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

2017-18 Fall
Final Year Project
Term-End Report

KY1701 Automatic Piano Reduction
backend (chord Identification)

Students: Li Cheuk Yau (1155048011)
Jin Xiamu (1155046896)

Supervisors: Prof. Yip Yuk Lap, Kevin
Prof. Lucas Wong

Index

1. Background

1.1. Introduction

Music is always considered as one of the greatest arts in the world, it provides both recreational and spiritual comforts to human beings. Throughout the history, thousands of millions of musical masterpieces are created by musicians. However, one cannot always find all parts or all instruments that are used by the composer originally, especially when it comes to orchestral arrangement, where dozens of uncommon instruments used in a whole music piece. Therefore, performing reduction on complicated music pieces becomes a crucial topic.

However, reduction requires a lot of musical and mathematical analysis in the original score. Different musicians may hold different opinions on how the reduction should be processed. Thus, reduction is not a simple and unidimensional task that is easy to be dealt with.

Yet recently, machine learning technology has been improved and developed quickly, it becomes to be applied to almost every aspects in our daily life. Reduction using computational algorithms by learning musicians' reduction pattern becomes realistic. Obviously, reduction performed by machine learning method would be much less time-consuming compared with traditional human calculation, and, if optimal algorithms are applied, it would be even more precise than traditional methods.

The goal of the whole project is to develop an automatic piano reduction software tool. This year, our group focus on chord identifications. We continued on the works of the previous years, collaborating with the front-end team and machine learning team on developing a chord identification application. The main focus of our team is to improve the accuracy and develop a more general reduction method that can suit more types of pieces.

1.2. Music theory

1.2.1. Note and Accidental

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(~ 261 Hz) is the ‘middle C’ on the music keyboard and C5(~ 523 Hz) is another note named C which is an octave higher than C4.

Apart from the seven note used, there are totally five accidental symbols which are used to alter the note by ‘semitone’, or in other name, half step. A semitone or half step has a frequency ratio of $\sqrt[12]{2}$, approximately 1.059.

The five accidentals are: 1. Sharp ♯, raises a note by a semitone or half-step, and 2. Flat ♭, lowers it by the same amount. 3. Double-sharp 𝄪, raising the frequency by two semitones, and 4. Double-flat 𝄫, lowering it by two semitones. 5. A special accidental, the natural symbol ♮, is used to indicate an unmodified pitch.

1	2	3	4	5	6
C	C sharp (C♯)	D	D sharp (D♯)	E	F
	D flat (D♭)		E flat (E♭)		

7	8	9	10	11	12
F sharp (F♯)	G	G sharp (G♯)	A	A sharp (A♯)	B
G flat (G♭)		A flat (A♭)		B flat (B♭)	

Figure 1.2.1 Naming Convention of 12-tone Chromatic Scale Built on C

1.2.2. 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.

1.2.2.1. Chord Romanization

Roman numeral system bestows chords the “meaning” by using roman number (I,II,III,IV). It is also referred as scale degree in music. The key idea is to denotes a series of chords that are related to a particular music key. It is important because if we want to have a better picture of a music score, their music key must be known.

The table below shows a simplified roman numeral table in C major

I (major triad)	C E G (C major)

II (minor triad)	D A F (D minor)
III (minor triad)	E G B (E minor)
IV (major triad)	F A C (F major)
V (major triad)	G B D (G major)
VI (minor triad)	A C E (A minor)
VII (diminished triad)	B D F (B diminished)

1.2.2.2. Chord Inversion

Chord inversion inverts chords, rearrange the pitch order of how chords are played. For example, a C major chord is composed of C,E,G. If E becomes the bass note (E,G,C), the chord is in first inversion. If G becomes the bass note (G,C,E), the chord is in second inversion.

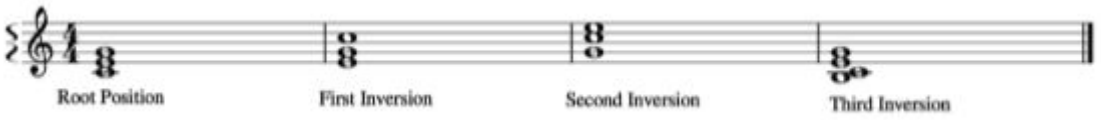


Figure 1.2.2.2 The root position and three inversions of C major 7th chord

1.2.3. Beat

1.2.3.1. Beat

In music and music theory, beat is literally the basic unit of time, the pulse, of the mensural level. By figuring out the beat of a certain music piece, we are not only able to understand the time attribute of the piece, we can also get clues of its rhythm pattern and chord pattern.

1.2.3.2. Time Signature

In music, a time signature tells you the meter of the piece you're playing. Composers decide the number of beats per measure early on and convey this information with a time signature. It is formed by two numbers -- the upper one denotes the number of beats and the lower one defines the unit of each beat. For example, a 4/4 time signature indicates that each measure contains four quarter note beats.

Figure 1.2.3.2 A 4/4 Time Signature

1.2.4. Key

In music theory, the key of a piece is the group of pitches, or scale that form the basis of a music composition in classical, Western art, and Western pop music. A key specify a set of notes that follow a specific pattern, which would be commonly used in the piece. The keys could be major or minor, both of which follow a similar pattern, containing a key signature and seven notes.

1.2.4.1. Key Signature

In musical notation, a key signature is a set of sharp (♯), flat (♭), and rarely, natural (♮) symbols placed together on the staff. It designates notes that are to be played higher or lower than the corresponding natural notes and applies through to the end of the piece or up to the next key signature. For example, the key signature of A major/ F# minor has three sharps, F#, C# and G#, indicating that every F, C and G has to be modified one semitone higher.

Figure 1.2.4.1 A major/ F# minor Key Signature

1.2.4.2. Major and Minor

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

We shall focus on the differences between the major key and the minor key by illustrating C major and C minor in the remaining of this section.

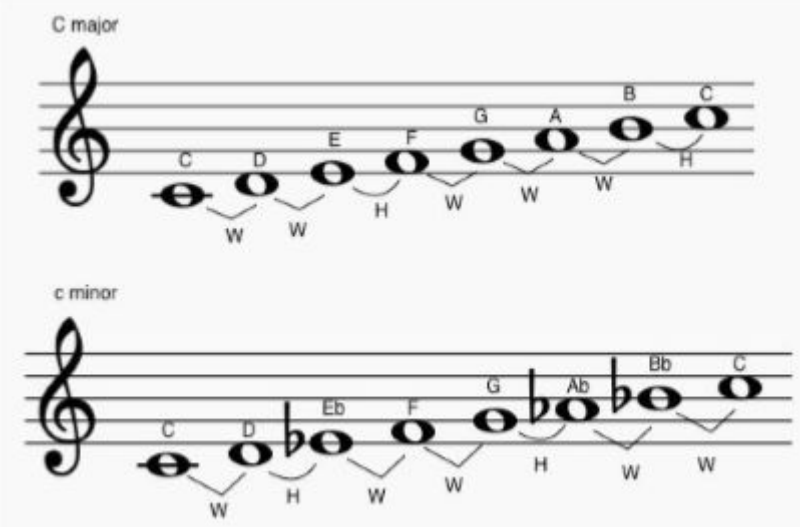


Figure 1.2.4.2 C major and C minor

The above figure 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' stands for a full step (2 semitones) and 'H' stands for a half step (1 semitones). In any scale of major and minor, there exists 5 full steps and 2 half steps with position same as the above figure.

1.2.5. Melodic Decoration

1.2.5.1. Passing Notes

A passing note falls in between two different notes a third apart. For example, as the figure below has shown, the notes C and E are a third apart. The D falls between them, so it is a passing note.

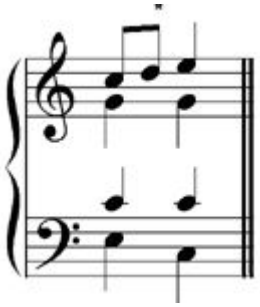


Figure 1.2.5.1 Passing Notes (the D between C and E)

1.2.5.2. Auxiliary Notes (also called "Neighbour Notes")

An auxiliary note falls between two identical chord notes. It can be higher or lower than the chord note. An auxiliary note which is higher than the chord note is an "upper auxiliary note" and a "lower auxiliary note" is lower than the chord note. Also, auxiliary notes can be either accented or unaccented, just like passing notes.

Figure 1.2.5.2 Auxiliary Note (marked with *)

1.2.5.3. Anticipation

An anticipation happens when we write one chord note earlier than the rest of the chord - in the beat before the rest of the chord sounds. Here, the B is part of the G major chord. The G major chord is sounded on the 2nd beat, but the B is sounded earlier, on the half beat before, so it is an anticipation. Anticipations are usually approached by a downwards motion (e.g the C falls to B).

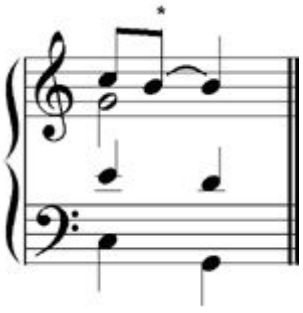


Figure 1.2.5.3 Anticipation

1.2.5.4. Suspension

Suspensions are the opposite of anticipations.

A suspension happens when we write one chord note later than the rest of the chord - during the beat after the rest of the chord sounds. In this example, the B doesn't sound immediately with the rest of the G major chord - instead, the C from the C major chord is held on for a little longer, and then falls to the B half a beat after the G major chord has sounded. The C is not part of the G major chord, so it is a non-chord note. The C is a suspension.

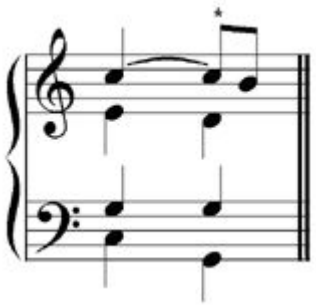


Figure 1.2.5.4 Suspension

1.2.5.5. Pedal

A pedal is either the tonic or dominant note played in one part continuously, while the chords in the other voices change.

Pedals normally occur in the bass, (but it is possible to find them in any of the other voices too). The pedal note is either held on for a long time, or repeated several times.

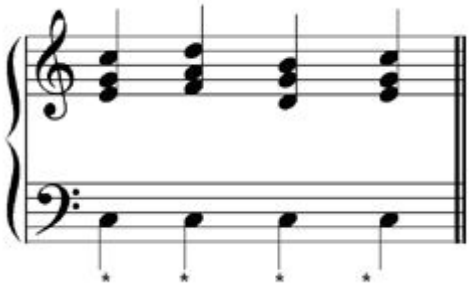


Figure 1.2.5.5.1 A Tonic Pedal

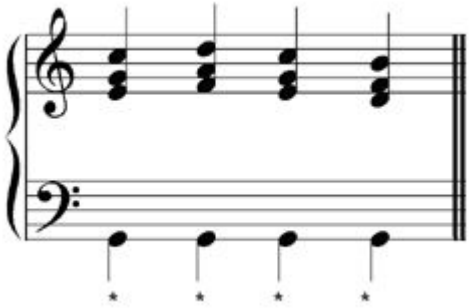


Figure 1.2.5.5.2 A Dominant Pedal

1.2.6. 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.

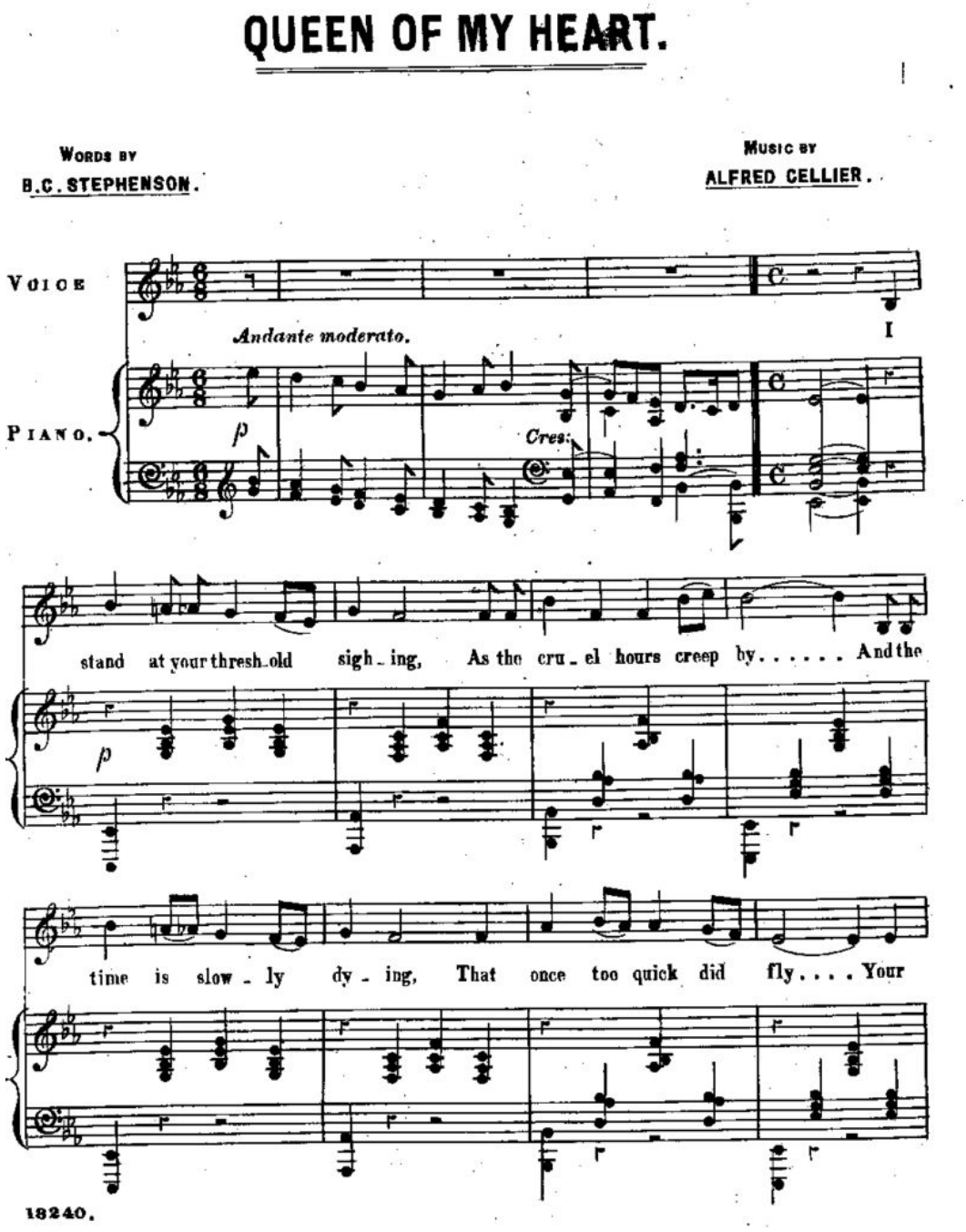


Figure 1.2.6 (https://en.wikipedia.org/wiki/Sheet_music#/media/File:QoMH.png)

