feature that is mutually exclusive with the ChordFunction feature. FirstComeFirstServe chooses the nearest chord that present in the ProgressionBank.

4.9.3. Invalid Result Dictionary

Throughout the progression algorithm, it is possible that there is no any correct progression for that match tuple. In order to prevent selecting these match tuple repeatedly, the invalid result dictionary is implemented to store the invalid match tuple. Every time before recursively calling the progression function, the match tuple will be checked with the dictionary. If the match tuple appears inside the dictionary, the result is confirmed to be invalid.

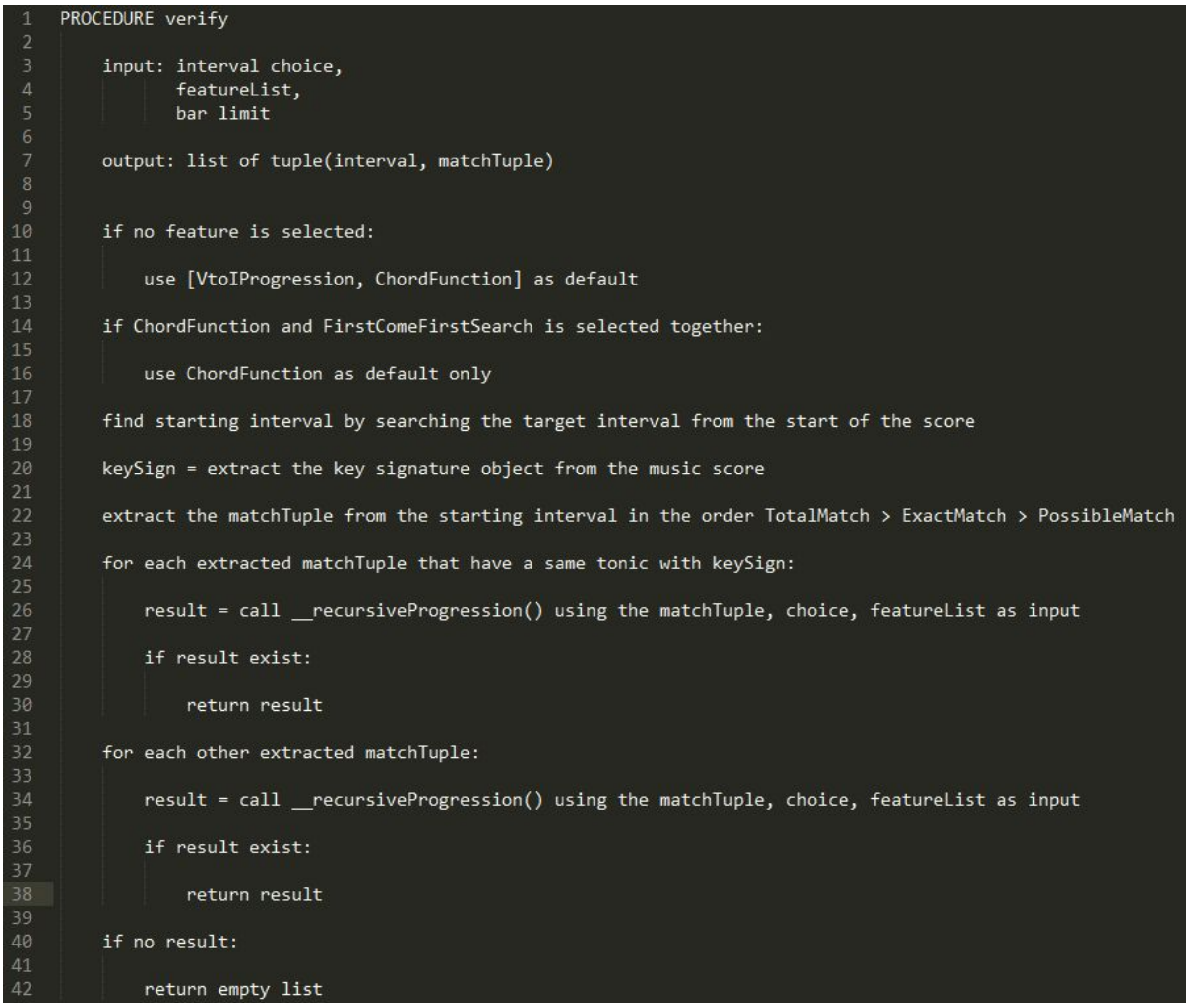
Due to the fact that different target interval type choice, progression feature combination and bar limit may result in different progression, the dictionary will have the following structure,



```
{
    intervalChoice: {
                        featureListString: {
                                                barLimit: {
                                                              interval: [
                                                                            matchTuple,
                                                                            matchTuple,
                                                                            ...
                                                                        ]
                                                          }
                                            }
                    }
}
```

The second key, featureListString, is the stringified featureList made for being a key in the dictionary. In the above structure, the invalid results in different modes and features can be stored separately.

4.9.4. Verify()



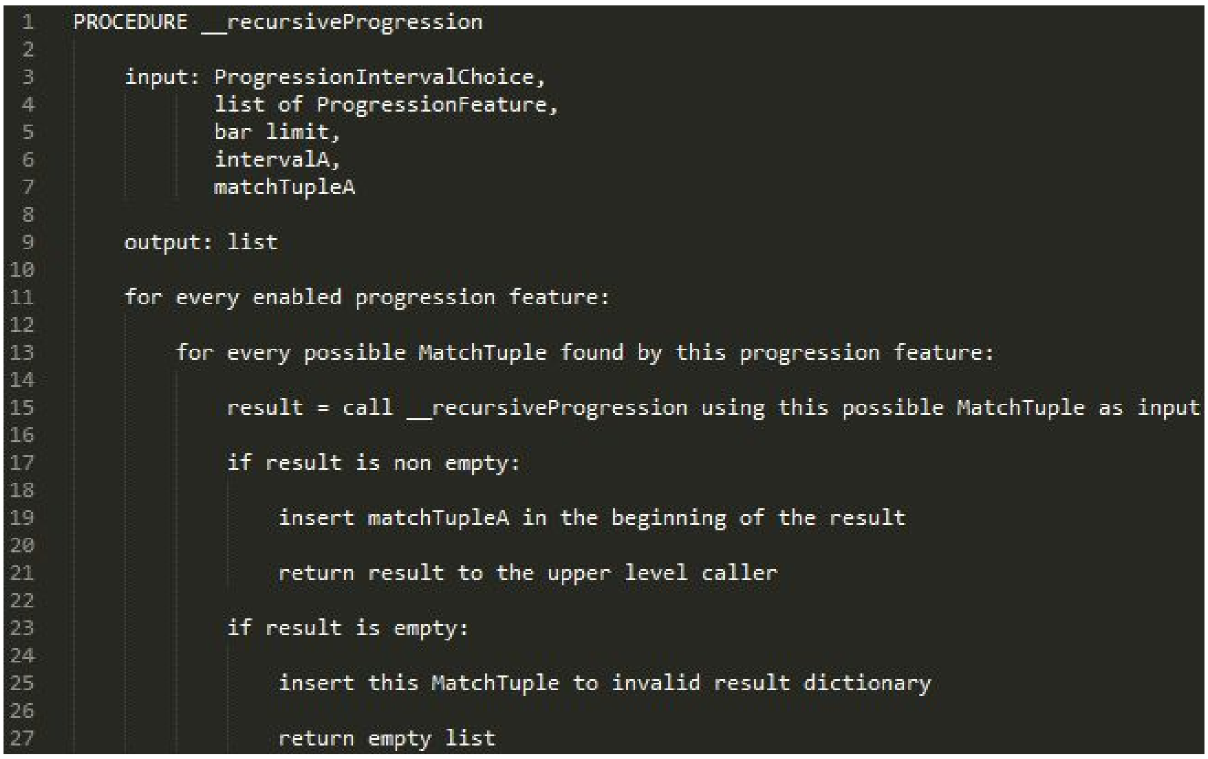
```
PROCEDURE verify

    input: interval choice,
           featureList,
           bar limit

    output: list of tuple(interval, matchTuple)


    if no feature is selected:

        use [VtoIProgression, ChordFunction] as default

    if ChordFunction and FirstComeFirstSearch is selected together:

        use ChordFunction as default only

    find starting interval by searching the target interval from the start of the score

    keySign = extract the key signature object from the music score

    extract the matchTuple from the starting interval in the order TotalMatch > ExactMatch > PossibleMatch

    for each extracted matchTuple that have a same tonic with keySign:

        result = call __recursiveProgression() using the matchTuple, choice, featureList as input

        if result exist:

            return result

    for each other extracted matchTuple:

        result = call __recursiveProgression() using the matchTuple, choice, featureList as input

        if result exist:

            return result

    if no result:

        return empty list
```

Figure 4.9.4.1 Verify Pseudocode

Verify() is the driving function of the progression algorithm. Firstly, the input feature list will be handled to make sure the combination is correct. For instance, ChordFunction and FirstComeFirstServe features should be mutually exclusive, therefore, only one of them can be selected. Besides, the starting point of the chord progression will be determined in this function and the recursive progression function will be started from this function.