1.3. Technical support

1. Python 3.6: main computer language for programming.
2. music21: a toolkit for computer-aided musicology. It is used for parsing and retrieving music information of files in MusicXML format.
3. MusicXML: a standard format for representing digital sheet music.
4. MuseScore: open source music composition and notation software for viewing and editing music files, including MusicXML.

2. Overall System

2.1. Chord Identification

2.1.1. Preprocessing

Lines for different instruments can be written in the same staff by two voices or chords. In order to handle cases of chords. We use a build-in function **Stream.voicesToParts()** in music21 library.

Here is an example extracted from KV588

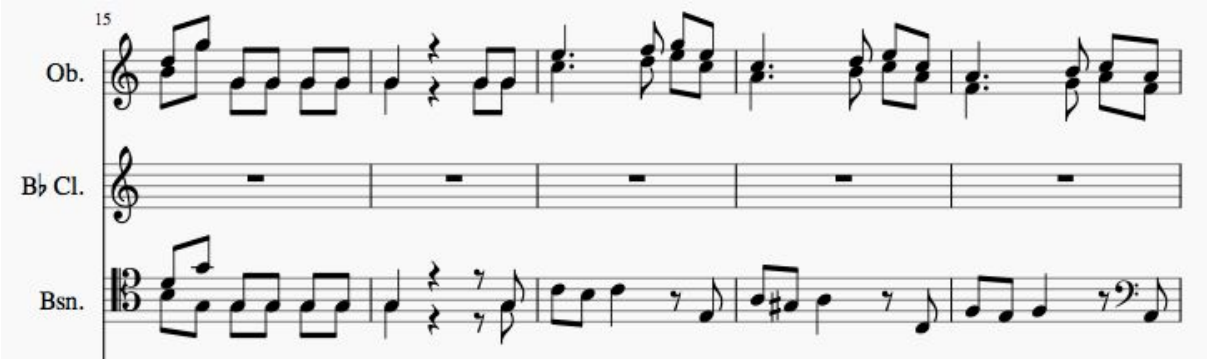


Figure 2.1 Original Score extract from KV588

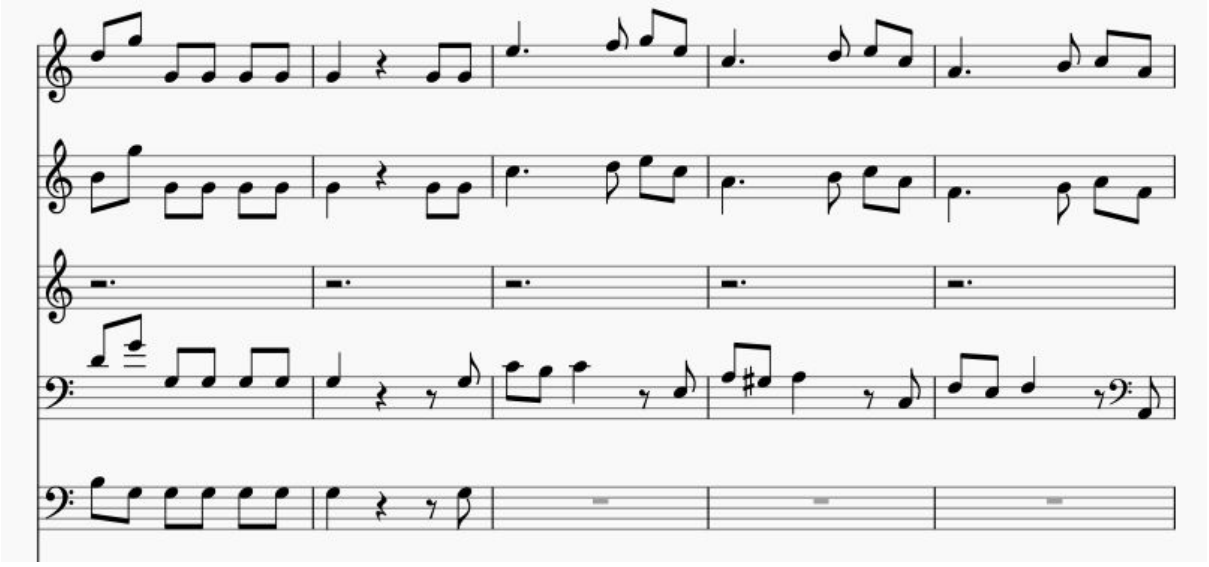


Figure 2.2 Modifing Score with voicesToParts()

In the above example, First Oboe, Second Oboe and First Bassoon, Second Bassoon are written on the same staff in the original score. They are separated to different staff after the process. Noted that the measures gray in color do not exist. The last note of Second Bassoon is G3.

To facilitate slicing measures in to beat segments, **Stream.sliceByBeat()** is introduced for the whole score, which slice all elements in the Stream that have a Duration at the offsets determined to be the beat from the local ***TimeSignature***.

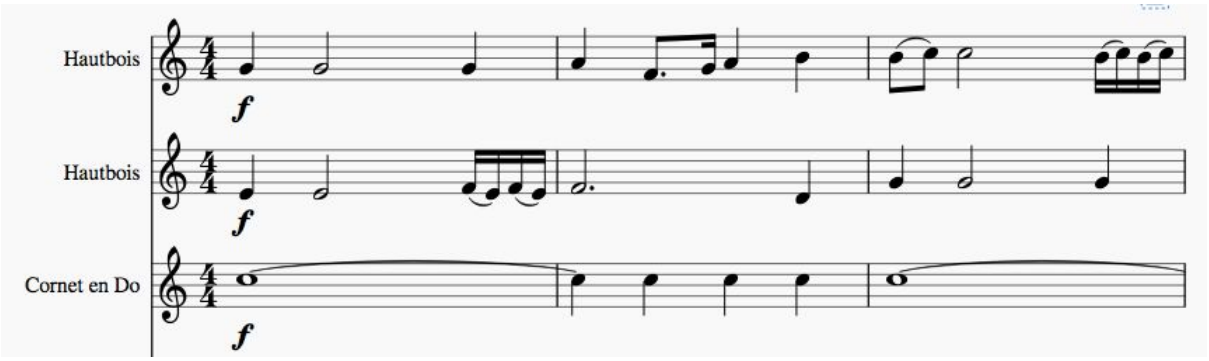


Figure 2.3 Original Score from K314

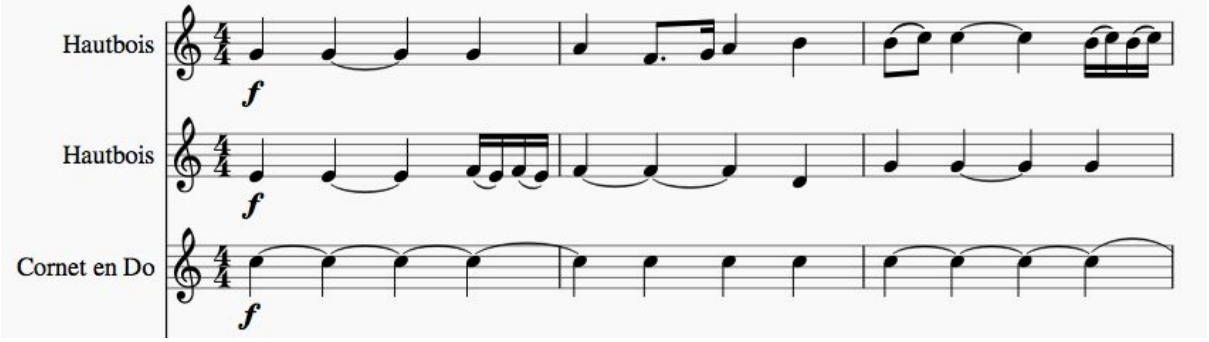


Figure 2.4 Modifing Score with sliceByBeat()

Notes will to changed to tied notes if their duration is longer than a beat length.

2.1.2. Segment slicing

We assume that music pieces have chord change approximately once in every measure, at most once in every stressed beat. We define each measure as a segment and split it into sub-segments by the stressed beat.

For example, the stressed beats in a 4/4 signature time would be the 1, 3 beat.

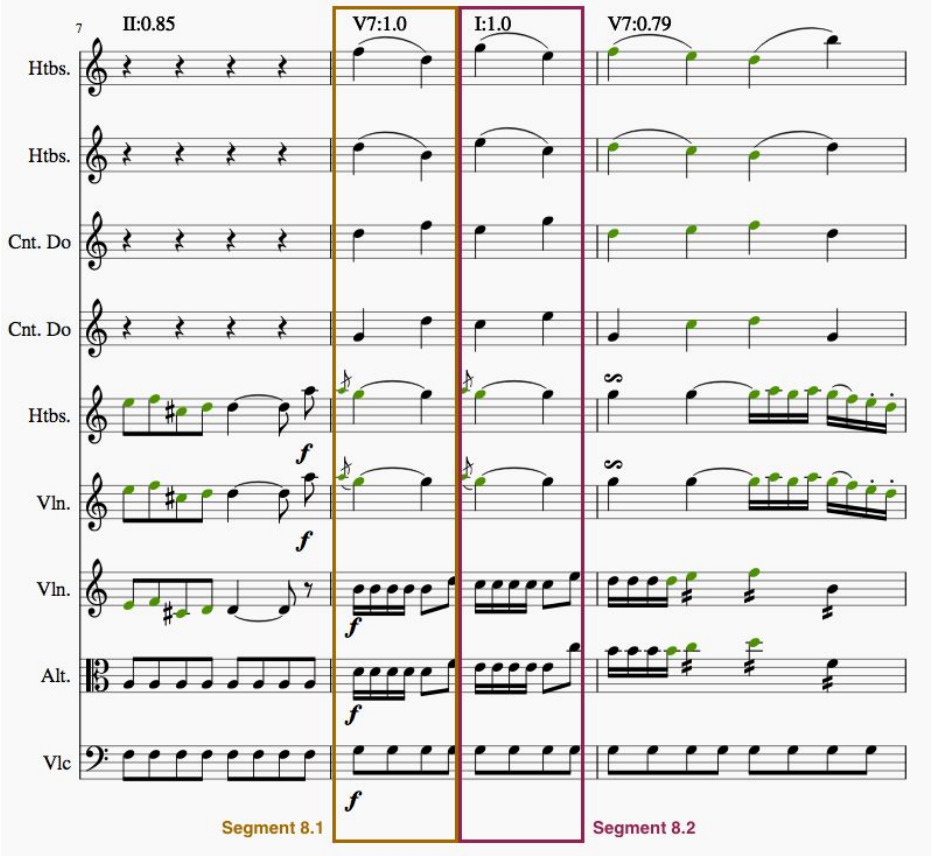


Figure 2.5 Segment slicing to two sub-segments

Each segments and sub-segments will be further processed for chord identification.

2.1.3. Weight Calculation

Weight of each note in a segment is determined by two factors: duration and the possibility to be a melodic decoration.

For duration, we use the quarter note length as the weight. For example, an eighth note has length of 0.5 and a quarter note has length of 1. (see 3.1.2.4. for implementation details)

Regarding the possibility of being a melodic decoration, we use the intervals between neighbouring notes for pattern recognition. As mentioned in section 1.2.5., for most

melodic decoration types, the dissonant notes have a stepwise interval change with its neighbouring notes most of the time. We use it as an indicator of possible dissonant note and give them 0 weighting, i.e. ignoring them in weight calculation. (see 3.1.1. 3.1.2.3. for implementation details)

In the following example, notes in green are ignored due to stepwise interval change with any of their neighbours. The first staff only has ‘C’ with weight 1 and the second staff has ‘G’ with weight 4.

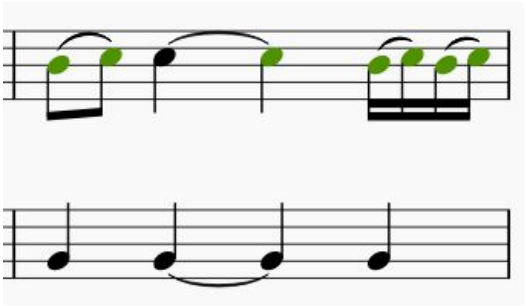


Figure 2.5 Example from K314, only first two staffs are shown

By accumulating the weight of the same pitch, a note dictionary with weight is generated. It is then further reduced to a note list, sorted by the weight in descending order.
For example, a dictionary {'A': 20.0, 'F': 7.0, 'D': 10.0, 'C': 6.0} will be reduced to [‘A’, ‘D’, ‘F’, ‘C’]
The sorted note list is returned for chord matching.

2.1.4. Chord Matching in Hashing Tree

To enhance the performance, we decided to use hashing tree structure for all the possible combination of chord notes and their corresponding chord name. For reasons of choosing this data structure please refer to section 3.2.

Here is a simplified graphical illustration for one of the branch ‘M6’ in major mode.

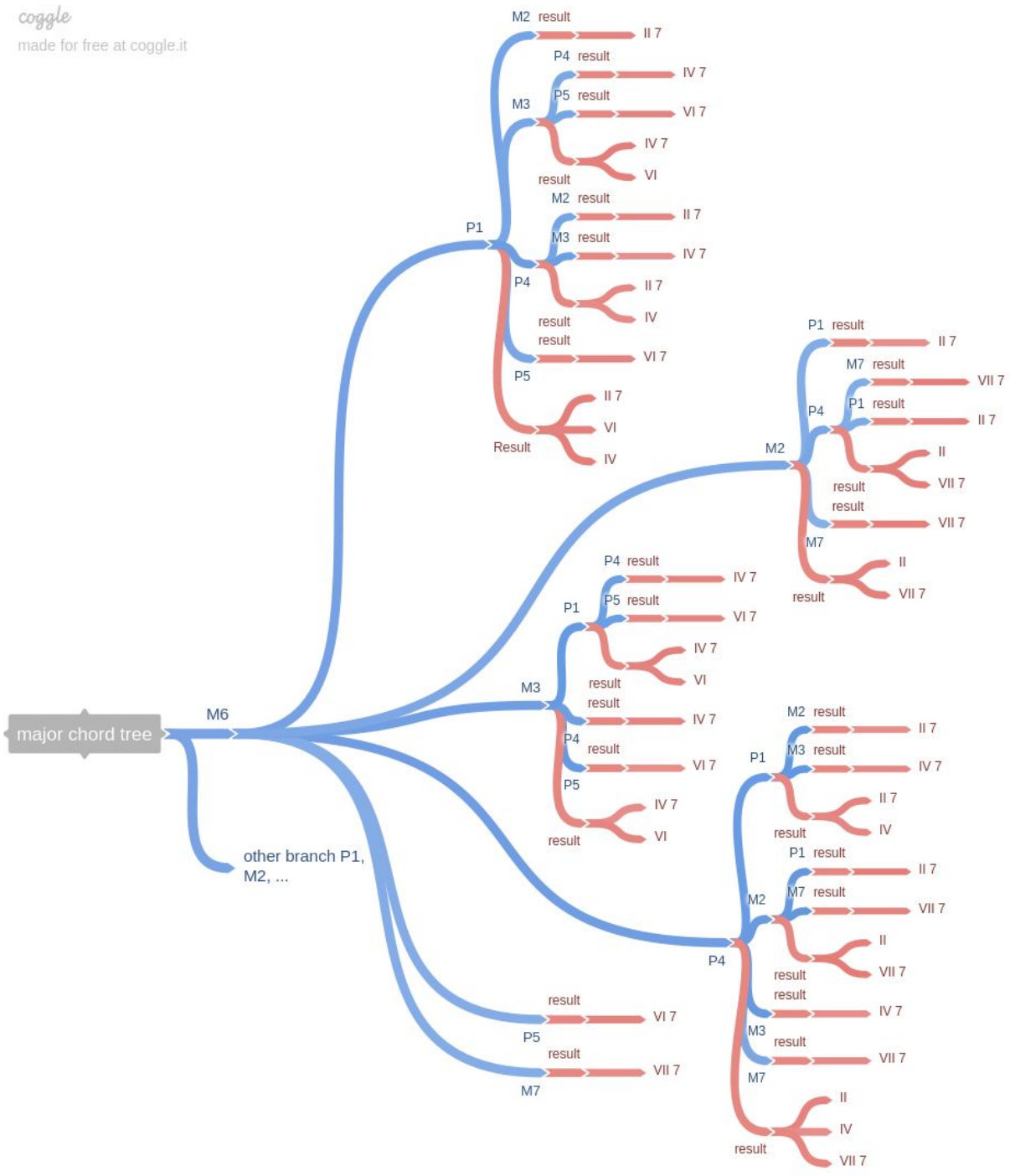


Figure 2.5 Tree illustration for branch ‘M6’ in major chord tree

Two trees are generated for both major and minor. In order to handle cases of different keys. Interval name with the tonic note is used instead of the note name. Therefore, the outputting note list from section 2.1.3 is needed to be converted into an interval list before searching.

Using the example in section 2.1.3., [‘A’, ‘D’, ‘F’, ‘C’] in C major key will be converted into [‘M6’, ‘M2’, ‘P4’, ‘P1’]. Best match of this interval list is chord ‘II7’.

For cases having dissonant notes in the interval list, e.g. [‘M6’, ‘M3’, ‘P1’, ‘M2’], the best match are chords ‘IV7’ ‘VI’ as there is no branch ‘M2’ under ‘P1’. For cases of having more than one possible chord choice, score calculation is introduced to select the best choice (see next section for the details).

2.1.5. Score Calculation

There are two stages for score calculation.

In the first stage, score calculation is done for each possible chord choice of segments and sub-segments. The score is a tuple ($Score_1$, $Score_2$) containing two parts:

1. Percentage of notes in a segment which are included in the chord:

$$Score_1 = \frac{\sum_{n \in N} W(n)}{\sum_{n \in N_S} W(n)},\ N = N_C \cap N_S$$

2. Percentage of notes in a chord which are included in the segment:

$$Score_2 = \frac{\sum_{n \in N} 1}{\sum_{n \in N_C} 1},\ N = N_C \cap N_S$$

N_C is the note set of the chord choice. N_S is the note set of the segment. W(n) is the weight of note n in the segment (refer to 2.1.3 ***Weight Calculation***).

By comparing Score1 then by Score2, best chord is selected for each segment and sub-segments. For example, in {'I': (1.0, 1.0), 'VI 7': (1.0, 0.75)}, chord I is the best chord since both ‘I’ and ‘VI 7’ has the same $Score_1$ and $Score_2$ of ‘I’ is higher than that of ‘VI 7’.

In the second stage, score of segment is compared with that of its sub-segments. Here is an example selected from K314 measure 8.

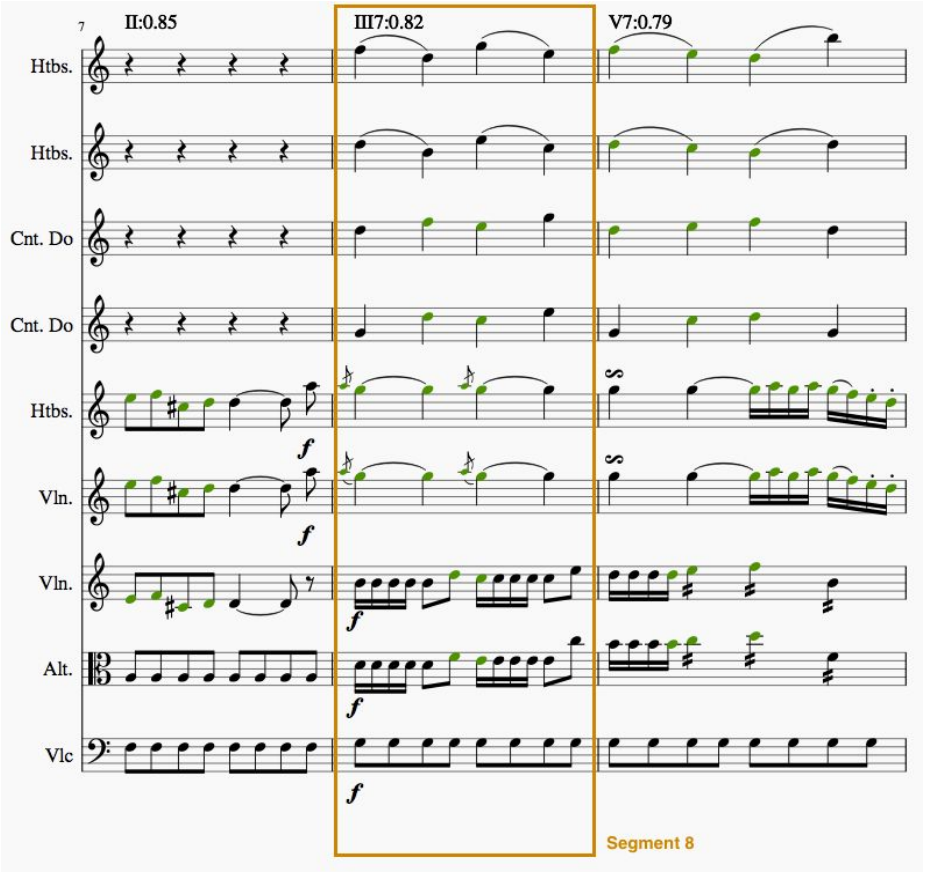


Figure 2.6 Example from K314. Best chord = {'III 7': (0.82, 1.0)}

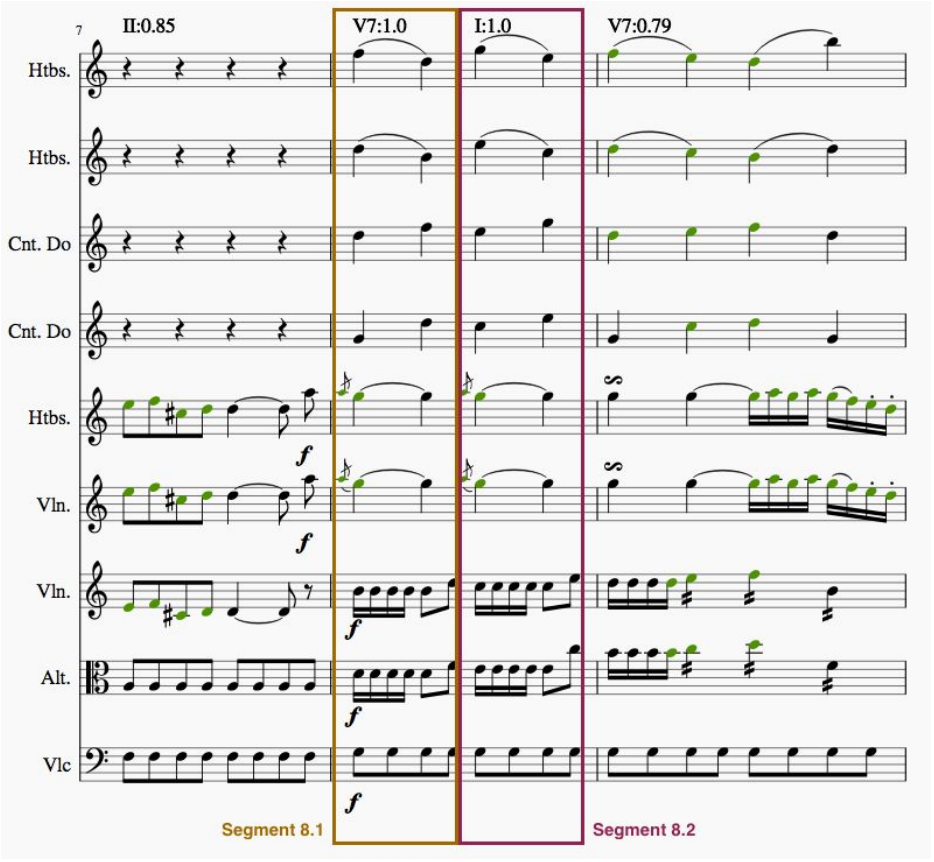


Figure 2.7 Example from K314. Best chord = {'V 7': (1.0, 1.0)} & {'I': (1.0, 1.0)}

If all scores of sub-segments are higher than that of the segment, chords of the sub-segments is chosen. Therefore in the above example, we choose ‘V7’ and ‘I’ as the final best chords for measure 8.

3. Implementation Details

3.1. Weight Calculation

3.1.1. Interval Calculation

3.1.1.1. Approach

In order to minimize the Time needed for calculating ***Step Score*** of each note. A interval list is generated before doing ***Step Score*** calculation. There are two major procedure for doing the interval calculation.