4.9.5. __recursiveProgression()

The recursive progression function is a function that try to find a correct chord progression path using the concept of Depth-Limited Search with bar limit as the limit.



```
1   PROCEDURE __recursiveProgression
2
3       input: ProgressionIntervalChoice,
4              list of ProgressionFeature,
5              bar limit,
6              intervalA,
7              matchTupleA
8
9       output: list
10
11      for every enabled progression feature:
12
13          for every possible MatchTuple found by this progression feature:
14
15              result = call __recursiveProgression using this possible MatchTuple as input
16
17              if result is non empty:
18
19                  insert matchTupleA in the beginning of the result
20
21                  return result to the upper level caller
22
23              if result is empty:
24
25                  insert this MatchTuple to invalid result dictionary
26
27                  return empty list
```

Figure 4.9.5.1 __recursiveProgression Pseudocode
In line 13, the progression features are going through in the order of,

```
VtoIProgression > [ChordFunction or FirstComeFirstServe]
```
.

In line 6 and 7, for each run of the recursive function,, there will be an input Interval and MatchTuple, named intervalA and matchTupleA, representing the latest MatchTuple in the current chord progression.

In line 13, the program will go through the ProgressionFeatures in the above order. Inside the algorithm of the features, if there is a MatchTuple that gives a valid progression with “matchTupleA”, it will recursively call the function itself using the new valid MatchTuple as an input.

In line 17 to 27, if the recursive function returns a non empty list, it means that a correct chord progression path has been found. The function will then insert the matchTupleA in the beginning of the list and return the list to its upper level caller. However, if the result from the deeper level of the recursive function is an empty list, it means that there is no valid progression using this MatchTuple at this point. The program will append this MatchTuple to the invalid result dictionary mentioned in section 4.9.3, and proceed on the other possible MatchTuple.

5. Experiment and Result

The ChordIdentifier module provides numbers of option to run a chord progression algorithm. In order to examine the accuracy of the progression features algorithm, tests was carried out. In this section, results of the tests will be discussed.

5.1. Experiment Setting

The tests were carried out by firstly generate numbers of result using different combinations of ProgressionIntervalChoice and ProgressionFeatures, then compare the result of the generated score with the professional musical analysis of the target scores on the Internet. The target scores and the combinations will be discussed below.

5.1.1. Testing Combinations

There are totally 12 combinations of different features.

For every ProgressionIntervalChoice, there will be 4 possible progression features as following,

[ChordFunction]; [VtoIProgression, ChordFunction];

[FirstComeFirstServe]; [VtoIProgression, FirstComeFirstServe];

5.1.1.1. Interval Type Features Explained

The algorithm allows user to choose between the three interval types introduced in 4.9.1. for running the progression algorithm. These choices are based on the following philosophy:

OnBeat:	As musicians, especially those in classic period, put emphasis on beats, including changing chords on beats

AllInterval: As introduced in 2.4.4.5 and 2.4.4.6, non-harmonic notes may be accented, leading to the failure of analysis purely relied on OnBeat. AllInterval analysis can be an aid for the situation

ChangedBaseLine: As baseline progression commonly build the sense of tonality in classic music, checking by ChangedBaseLine may not be promising but worth trying algorithm.

5.1.1.2. Progression Features Explained

As mentioned in 5.1.1., total of 4 modes of progression features are allowed to be chosen which are designed as the following reasons:

VtoIProgession: As V7-I, V-I (in major key) and V+7-I+, V+-I+, V-I (in minor key) is commonly used for transition to the I chord. This algorithm also act as the key algorithm that allows tonality (key) change

ChordFunction: Based on the discussion of section 2.5.1., the chord function is highly relied in this algorithm

FirstComeFirstServe: A naive mean to check the progression. Yet could be very powerful when the chord arrangement is clear

5.1.2. Experimental Pieces

In the experiment, two pieces will be under test -- Canon in D and Air on G String (Excerpts from bar 1 to bar 6). The pieces with general chord analysis can be found at Appendix C and D. Some characteristics of the pieces would be discussed in the following sections.

5.1.2.1. Canon in D

Canon in D is a string quartet written by Pachelbel in around 1680. Pachelbel used a very simple structure by constructing a looping chord progression (I-V-VI-III-IV-I-IV-V) and assign a bass to loop on the eight bass note, representing the chord changes. Thus, Canon in D should be an easy piece to test the accuracy of the algorithm.

5.1.2.2. Air on G String

Air on G String was composed by Bach. Difference from Canon in D, Air on G String is rather hard as it changes tonality 4 times in 6 bars time: at first D major, then change to Em in the 3rd beat of bar 2 (note that the V chord of the 2nd beat of A major act as a V-I transition to the fifth chord of D major), back to D major at the beginning of bar 3, further to the dominant key of D major -- A major after a bar and return back to its origin tonic, D major at the 3rd beat of bar 6. Moreover, the chord progression is much more complicated that the 8 chord progression of Canon in D; there even a chord changes happens on offbeat (bar 6 the 2nd eighth note). Therefore, it is a suitable test to try the limit of the program.

5.2. Experiment Result and Analyze

5.2.1. Canon in D

Bar 3 to 6

IntervalChoice	ProgressionFeature	Total Correct	Partial Correct
AllIntervalType	ChordFunction	14/16(87.5%)	0/16 (0%)
AllIntervalType	FirstComeFirstServe	16/16 (100%)	0/16 (0%)
AllIntervalType	VtoIProgression, ChordFunction	14/16 (87.5%)	1/16 (6.3%)
AllIntervalType	VtoIProgression, FirstComeFirstServe	15/16 (93.8%)	1/16 (6.3%)
ChangedBaseline	ChordFunction	12/16 (75%)	0/16 (0%)
ChangedBaseline	FirstComeFirstServe	16/16 (100%)	0/16 (0%)
ChangedBaseline	VtoIProgression, ChordFunction	9/16 (56.3%)	1/16 (6.3%)
ChangedBaseline	VtoIProgression, FirstComeFirstServe	11/16 (68.8 %)	1/16 (6.3%)
OnBeat	ChordFunction	14/16 (87.5%)	0/16 (0%)
OnBeat	FirstComeFirstServe	16/16 (100%)	0/16 (0%)
OnBeat	VtoIProgression, ChordFunction	14/16 (87.5%)	1/16 (6.3%)
OnBeat	VtoIProgression, FirstComeFirstServe	15/16 (93.8%)	1/16 (6.3%)

The combination of {ChangeBaseLine, FirstComeFirstServe}

Bar 19 to 22

IntervalChoice	ProgressionFeature	Total Correct	Partial Correct
AllIntervalType	ChordFunction	12/16 (75%)	0/16 (0%)
AllIntervalType	FirstComeFirstServe	13/16 (81.3%)	0/16 (0%)
AllIntervalType	VtoIProgression, ChordFunction	0/16 (0%)	6/16 (37.5%)
AllIntervalType	VtoIProgression, FirstComeFirstServe	0/16 (0%)	8/16 (50%)
ChangedBaseline	ChordFunction	12/16 (75%)	0/16 (0%)
ChangedBaseline	FirstComeFirstServe	14/16 (87.5%)	0/16 (0%)
ChangedBaseline	VtoIProgression, ChordFunction	4/16 (25%)	4/16 (25%)
ChangedBaseline	VtoIProgression, FirstComeFirstServe	4/16 (25%)	4/16 (25%)
OnBeat	ChordFunction	12/16 (75%)	0/16 (0%)
OnBeat	FirstComeFirstServe	14/16 (87.5%)	0/16 (0%)
OnBeat	VtoIProgression, ChordFunction	12/16 (75%)	2/16 (12.5%)
OnBeat	VtoIProgression, FirstComeFirstServe	12/16 (75%)	2/16 (12.5%)