3.1.1.2. Getting Interval Group between Chords and Notes

Interval calculation between two notes can be done easily. However, preprocess is needed for interval between chords and notes. In music21, a chord is denoted by a list containing notes.

Dealing with Intervals between notes and chords, we use the following approach:

1. We use a ***list*** to store all intervals between the notes and chords. We called it ***Interval Group***
2. size of a ***Interval Group*** = size of the note/chord with more number of note
3. for every note with index *i*, do interval calculation with neighbouring note that has the same index *i*, or the last note if *i* doesn't exist

In the following examples, interval list of example1 = [['M10', 'M6', 'P1'], ['P1', 'P5', 'M10']]. interval list of example2 = [['P5', 'P1'], ['P1', 'P1']]

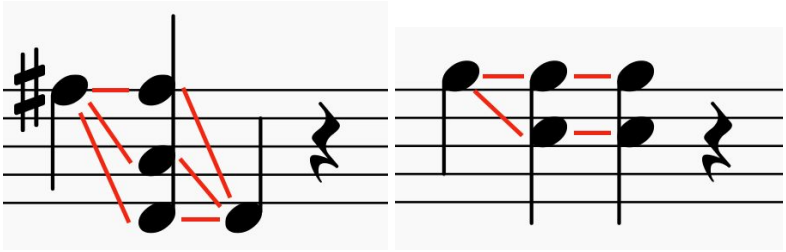


Figure 3.1 Two examples on interval list mapping

Here is the pseudocode for **getIntervalGroupOfNotes()**

```
function getIntervalGroupOfNotes(n1,n2):
      intervalGroup = []
      if n1, n2 is not list: n1=[n1], n2=[n2]
      if len(n2) > len(n1):
            # interchange n1 n2
      for index, note1 in n1:
            if n2 has index1:
                  note2 = n2[index]
```

```
            else:
                  note2 = last(n2)
            intervalGroup.append(interval.Interval(note1,note2))
      return intervalGroup
```

3.1.1.3. Getting Intervals List from Note List

The idea can be explained using the following example. A note list with n elements returns an interval list of n-1 elements containing the interval between neighbouring notes, except cases that the neighbouring note is a rest.
Human percepts that notes are continuous even though a short rest is in between. It is feasible to calculate the interval between notes before and after the rests. In order not to reduplicate the procedure, calculation is only done for the note after rests (4 in Figure 3.2), while others are denoted as *None* (3 in Figure 3.2).
This approach turns out to be beneficial that it simplifies the whole procedure.

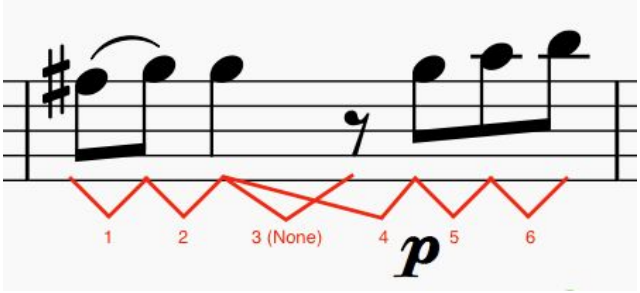


Figure 3.2 example of intervals list

Here is the pseudocode for **getIntervalList()**

```
function getIntervalList(noteList):
      intervalList = []
      for index, note in noteList:
            if note is last(noteList): break
            n1 = note
            n2 = noteList[index+1]
            while n1 is Rest and index > 0:
                  n1 = noteList[--index]
            if n1 is Rest or n2 is Rest:
                  intervalList.append(None)
            else:

intervalList.append(getIntervalGroupOfNotes(n1,n2))
      return intervalList
```

3.1.2. Weight Analysis

3.1.2.1. Approach

We use the stepwise characteristics and duration of each note for the calculation of weight. Three lists are generated (***Pitch List***, ***Step Weight List***, ***Duration Weight List***) from each staff in the segment for simplifying the calculation. The lists then undergo reduction to get the final weighted note list.

3.1.2.2. Pitch List

Only pitch name of each note is extracted but not its octave. Pitches with same name but from different octave can be mapped to the same key in later stage (see 3.1.2.5. Reduce Stage).

The pseudocode of **getPitchList()**:

```
function getPitchList(noteList):
      pitchList = []
      for n in noteList:
            if n is Rest:
                  pitch = [None]
            if n is Note:
                  pitch = [n.name]
            if n is Chord:
                  weight = [a.name for a in n]
```

3.1.2.3. Step Weight List

Making use of the ***Interval List*** generated from the previous section (see 3.1.1. Interval Calculation), we can assign weight to each note in the input ***Note List***.

For each note / chord with index *i* in ***Note List***, intervalList[i-1] is the interval with the previous note / chord and intervalList[i] is that with the next note / chord.

For cases with rests, in order not to reduplicate the procedure, calculation is only done for the note after rests (as mentioned in section 3.1.1.3.). The approach we use is that when we are getting the Interval Group with next note / chord and a value *None* is met, we will loop through the interval list for the next value until a non *None* ***Interval Group*** is got. For example, considering the marked note in Figure 3.3, the ***Interval Group*** with the next note (intervalList[2]) is *None*. We will use the next value intervalList[3] instead. For Figure 3.4, since there is no more value after intervalList[4], it will return *None*.

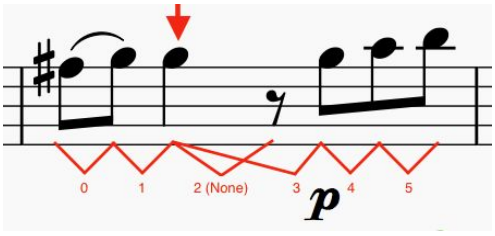


Figure 3.3 Example of interval list

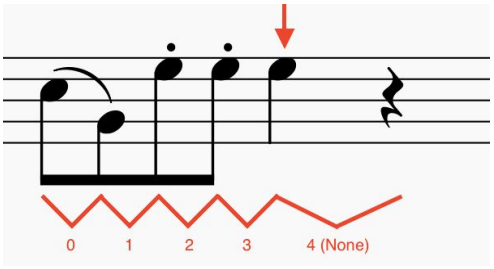


Figure 3.4 Example of interval list

For each ***Interval Group*** extracted, we will assign weight to each note using the intervals. Since we are trying to ignore all the notes with stepwise change, 0 and 1 is assigned in order to keep or remove the reduced score using multiplication in the next stage (see 3.1.2.5. Reduction Stage). 0 weight is assigned if the interval with the previous / next note is a step, else 1 is assigned.

Pseudocode of **getStepWeightList()**:

```
function getStepWeightList(noteList, intervalList):
      weightList = []
      for index, n in noteList:
            subWeightList = []
            if n is rest:
                  weight = [0]
            # 1: intervals with previous note/chord
```

```
            if intervalList has key index-1:
                  interval1 = intervalList[index-1]
            else:
                  interval1 = None
            for index1, l1 in interval1:
                  if l1 is Step:
                        subWeightList[index1] = 0
                  else:
                        subWeightList[index1] = 1

            # 2: intervals with next note/chord
            while interval2 is None and intervalList has key index:
                  interval2 = intervalList[index]
                  index++
            for index2, l2 in interval2:
                  if l1 is Step:
                        subWeightList[index2] = 0
            weightList.append(subWeightList)
      return weightList
```

3.1.2.4. Duration Weight List

In music21, duration of each note can be represented by quarter note length. We use the built-in attribute ***Note.duration.quarterLength*** to extract it in float number.

```
function getDurationWeightList(noteList):
      weightList = []
      for n in noteList:
            if n is Rest:
                  weight = [0]
            if n is Note:
                  weight = [n.duration.quarterLength]
            if n is Chord:
                  weight = [n.duration.quarterLength] * len(n)
#number of note in chord
            weightList.append(weight)
```

3.1.2.5. Reduction Stage

In python, we can simply use one line of code to reduce the above three lists into a list of tuple (pitch, score) with the follow:

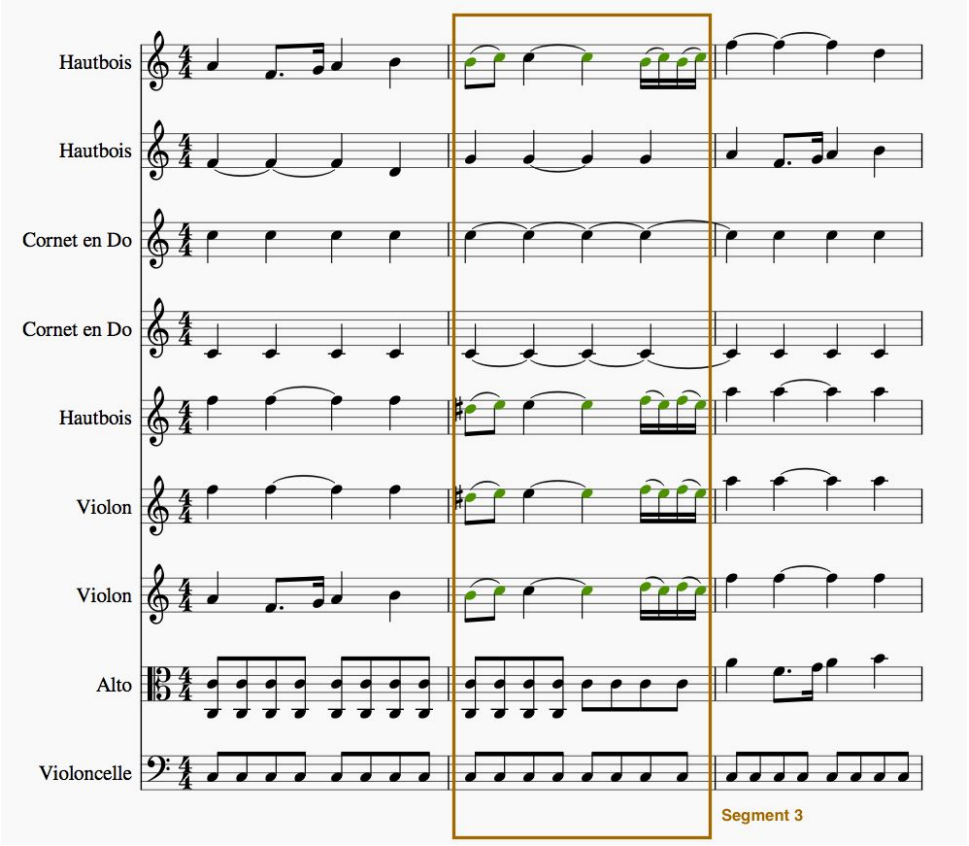
```
pitchScoreTuples = [(a,b*c) for w,x,y in zip(pitchList,
stepWeightList, durationWeightList) for a,b,c in zip(w,x,y)]
```

The score of each tuple is accumulated to a ***Pitch Weight Dictionary***. Here is the pseudo code for **addWeightToDict()**.

```
function addWeightToDict(pitchWeightDict, pitchScoreTupleList):
      for pitch, score in pitchScoreTupleList:
            if pitch not equal None and score not equal 0:
                  pitchWeightDict[pitch] += score
      return pitchWeightDict
```

3.1.2.6. Example

Here is an example of doing the weight analysis in K314 measure 3.



pitchList	stepWeightList	durationWeightList	pitchScoreTuple
['B'], ['C'], ['C'], ['C'], ['B'], ['C'], ['B'], ['C']	[0], [0], [1], [0], [0], [0], [0], [0]	[0.5], [0.5], [1.0], [1.0], [0.25], [0.25], [0.25], [0.25]	('B', 0.0), ('C', 0.0), ('C', 1.0), ('C', 0.0), ('B', 0.0), ('C', 0.0), ('B', 0.0), ('C', 0.0)
['G'], ['G'], ['G'], ['G']	[1], [1], [1], [1]	[1.0], [1.0], [1.0], [1.0]	('G', 1.0), ('G', 1.0), ('G', 1.0), ('G', 1.0)
['C'], ['C'], ['C'], ['C']	[1], [1], [1], [1]	[1.0], [1.0], [1.0], [1.0]	('C', 1.0), ('C', 1.0), ('C', 1.0), ('C', 1.0)

['C'], ['C'], ['C'], ['C']	[1], [1], [1], [1]	[1.0], [1.0], [1.0], [1.0]	('C', 1.0), ('C', 1.0), ('C', 1.0), ('C', 1.0)
['D#'], ['E'], ['E'], ['E'], ['F'], ['E'], ['F'], ['E']	[0], [0], [1], [0], [0], [0], [0], [0]	[0.5], [0.5], [1.0], [1.0], [0.25], [0.25], [0.25], [0.25]	('D#', 0.0), ('E', 0.0), ('E', 1.0), ('E', 0.0), ('F', 0.0), ('E', 0.0), ('F', 0.0), ('E', 0.0)
['D#'], ['E'], ['E'], ['E'], ['F'], ['E'], ['F'], ['E']	[0], [0], [1], [0], [0], [0], [0], [0]	[0.5], [0.5], [1.0], [1.0], [0.25], [0.25], [0.25], [0.25]	('D#', 0.0), ('E', 0.0), ('E', 1.0), ('E', 0.0), ('F', 0.0), ('E', 0.0), ('F', 0.0), ('E', 0.0)
['B'], ['C'], ['C'], ['C'], ['D'], ['C'], ['D'], ['C']	[0], [0], [1], [0], [0], [0], [0], [0]	[0.5], [0.5], [1.0], [1.0], [0.25], [0.25], [0.25], [0.25]	('B', 0.0), ('C', 0.0), ('C', 1.0), ('C', 0.0), ('D', 0.0), ('C', 0.0), ('D', 0.0), ('C', 0.0)
['C', 'C'], ['C', 'C'], ['C', 'C'], ['C', 'C'], ['C'], ['C'], ['C'], ['C']	[1, 1], [1, 1], [1, 1], [1, 1], [1], [1], [1], [1]	[0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5, 0.5], [0.5], [0.5], [0.5], [0.5]	('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5)
['C'], ['C'], ['C'], ['C'], ['C'], ['C'], ['C'], ['C']	[1], [1], [1], [1], [1], [1], [1], [1]	[0.5], [0.5], [0.5], [0.5], [0.5], [0.5], [0.5], [0.5]	('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5), ('C', 0.5)
		pitchWeightDict	'C': 20.0, 'G': 4.0, 'E': 2.0

3.2. Chord Matching

As mensioned in section 2.1.4., a multidimensional dictionary is used in python for searching.