5.2.2. Air on the G String (excerpts)

IntervalChoice	ProgressionFeature	Total Correct	Partial Correct
AllIntervalType	ChordFunction	7/19 (36.8%)	4/19 (21.1%)
AllIntervalType	FirstComeFirstServe	8/19 (42.1%)	6/19 (31.6%)
AllIntervalType	VtoIProgression, ChordFunction	4/19 (21.1%)	8/19 (42.1%)
AllIntervalType	VtoIProgression, FirstComeFirstServe	5/19 (26.3%)	8/19 (42.1%)
ChangedBaseline	ChordFunction	no result	no result
ChangedBaseline	FirstComeFirstServe	no result	no result
ChangedBaseline	VtoIProgression, ChordFunction	no result	no result
ChangedBaseline	VtoIProgression, FirstComeFirstServe	no result	no result
OnBeat	ChordFunction	6/19 (31.6%)	2/19 (10.5%)
OnBeat	FirstComeFirstServe	8/19 (42.1%)	4/19 (21.1%)
OnBeat	VtoIProgression, ChordFunction	6/19 (31.6 %)	4/19 (21.1%)
OnBeat	VtoIProgression, FirstComeFirstServe	9/19 (47.4%)	2/19 (10.5%)

5.2.3. Time Cost

The time cost of the above tests have also been recorded. For running all chord progression algorithm for 12 different combinations, the time spent are as following,

Score	Number of Bars	Time used
Air on the G String (excerpts)	6	771ms
Canon in D	57	905ms

6. Discussion

6.1. Modules

It can be seen clearly that the combination of {ChangedBaseLine, VtoIProgression} fails most of the cases. As ChangedBaseLine is specially designed for music that changed chords with the progression of baseline, it is clear that it would work better with pieces like Canon in D.

For Air on G String, it can be seen that the VtoIProgression feature cannot successfully to act as tonic change detection. The reason behind is that the algorithm does not general all the rules due to the time complexity to construct it.

The failure in the algorithm is mainly due to the time complexity. As the algorithm is relied on DFS at present, which aim to cut time in return of the accuracy. If the algorithm walk through every possibility, there is no doubt that it could find out the best solution. Yet, due to the limitation of greedy algorithm, best solution may not be general and thus led to the loss in accuracy.

6.2. Limitation of the Current Algorithm

6.2.1. Chord Recognition Interval

The ChordInterval object is divided by the music21 function, Chordify(). The function will group simultaneous music notes together to a chord object. The ChordIdentifier module has used the same method in dividing music notes into group as a ChordInterval and run analysis with the component notes inside the ChordInterval. However, in musical sense, the "Interval" concept should not be used in analyzing chord due to the fact that musicians may split up the key component notes of a chord inside a bar. When human musicians running an chord analysis by looking at the score, it is much easier for them to recognize the formation of the chord. On the other hand, the division method of the ChordIdentifier module may have unsatisfying result on these types of music score.

6.2.2. Best Chord Progression Path

The chord progression algorithm of the ChordIdentifier module adopts the greedy algorithm, which dividing the problem into small stages, which is ChordInterval, and find the best matchTuple at each ChordInterval as progression chord. Unfortunately, chord analyzing does not appear to be having good results using the above algorithm as a matter of fact that there is no absolute correct answer for chord analysis even for human.

6.2.3. ProgressionVerifier Features

The ChordIdentifier module have implemented with numbers of chord progression features. However, these features were designed based on some musical theories that might not be applied to all music scores. The ChordIdentifier module is not capable of recognizing the best progression features for an input score, and even though it can generates numbers of result by running with all possible combinations, there is still no accurate rules or indicators to tells the module which is the best chord progression path.

6.3 Further Development

6.3.1. ChordInterval

In the future, it is hope that the ChordInterval sampling range can be improved and generalized with more musical theories. ChordInterval is the fundamental object throughout the whole algorithm. However, the accuracy of dividing it is still needed to be improved.

6.3.2. ProgressionFeature

ProgressionFeature is another important part of the ChordIdentifier module. It is hoped that the ProgressionFeatures can be extended to consider more case on tonic change, chord progression accuracy, etc.

6.3.3. Possibility on Machine Learning

Throughout the project, it is found that it is really difficult to generalize rules for analyzing music chords. For the similar type of problems in Computer Science, there is a newly developed approach to search for answers, machine learning. Originally, the chord identification module is introduced in order to improve accuracy on automatic piano reduction. As a result, the chord identification process is imagined to be small and light weighted in order to be plugged into the main reduction program. However, after some trials and studies on the topic, it is suggested that machine learning algorithm could also be a possible method to improve the accuracy of chord identification.

7. Conclusion

Automatic piano reduction is an ongoing project which aims to undergo automatic piano reduction with the aid of machine learning. Although basic reduction has been attained by Cherry, harmonic analysis is still a shortcoming of the reduction algorithm. Despite of many effort invested this year, a suitable algorithm is still yet to be designed. It is true that music rules can be modularize by mathematics, throughout the development on this project, however, it is found that running a chord analysis on a computer program is difficult. Due to the fact the it is hard to generalize perfect rules on chord identifying for all possible music scores, the ChordIdentifier module is still required with amounting time for development and design. It is hoped that the infrastructure of the ChordIdentifier module can be improved and a more promising ProgressionFeatures can be developed in the future.

8. Work Distribution

This project requires a lot of musical theories to support the algorithm design of the ChordIdentifier module.

Our group member, Cheuk Bun Wong, is mainly responsible in musical theory based algorithm design. He has been a violinist and member of a choir. He shows his strength in musical theories and knowledge, hence generalizes and transforms musical rules into computer algorithm. For example, he has designed the basic chord analysis algorithm and the progression features algorithm using his knowledge and musical theories.

Our other group member, Wai Man Yip, mainly responsible in programming. He has designed and implemented the ChordIdentifier module structure and various class objects to provide a framework for the analysis algorithm to run. Besides, he has also designed the recursive program flow of the ProgressionVerifier and implemented the progression features algorithm provided by Wong.

9. Reflections and Learnings

10. Reference

The following open-source libraries are used in this project:

Music21 - http://web.mit.edu/music21/

Pybrain - http://pybrain.org/

The following websites are referenced in this project report:

https://www.w3.org/TR/xml/

https://www.musicxml.com/

http://web.mit.edu/music21/

http://pybrain.org/

The following books are referenced in the music theory part:

1. O. Károlyi, Introducing Music, New Edition Edition ed. Penguin Books Ltd, 1991, p. p. 63.
2. Joutsenvirta, Aarre, and Jari Perkiömäki. "Music Theory Non-Harmonic Notes". Www2.siba.fi. N.p., 2017. Web. 12 Apr. 2017.
3. "Pedal Point". En.wikipedia.org. N.p., 2017. Tue. 11 Apr. 2017.
4. "Chord Progression". En.wikipedia.org. N.p., 2017. Web. 18 Apr. 2017.

Appendixies A

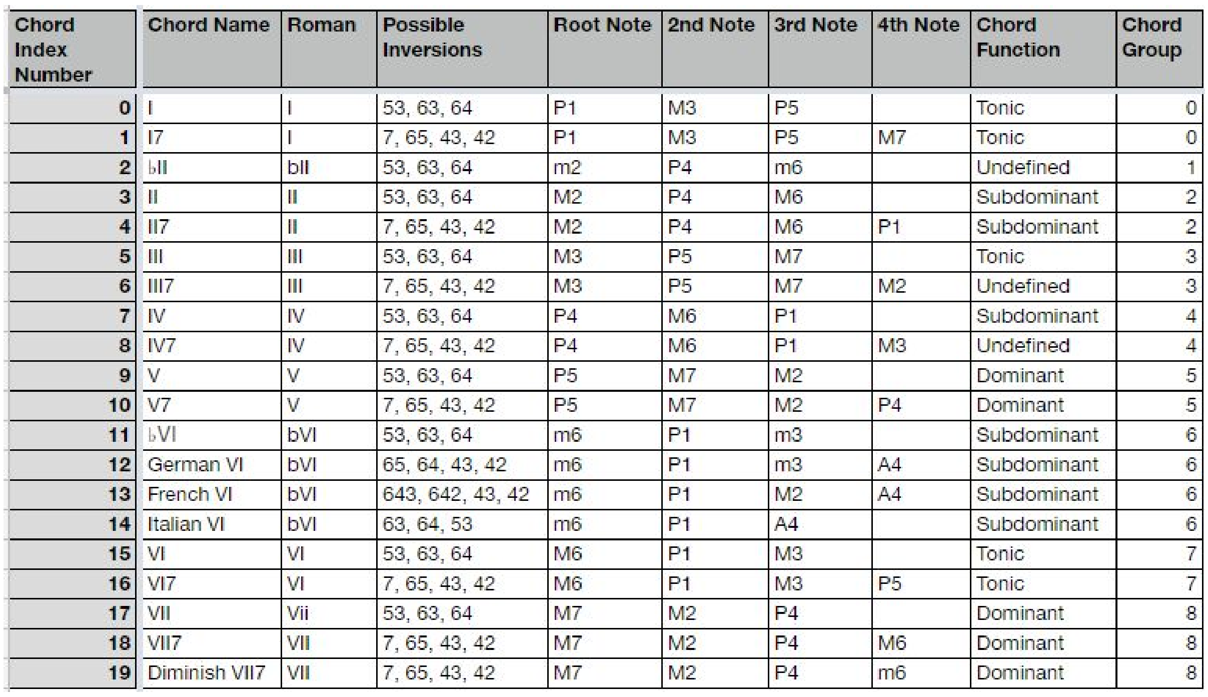
Common Interval used in the project

Interval Name	Short Hand	Semitone Difference*	Note Difference*
Union	P1	0	0
Minor 2nd	m2	1	1
Major 2nd	M2	2	1
Minor 3rd	m3	3	2
Major 3rd	M3	4	2
Perfect 4th	P4	5	3
Augmented 4th	A4	6	3
Perfect 5th	P5	7	4
Minor 6th	m6	8	5
Major 6th	M6	9	5
Minor 7th	m7	10	6
Major 7th	M7	11	6

* with reference to the tonic note

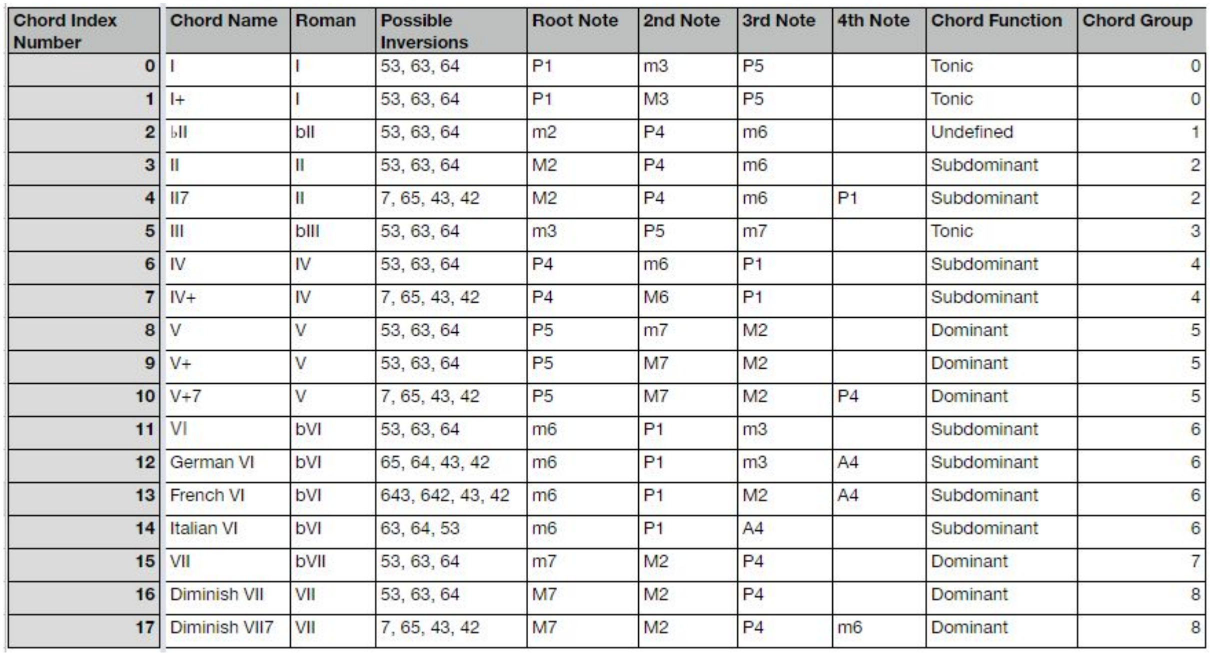
Appendixies B

Common Major Chord handled in the project



Chord Index Number	Chord Name	Roman	Possible Inversions	Root Note	2nd Note	3rd Note	4th Note	Chord Function	Chord Group
0	I	I	53, 63, 64	P1	M3	P5		Tonic	0
1	I7	I	7, 65, 43, 42	P1	M3	P5	M7	Tonic	0
2	♭II	bII	53, 63, 64	m2	P4	m6		Undefined	1
3	II	II	53, 63, 64	M2	P4	M6		Subdominant	2
4	II7	II	7, 65, 43, 42	M2	P4	M6	P1	Subdominant	2
5	III	III	53, 63, 64	M3	P5	M7		Tonic	3
6	III7	III	7, 65, 43, 42	M3	P5	M7	M2	Undefined	3
7	IV	IV	53, 63, 64	P4	M6	P1		Subdominant	4
8	IV7	IV	7, 65, 43, 42	P4	M6	P1	M3	Undefined	4
9	V	V	53, 63, 64	P5	M7	M2		Dominant	5
10	V7	V	7, 65, 43, 42	P5	M7	M2	P4	Dominant	5
11	♭VI	bVI	53, 63, 64	m6	P1	m3		Subdominant	6
12	German VI	bVI	65, 64, 43, 42	m6	P1	m3	A4	Subdominant	6
13	French VI	bVI	643, 642, 43, 42	m6	P1	M2	A4	Subdominant	6
14	Italian VI	bVI	63, 64, 53	m6	P1	A4		Subdominant	6
15	VI	VI	53, 63, 64	M6	P1	M3		Tonic	7
16	VI7	VI	7, 65, 43, 42	M6	P1	M3	P5	Tonic	7
17	VII	Vii	53, 63, 64	M7	M2	P4		Dominant	8
18	VII7	VII	7, 65, 43, 42	M7	M2	P4	M6	Dominant	8
19	Diminish VII7	VII	7, 65, 43, 42	M7	M2	P4	m6	Dominant	8

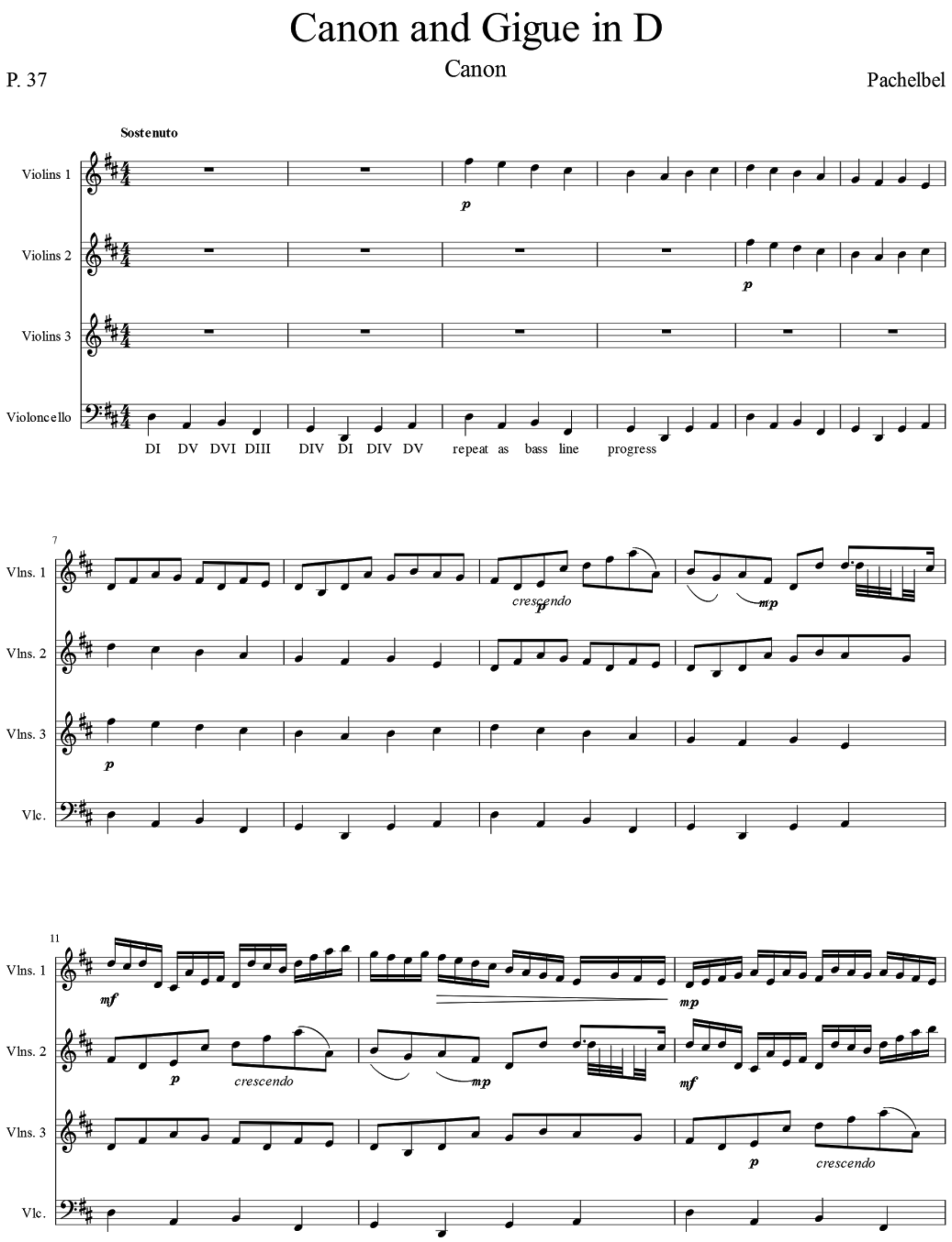
Common Minor Chord handled in the project