There are two major reasons of choosing this data structure:

1. Small time complexity in searching, which is approximately O(1)
2. Support easy searching of total match, exact match, possible match
    a. total match: The incoming pitch list is exactly the same as the best matching chord.
    b. exact match: Considered possible combination of notes in chord, e.g. ['A', 'C'] can be chord VI
    c. possible match: Serve the most weighted note first, until it has the best matching chord (greedy approach)

The following is the pseudocode of matching a pitch list with the best chord:

```
function getPossibleChords(pitchList, tonicNote):
      tree = majorChordTree or minorChordTree
      bestIndex = -1
      for index, pitch in pitchList:
            interval = interval between tonicNote and pitch
            if interval not in tree:
                  break
            bestIndex = index
            tree = tree[interval]
      return tree.matchChord
```

3.3. Score Calculation

We use the following function to calculate score1 and score2 for the equations in section 2.1.5.

Pseudocode for **getScoreOfChord()**:

```
function getScoreOfChord(chordPitchList, pitchDuration):
      totalDuration = 0;
      inChordDuration = 0;
      for pitch, duration in pitchDuration:
            if pitch in chordPitchList:
                  inChordDuration += duration
            totalDuration += duration
      score1 = inChordScore/totalScore
      score2 = sum(chordPitchList in
pitchDuration)/len(chordPitchList)
```

4. Testing Results

We generated two rudimental testing result using the following pieces

1. Score 1: W. A. Mozart, Oboe Concerto in C major, K. 314
2. Score 2: Beethoven Ludwig van, Symphony No.5, Op.67

For the output scores, please refer to Appendix.

5. Future Work

5.1. Key Change

The current program can suit into measures that don't have the key changes only. Key change identification will be introduced in the coming semester.

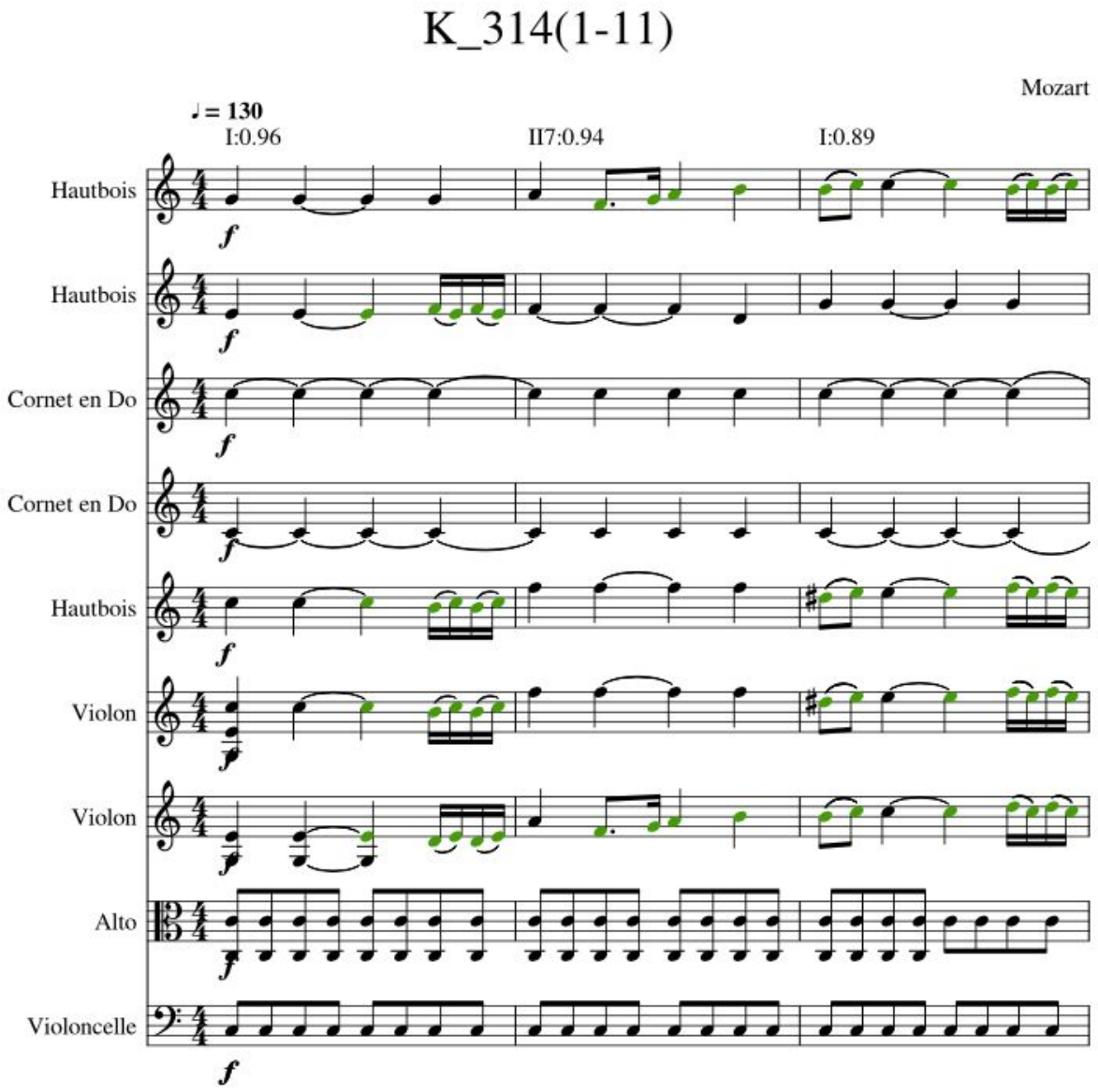
5.2. Transposition of instruments

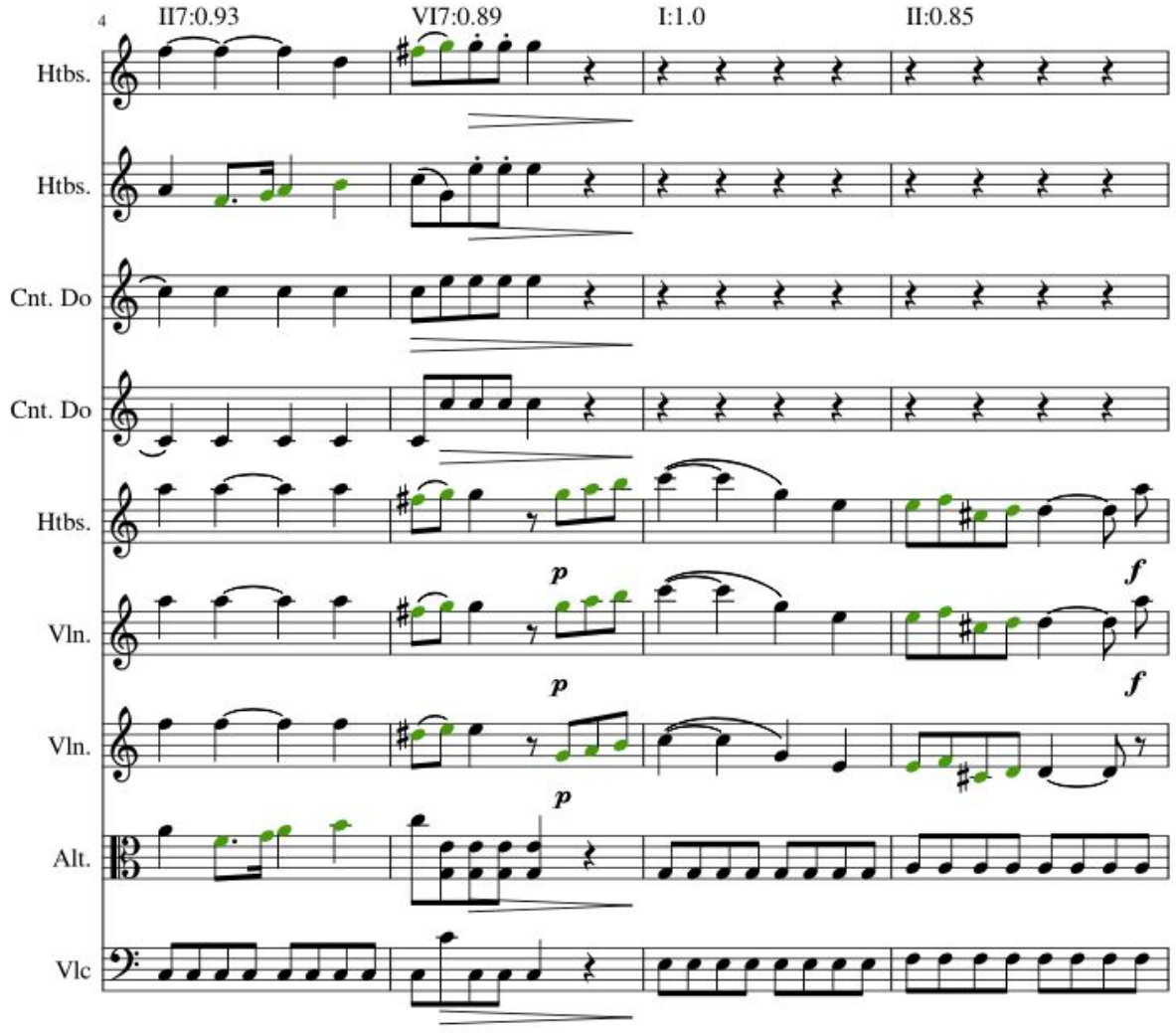
In musical ensembles, instruments using other keys are oftenly seen. Preprocess is need to identify these transposed staff before doing chord identification.