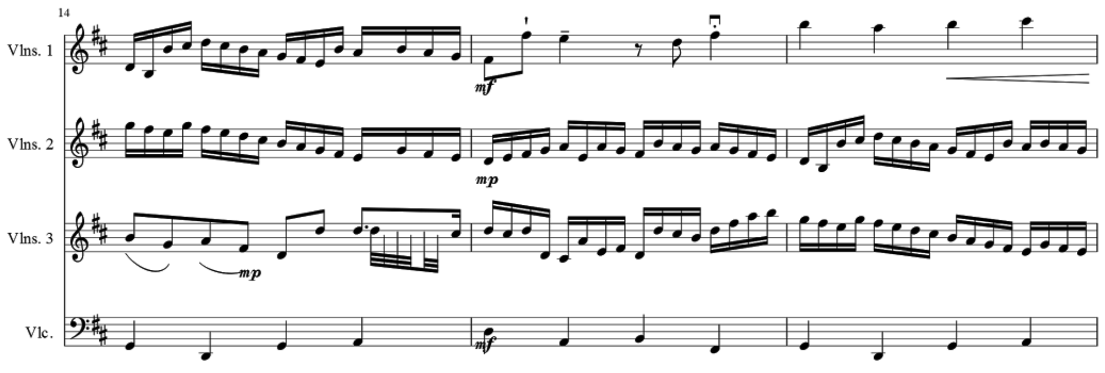


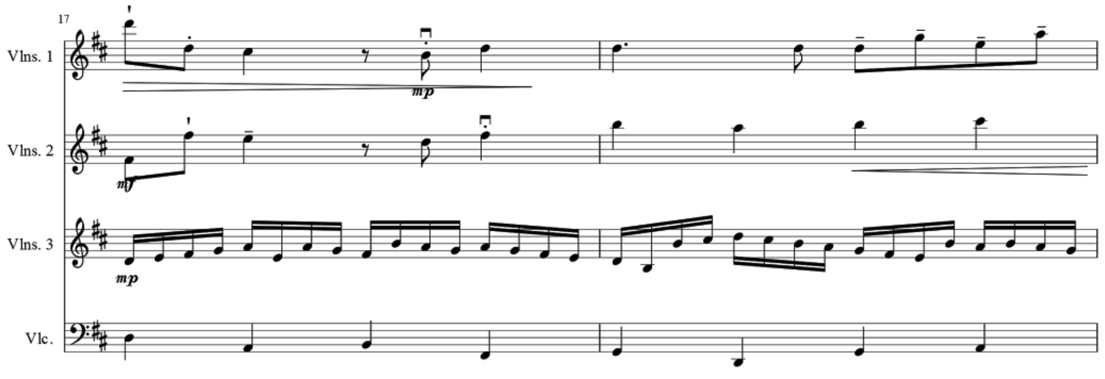
Chord Index Number	Chord Name	Roman	Possible Inversions	Root Note	2nd Note	3rd Note	4th Note	Chord Function	Chord Group
0	I	I	53, 63, 64	P1	m3	P5		Tonic	0
1	I+	I	53, 63, 64	P1	M3	P5		Tonic	0
2	♭II	bII	53, 63, 64	m2	P4	m6		Undefined	1
3	II	II	53, 63, 64	M2	P4	m6		Subdominant	2
4	II7	II	7, 65, 43, 42	M2	P4	m6	P1	Subdominant	2
5	III	bIII	53, 63, 64	m3	P5	m7		Tonic	3
6	IV	IV	53, 63, 64	P4	m6	P1		Subdominant	4
7	IV+	IV	7, 65, 43, 42	P4	M6	P1		Subdominant	4
8	V	V	53, 63, 64	P5	m7	M2		Dominant	5
9	V+	V	53, 63, 64	P5	M7	M2		Dominant	5
10	V+7	V	7, 65, 43, 42	P5	M7	M2	P4	Dominant	5
11	VI	bVI	53, 63, 64	m6	P1	m3		Subdominant	6
12	German VI	bVI	65, 64, 43, 42	m6	P1	m3	A4	Subdominant	6
13	French VI	bVI	643, 642, 43, 42	m6	P1	M2	A4	Subdominant	6
14	Italian VI	bVI	63, 64, 53	m6	P1	A4		Subdominant	6
15	VII	bVII	53, 63, 64	m7	M2	P4		Dominant	7
16	Diminish VII	VII	53, 63, 64	M7	M2	P4		Dominant	8
17	Diminish VII7	VII	7, 65, 43, 42	M7	M2	P4	m6	Dominant	8

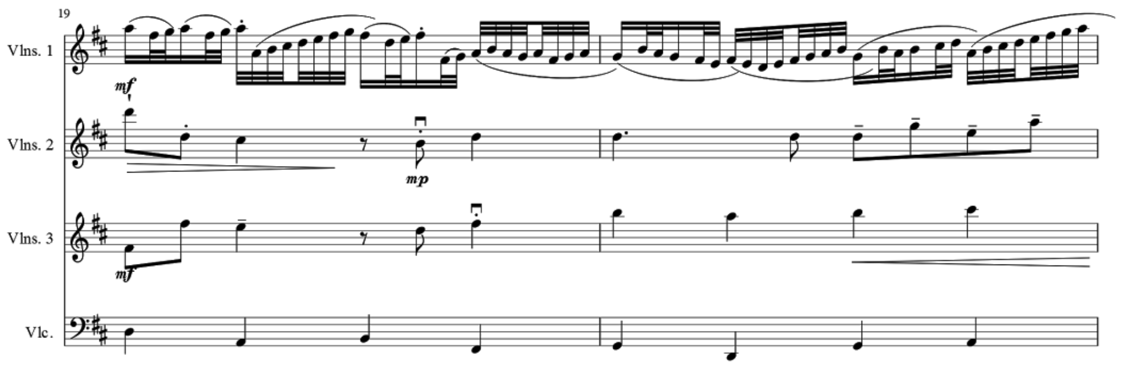
Appendixies C

Canon in D Full Score

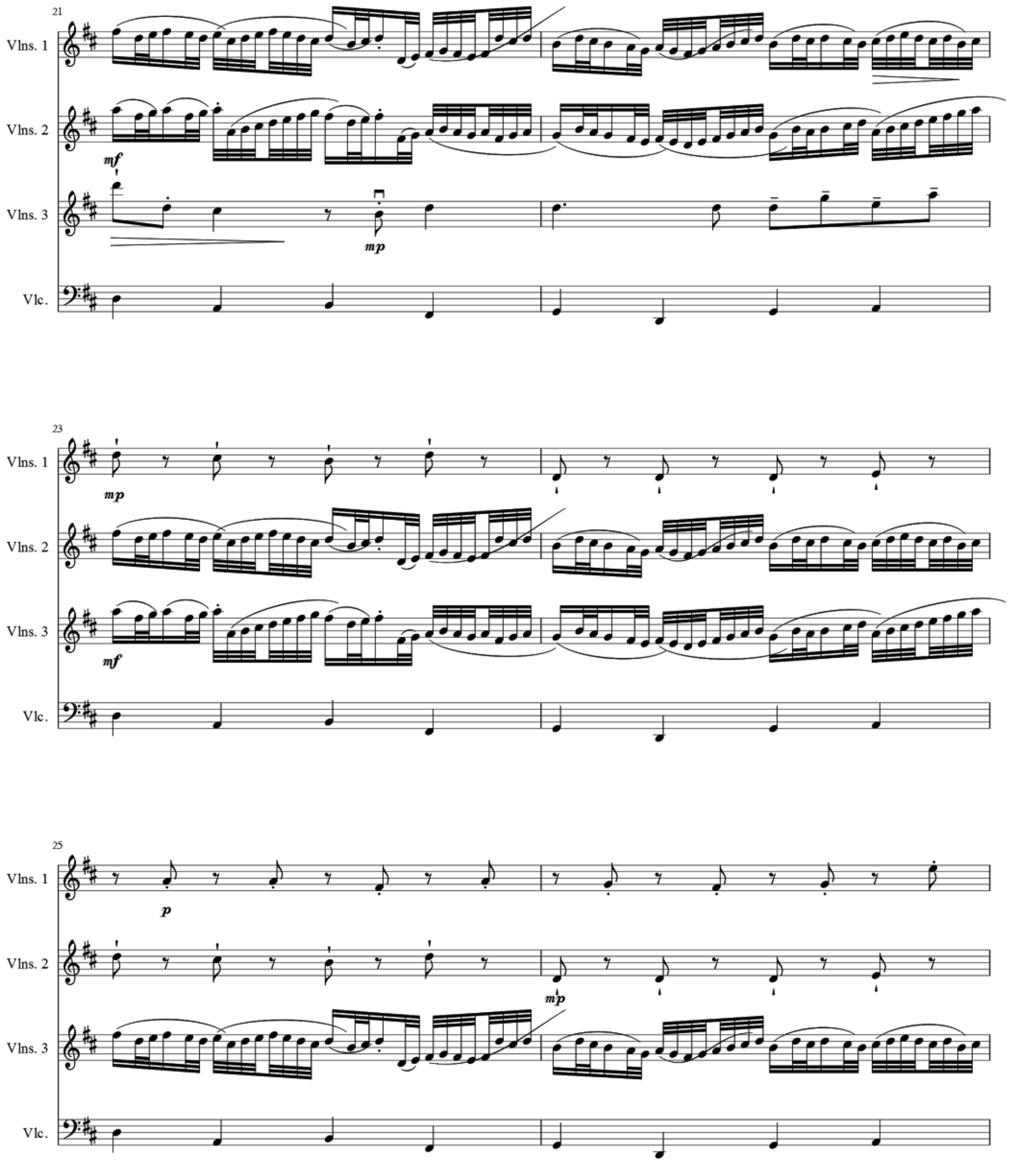



14
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
mf
mp
mp
mf

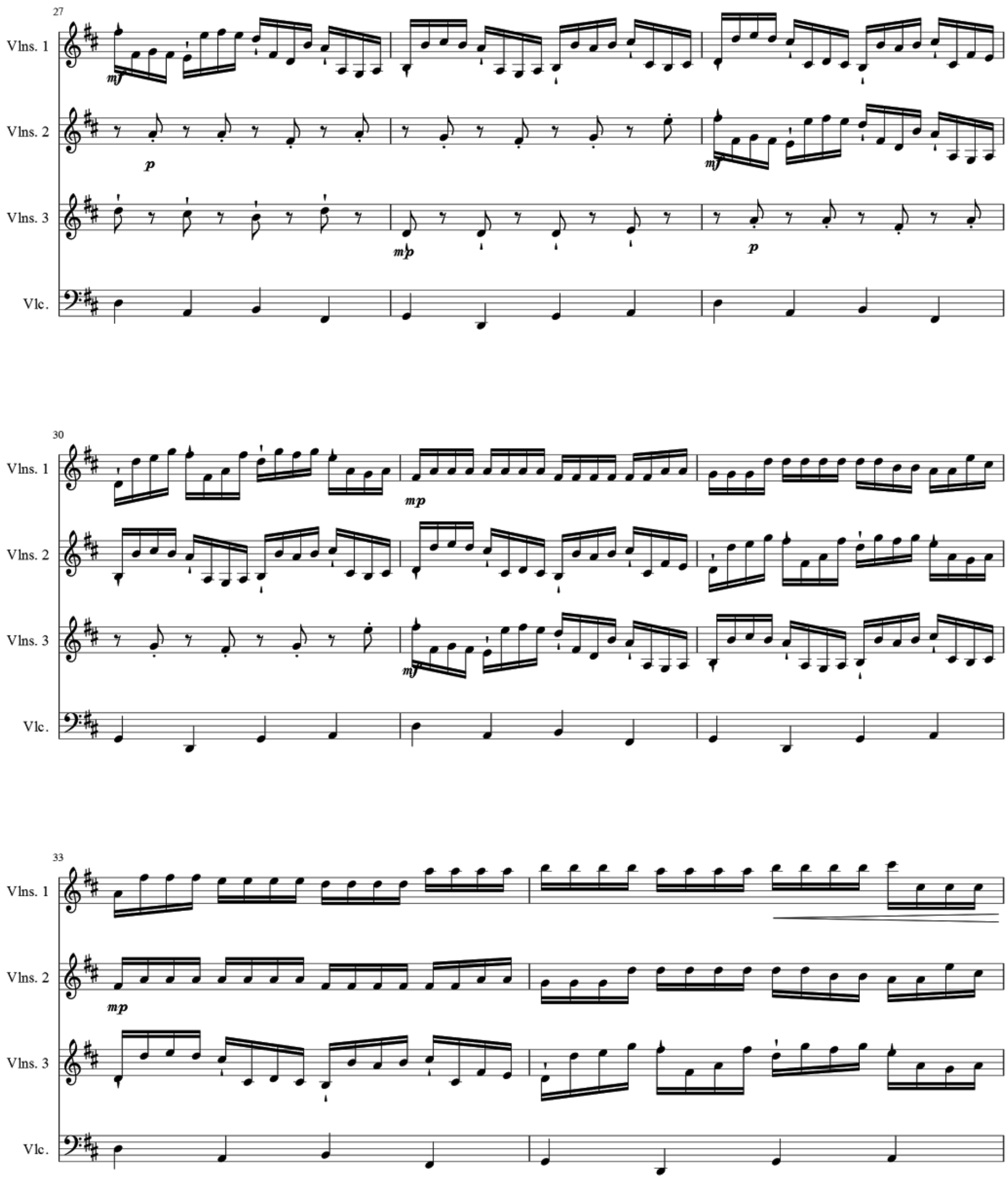

17
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
mp
mf
mp


19
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
mf
mp
mf

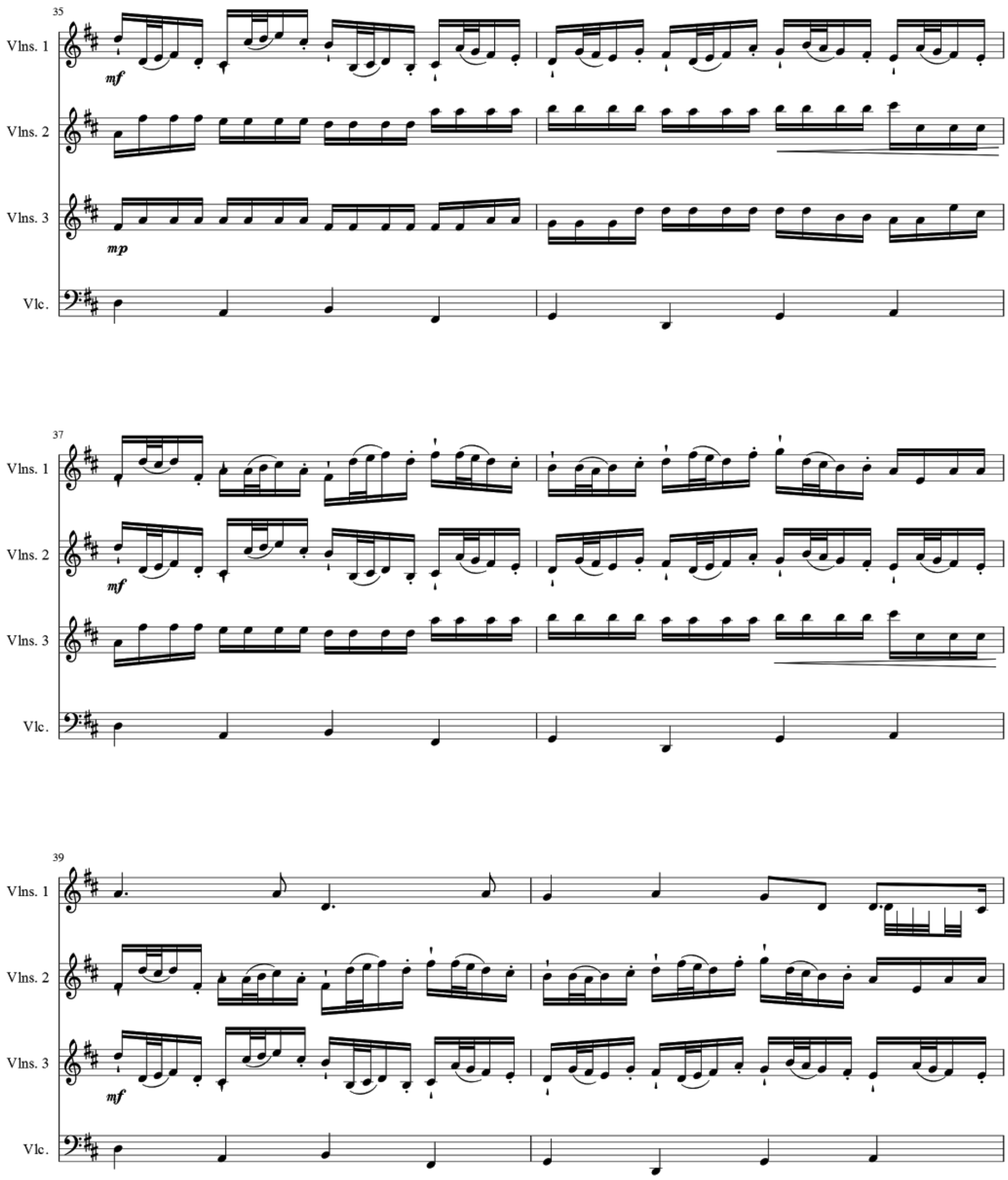
2


21
Vlns. 1
Vlns. 2
mf
Vlns. 3
mp
Vlc.
23
Vlns. 1
mp
Vlns. 2
Vlns. 3
mf
Vlc.
25
Vlns. 1
p
Vlns. 2