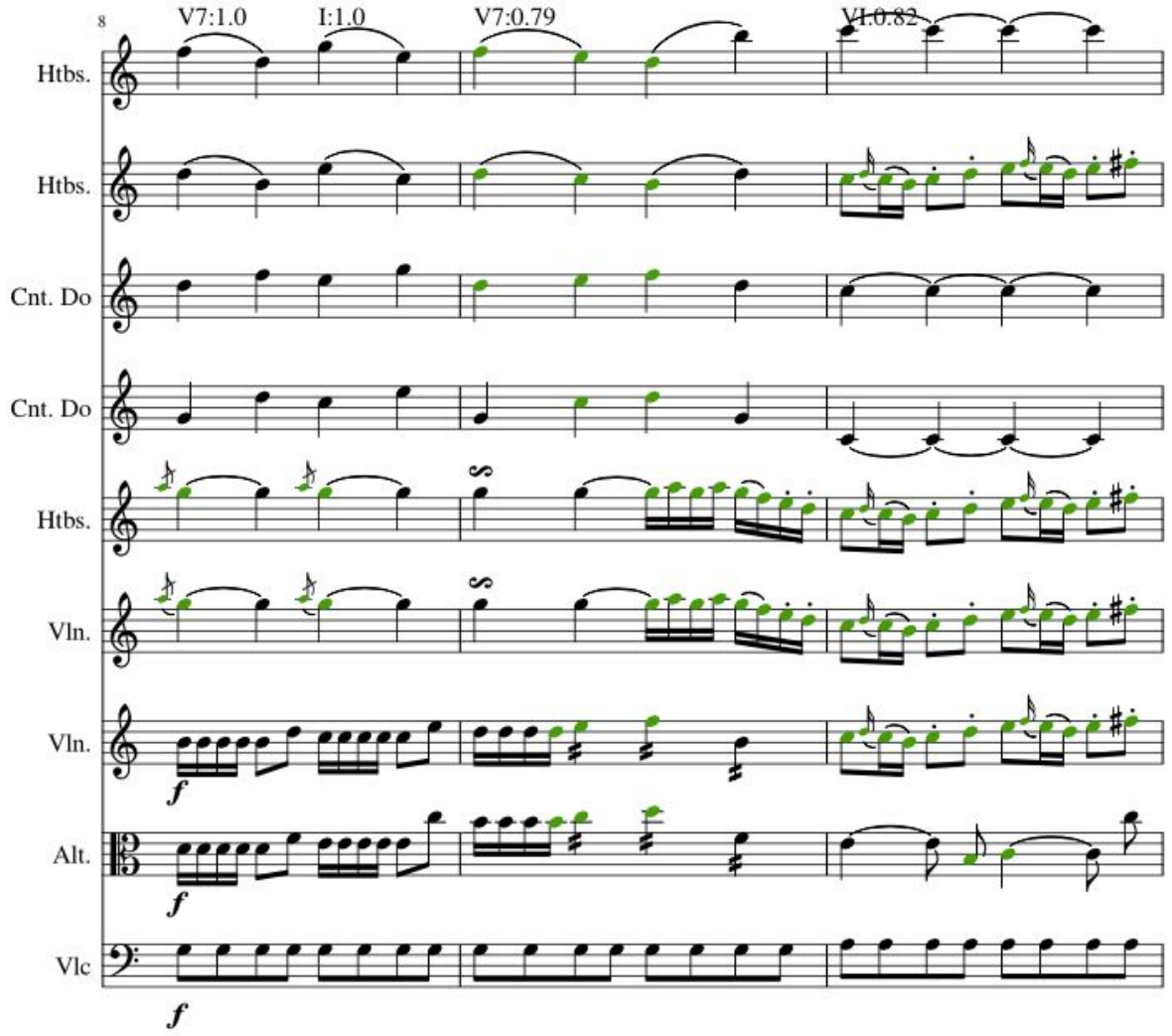
5.3. Prevent overfitting

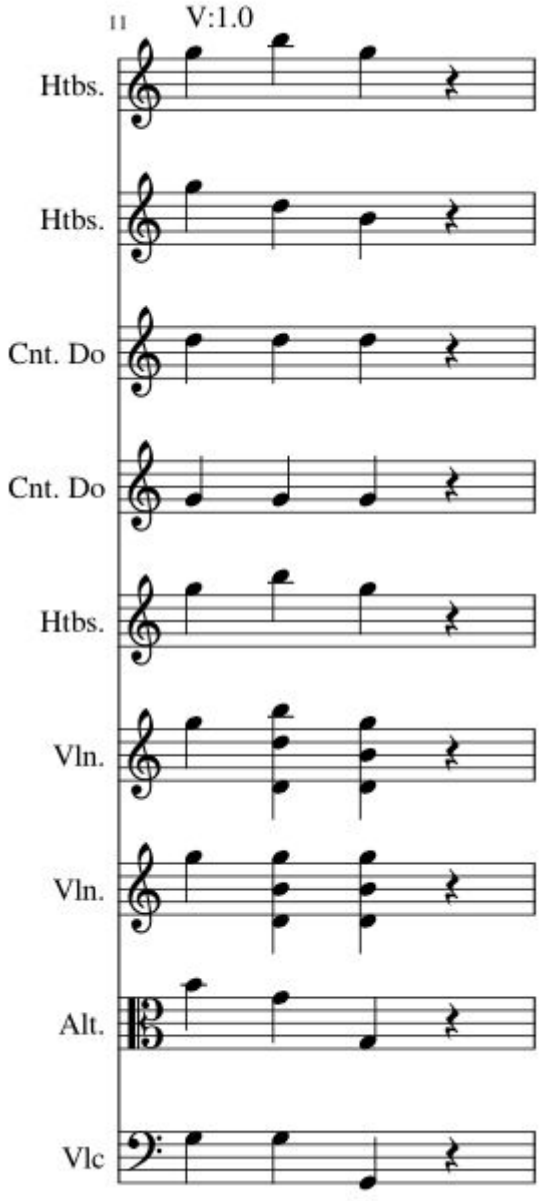
In the rudimental testing outputs, there are cases that a chord is wrongly identify to a 7 chord due to some dissonant notes. For example in K314 measure 2, a IV chord is wrongly identified as a II7 chord. More solutions will be introduced in the coming semester, e.g. score of chord progressing.

Appendix


K_314(1-11)
Mozart
♩ = 130
I:0.96
II7:0.94
I:0.89
Hautbois
Hautbois
Cornet en Do
Cornet en Do
Hautbois
Violon
Violon
Alto
Violoncelle
f