mp
Vlns. 3
Vlc.

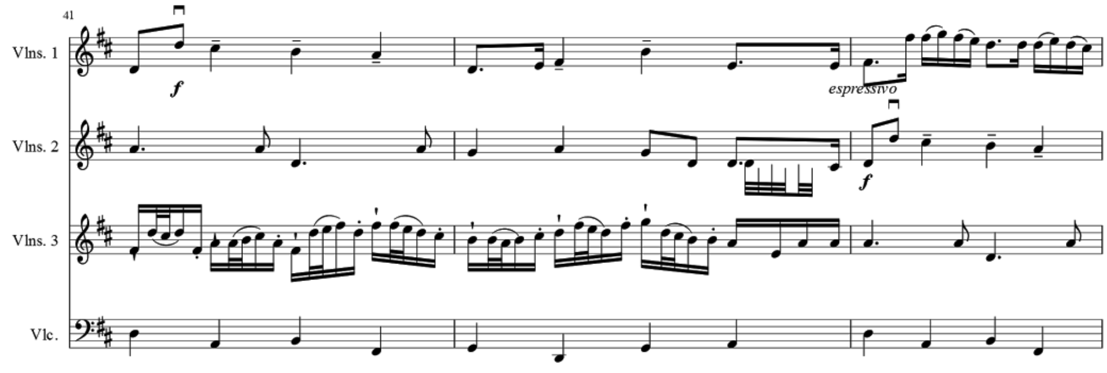
3

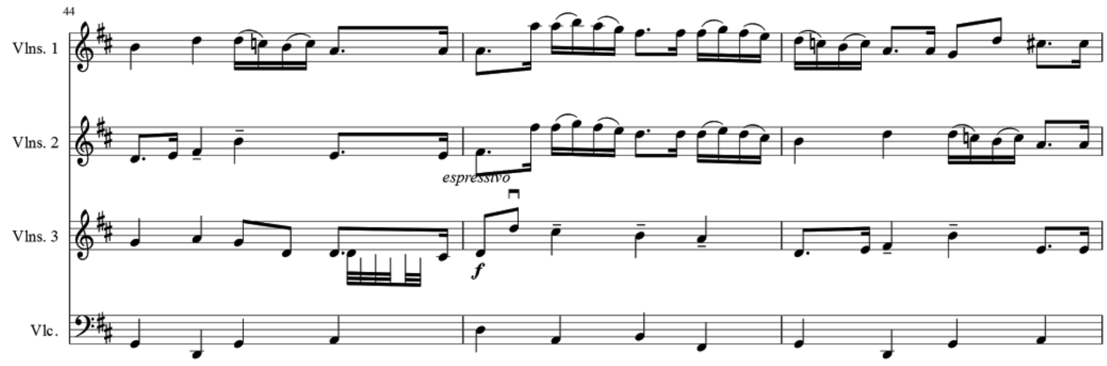


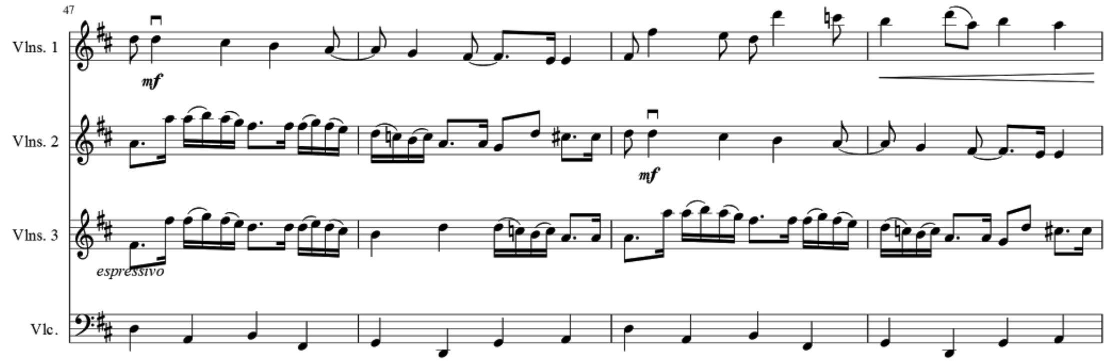
4


35
Vlns. 1
mf
Vlns. 2
Vlns. 3
mp
Vlc.
37
Vlns. 1
Vlns. 2
mf
Vlns. 3
Vlc.
39
Vlns. 1
Vlns. 2
Vlns. 3
mf
Vlc.

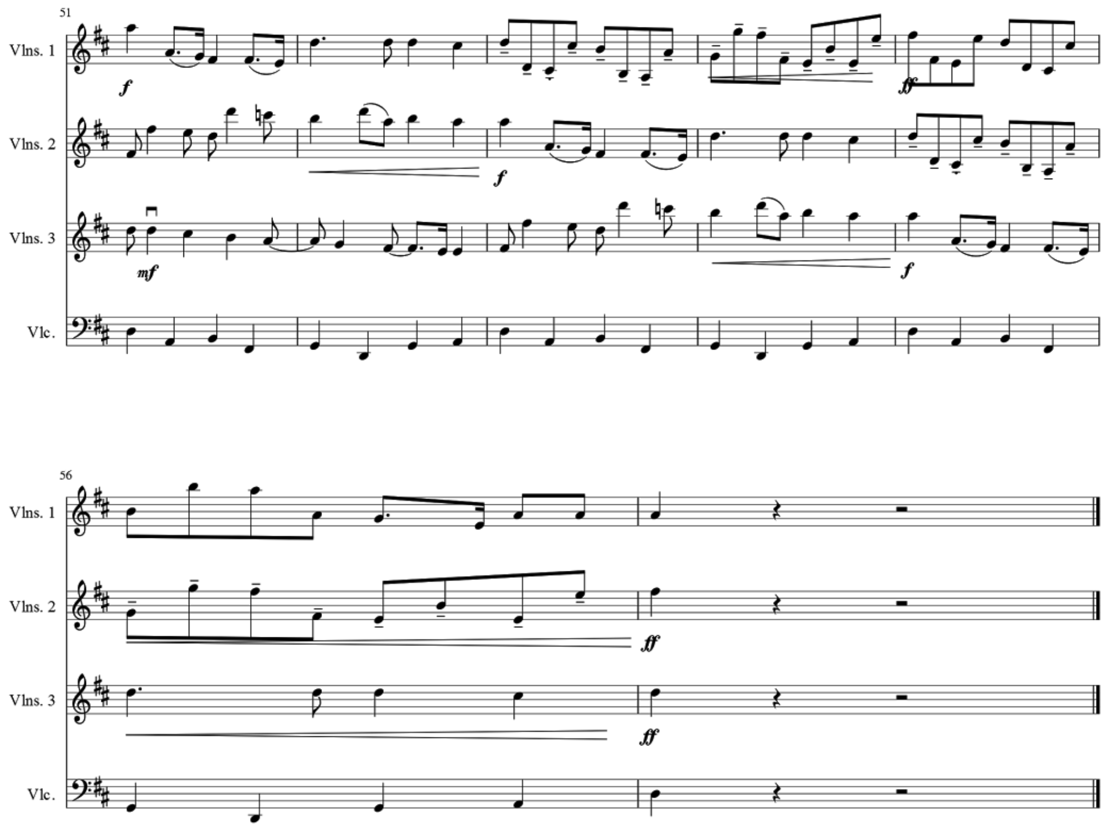
5


41
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
f
espressivo
f


44
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
espressivo
f

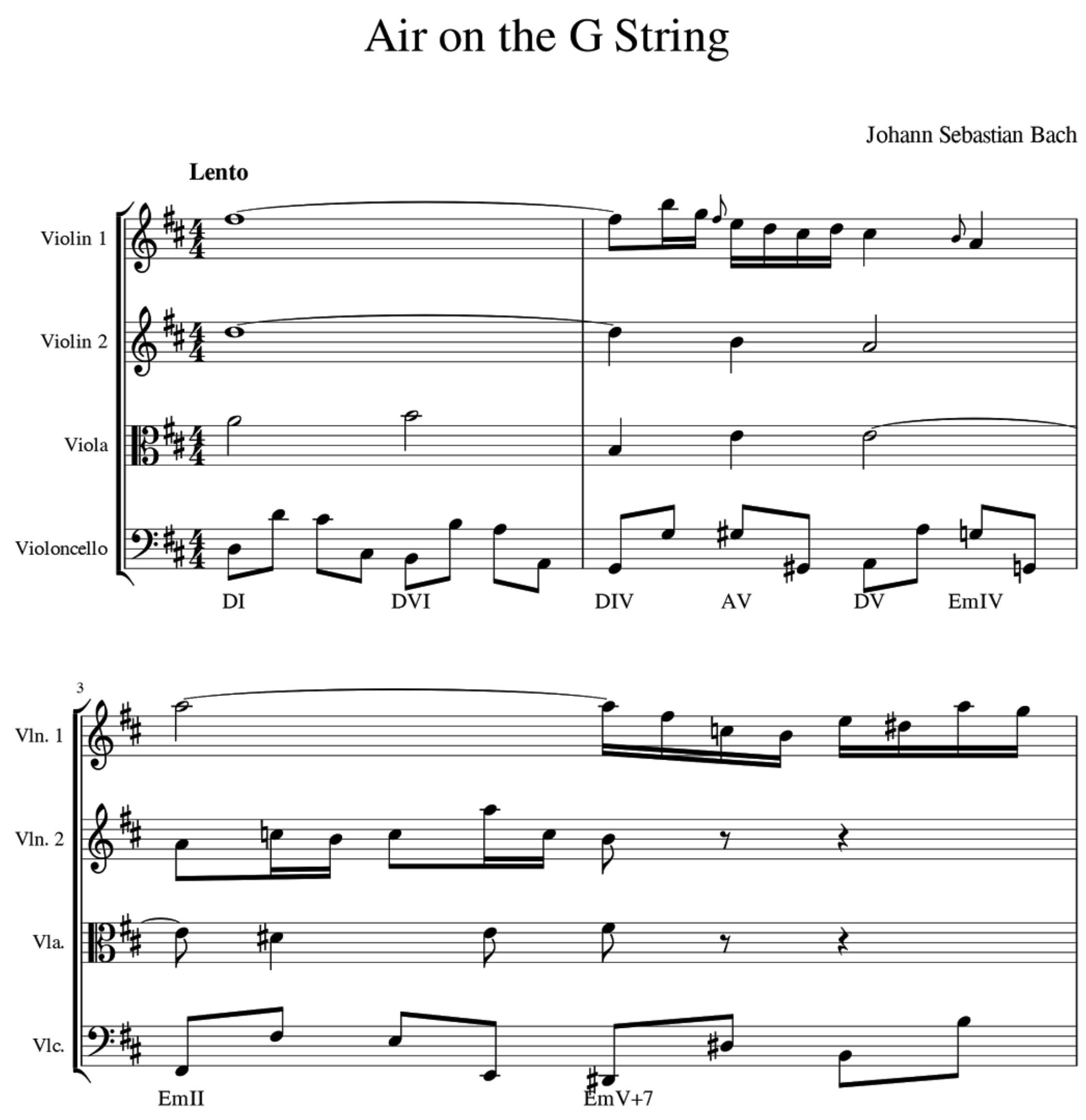

47