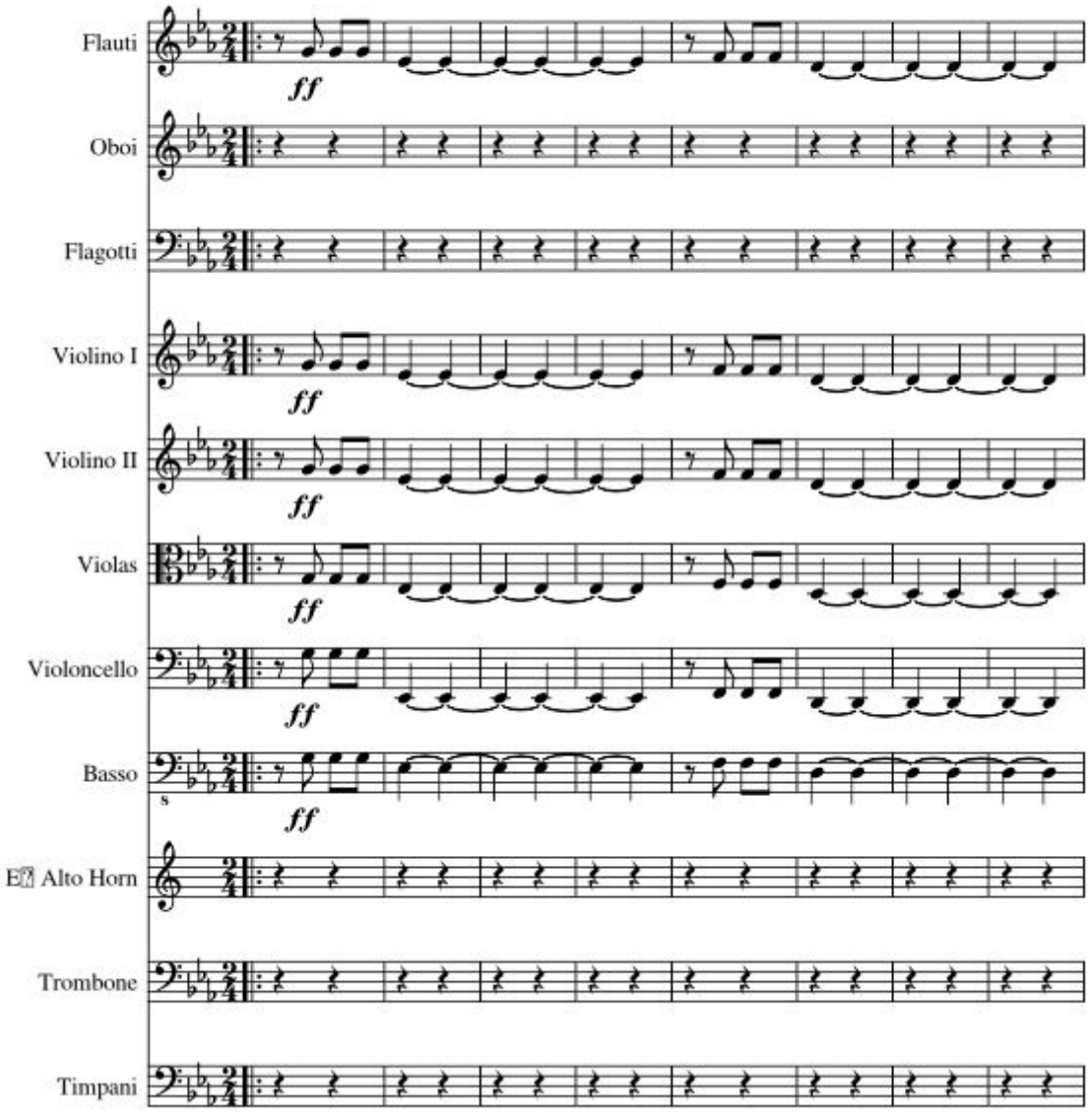

4
II7:0.93
VI7:0.89
I:1.0
II:0.85
Htbs.
Htbs.
Cnt. Do
Cnt. Do
Htbs.
Vln.
Vln.
Alt.
Vlc
p
p
p
f
f

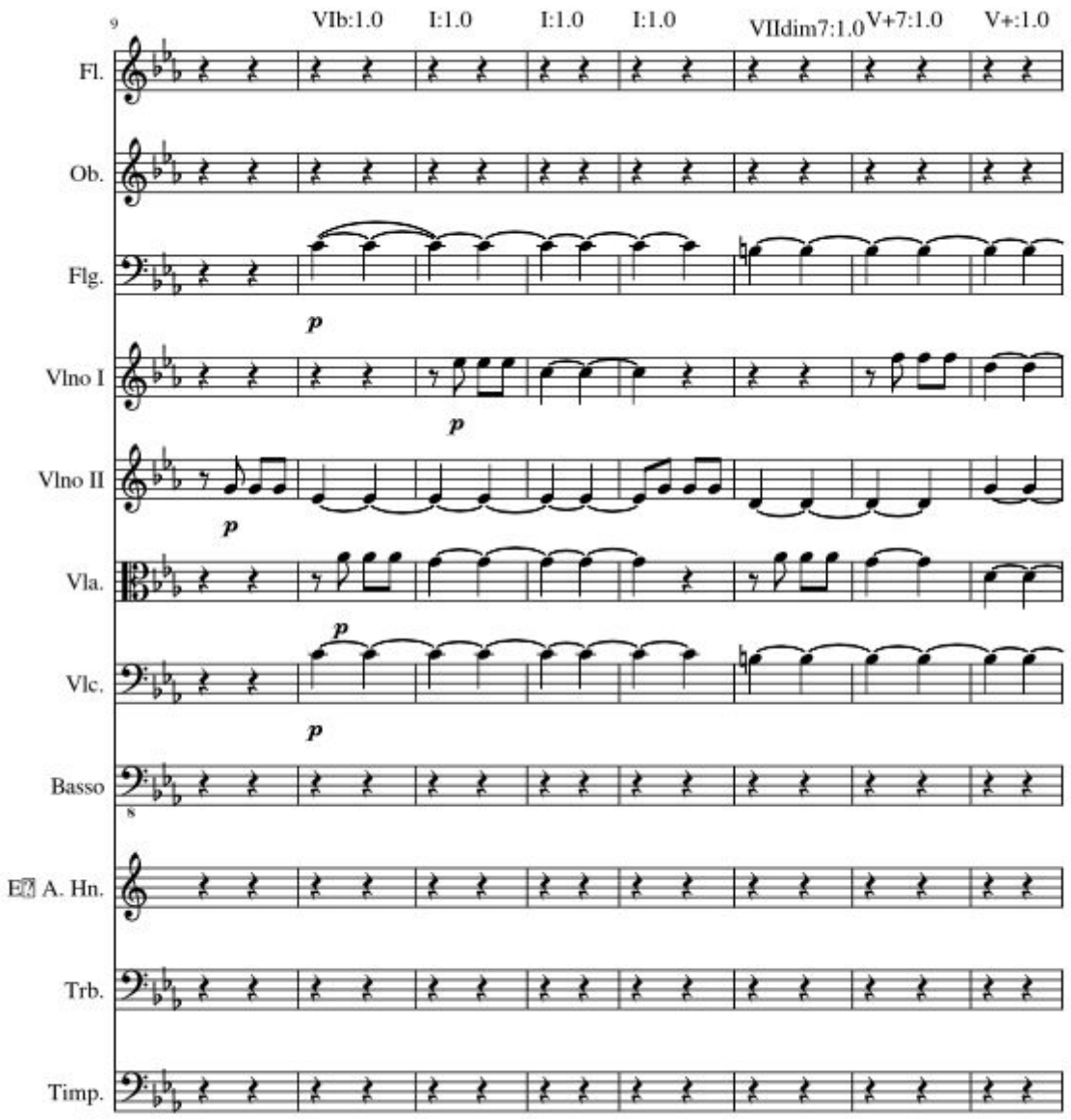

8
V7:1.0
I:1.0
V7:0.79
V1.0.82
Htbs.
Htbs.
Cnt. Do
Cnt. Do
Htbs.
Vln.
Vln.
Alt.
Vlc
f
f
f

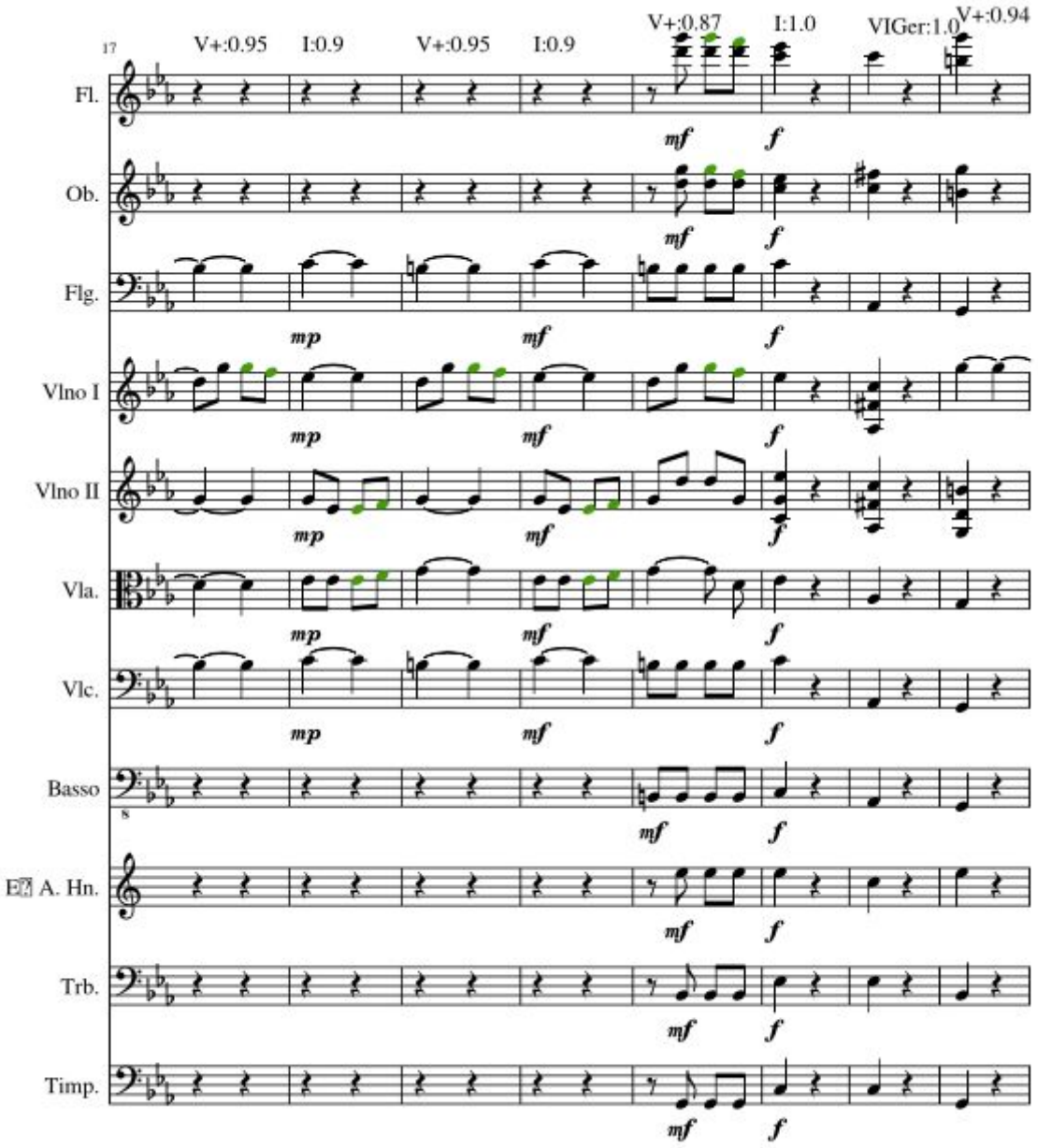

11
V:1.0
Htbs.
Htbs.
Cnt. Do
Cnt. Do
Htbs.
Vln.
Vln.
Alt.
Vlc

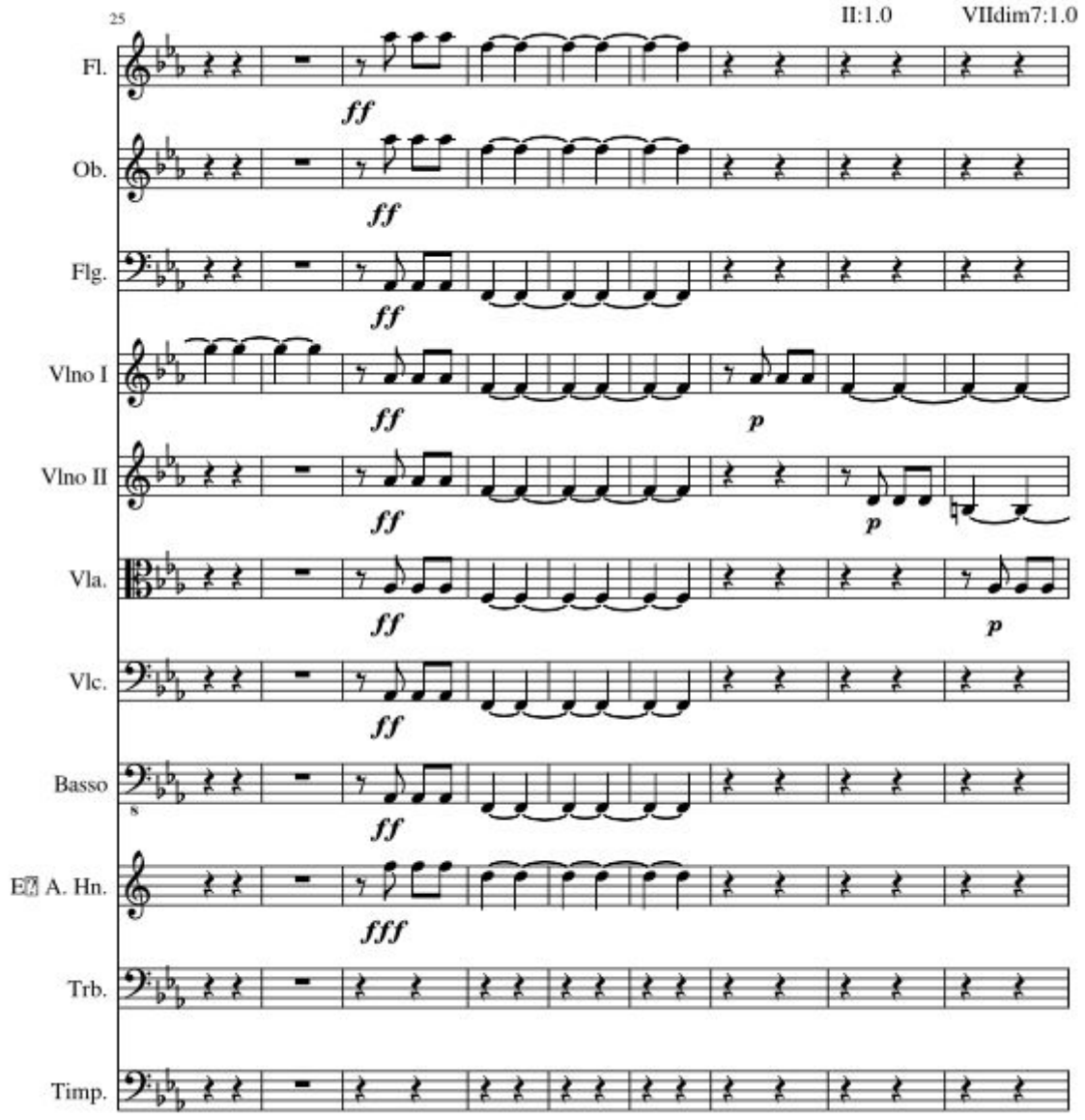
OP_67.xml

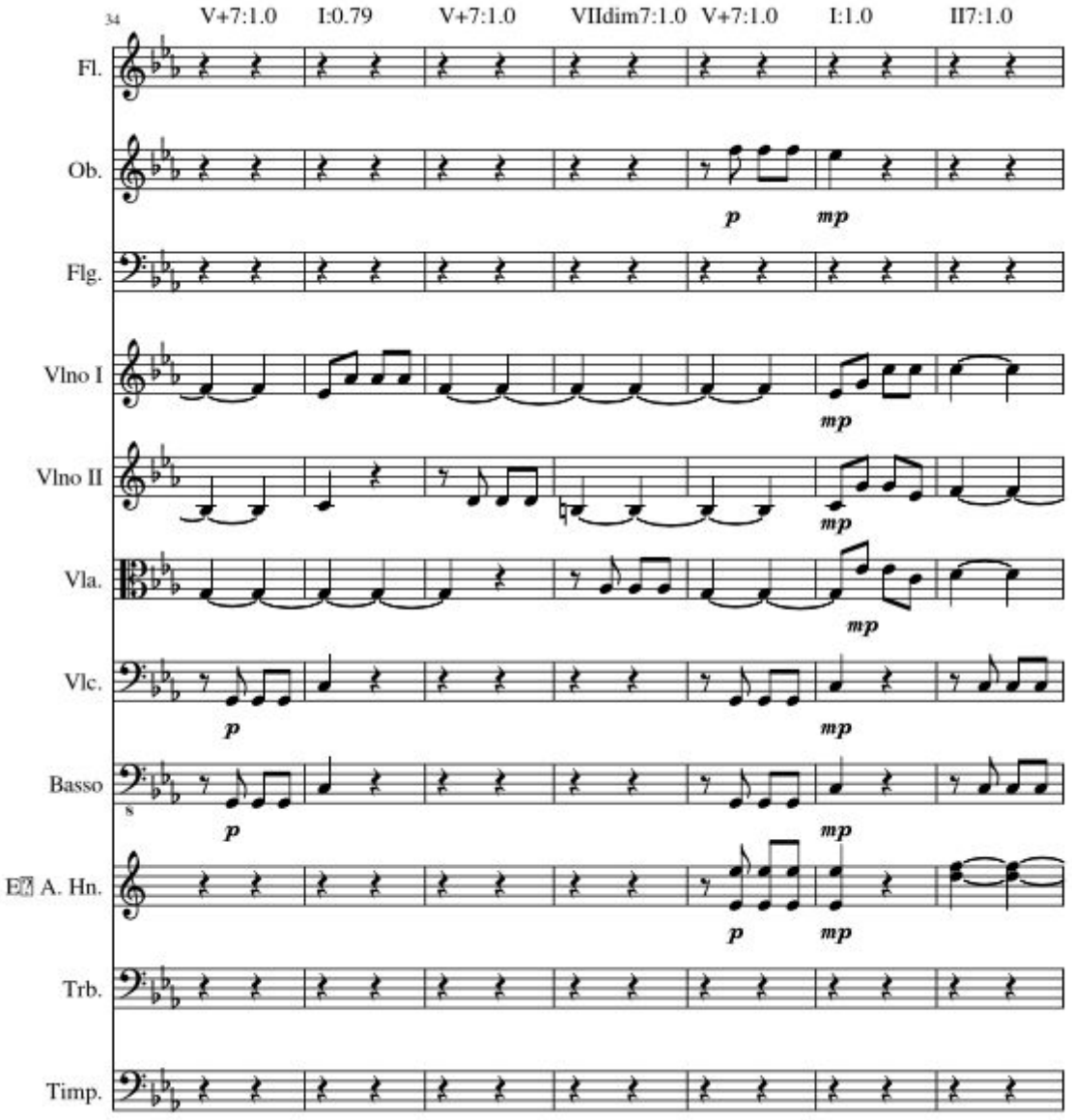
Music21

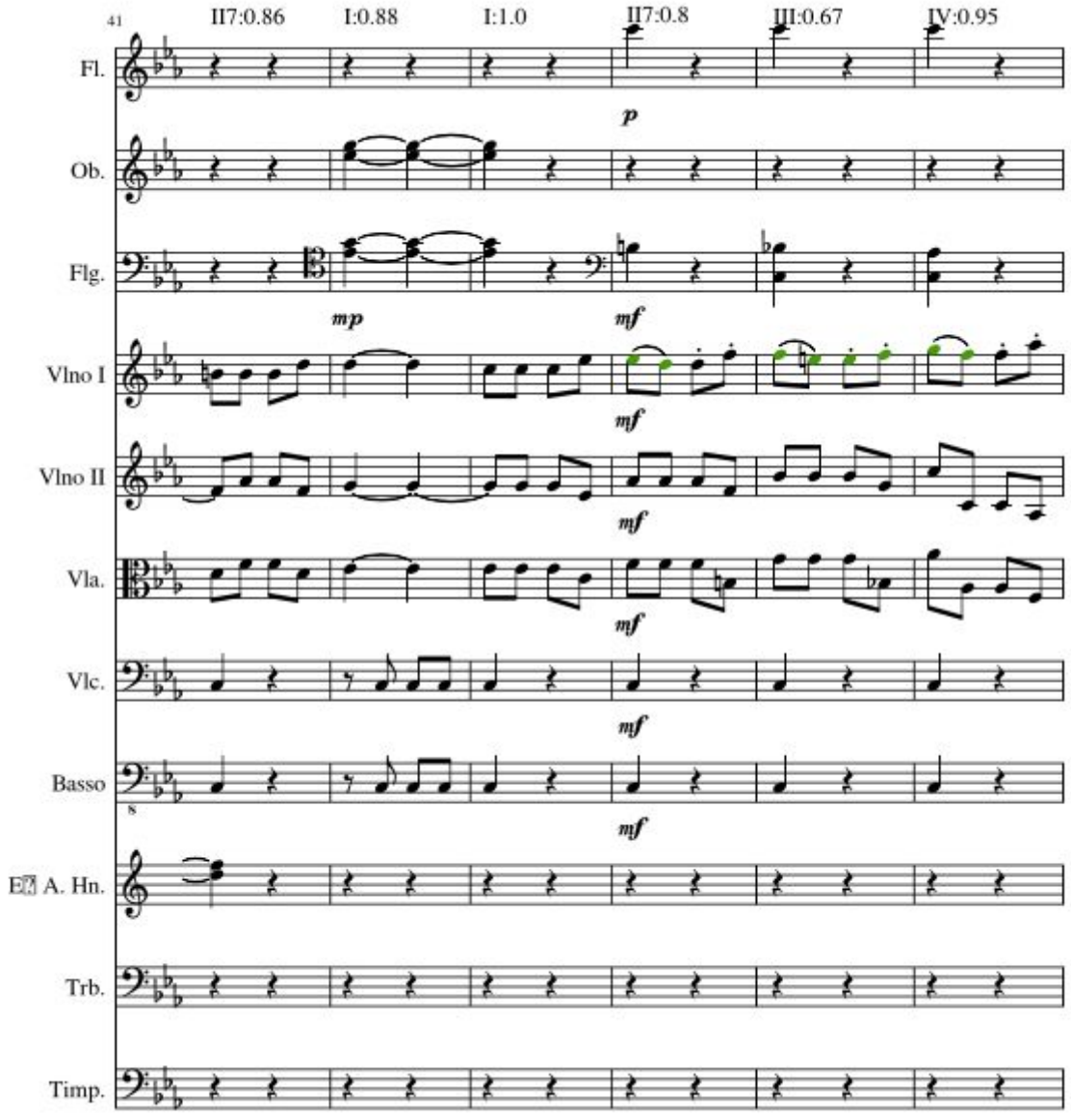