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
mf
mf
espressivo

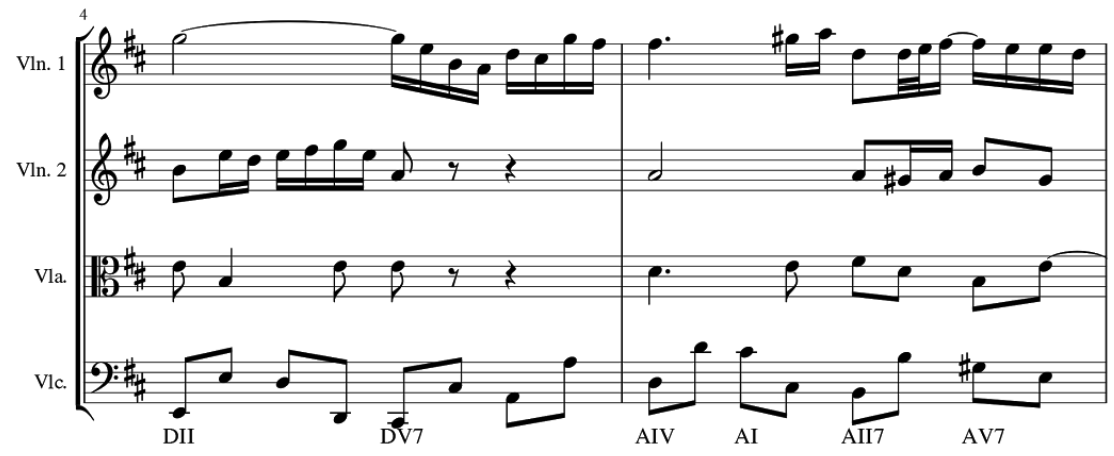
6

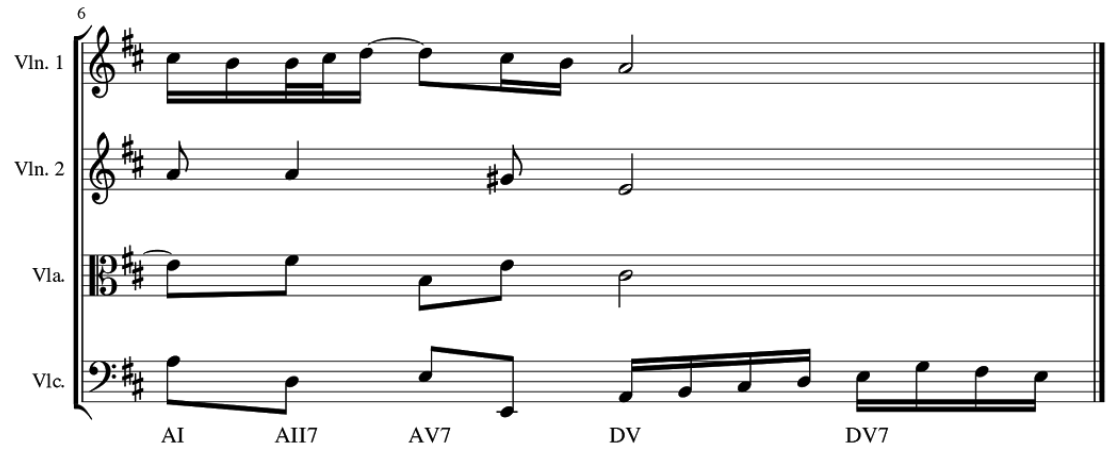

51
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
f
f
mf
f
ff
56
Vlns. 1
Vlns. 2
Vlns. 3
Vlc.
ff
ff

Appendixies D

Air on the G String (excerpts) Score




4
Vln. 1
Vln. 2
Vla.
Vlc.
DII
DV7
AIV
AI
AII7
AV7


6
Vln. 1
Vln. 2
Vla.
Vlc.
AI
AII7
AV7
DV
DV7

Appendixies E

ChordIdentifier module source code:

http://github.com/edmundyipll/ChordRecognizerProject