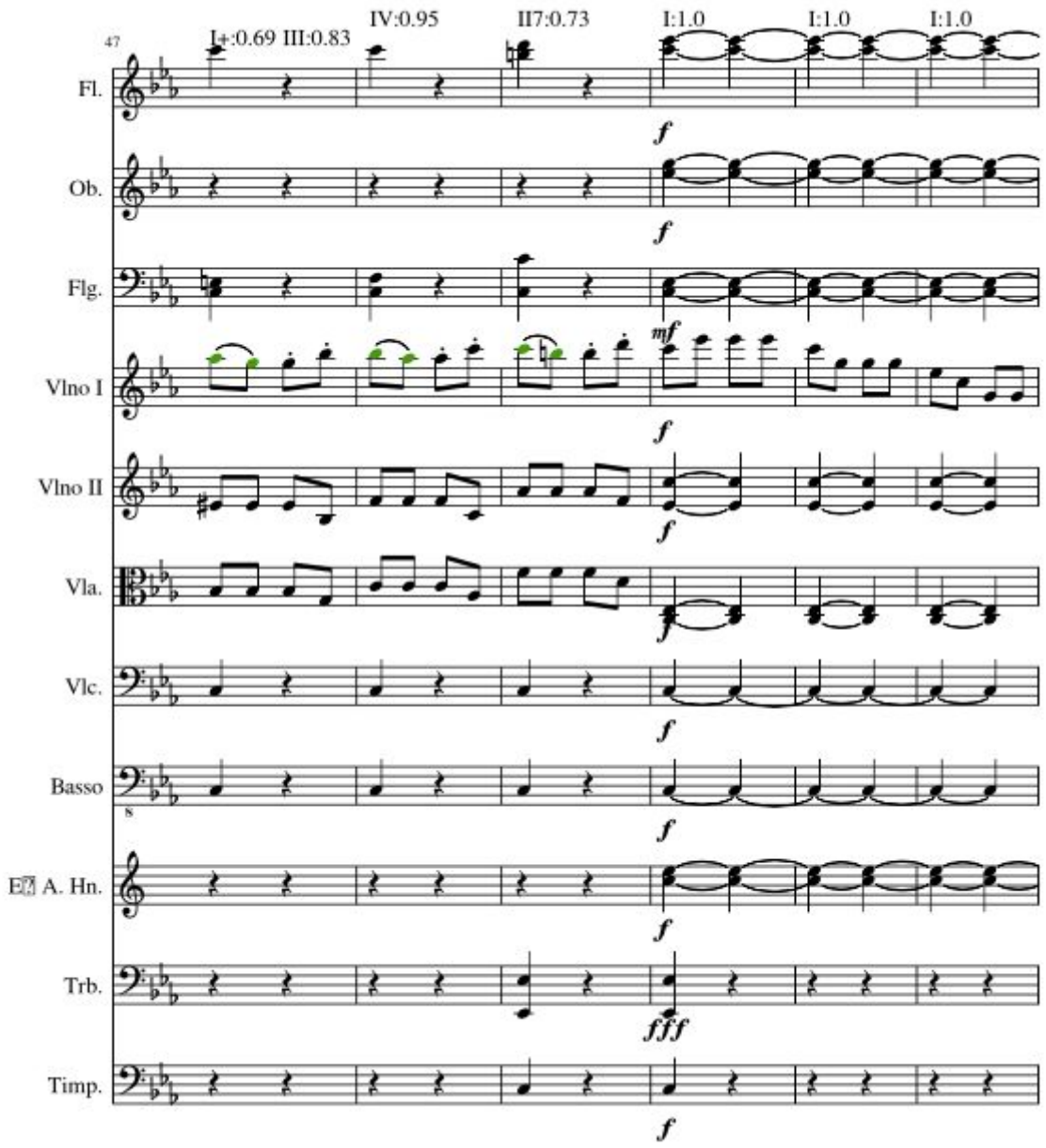

9
VIb:1.0
I:1.0
I:1.0
I:1.0
VIIdim7:1.0
V+7:1.0
V+:1.0
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♭ A. Hn.
Trb.
Timp.
p

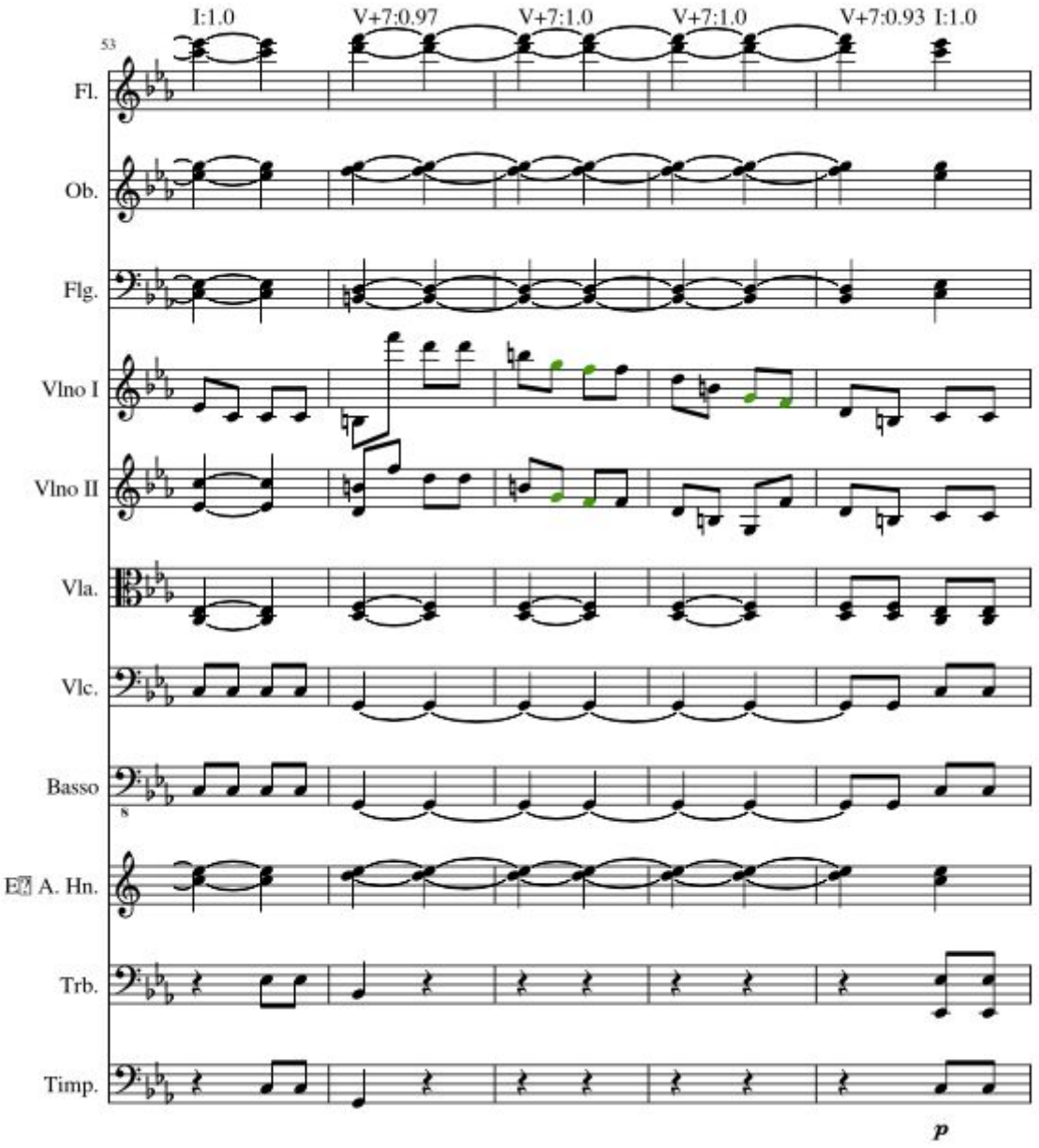

17
V+:0.95
I:0.9
V+:0.95
I:0.9
V+:0.87
I:1.0
VIGer:1.0
V+:0.94
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♵ A. Hn.
Trb.
Timp.
mp
mf
f

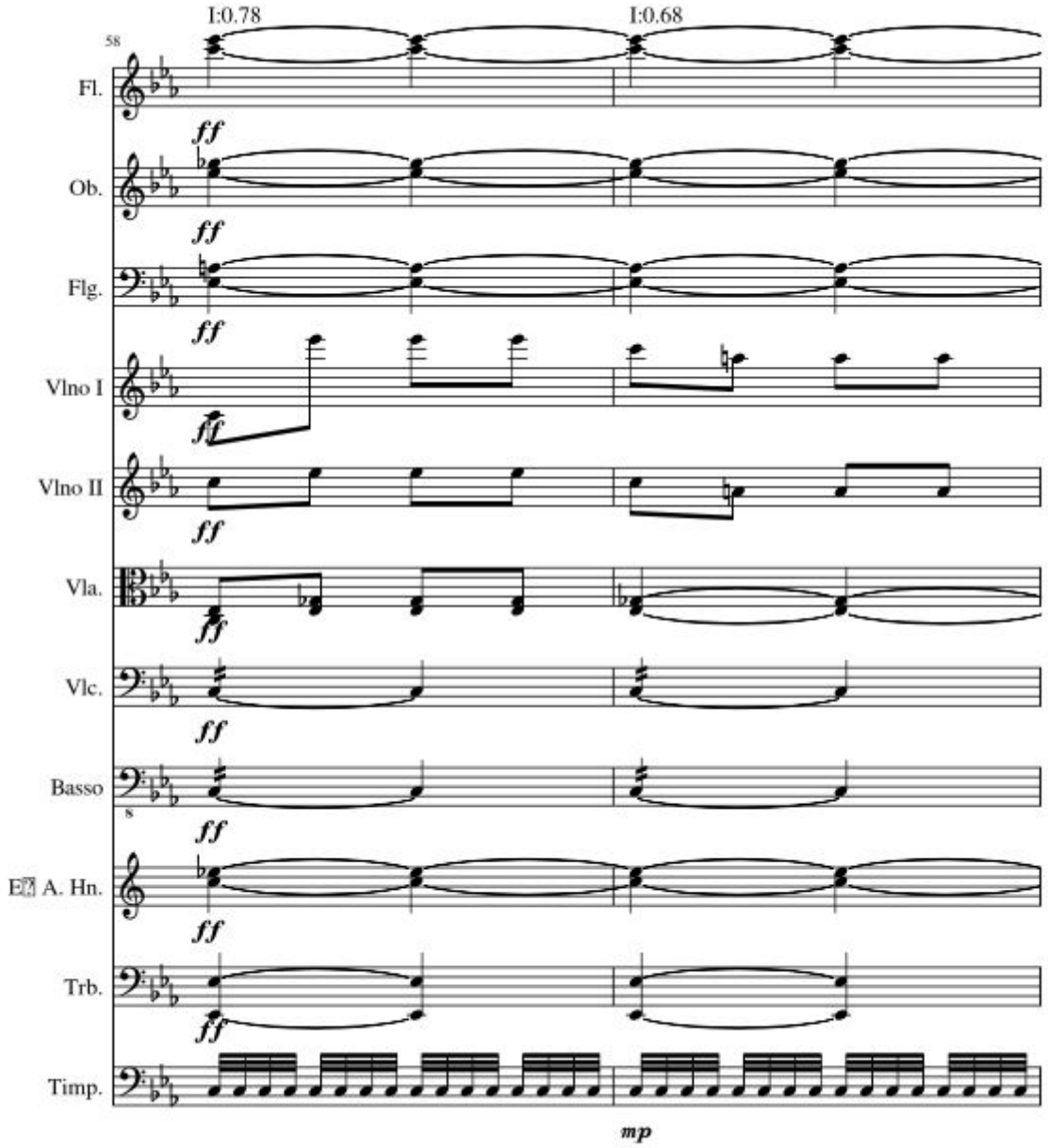

25
II:1.0
VIIdim7:1.0
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
E♭ A. Hn.
Trb.
Timp.
ff
p
fff

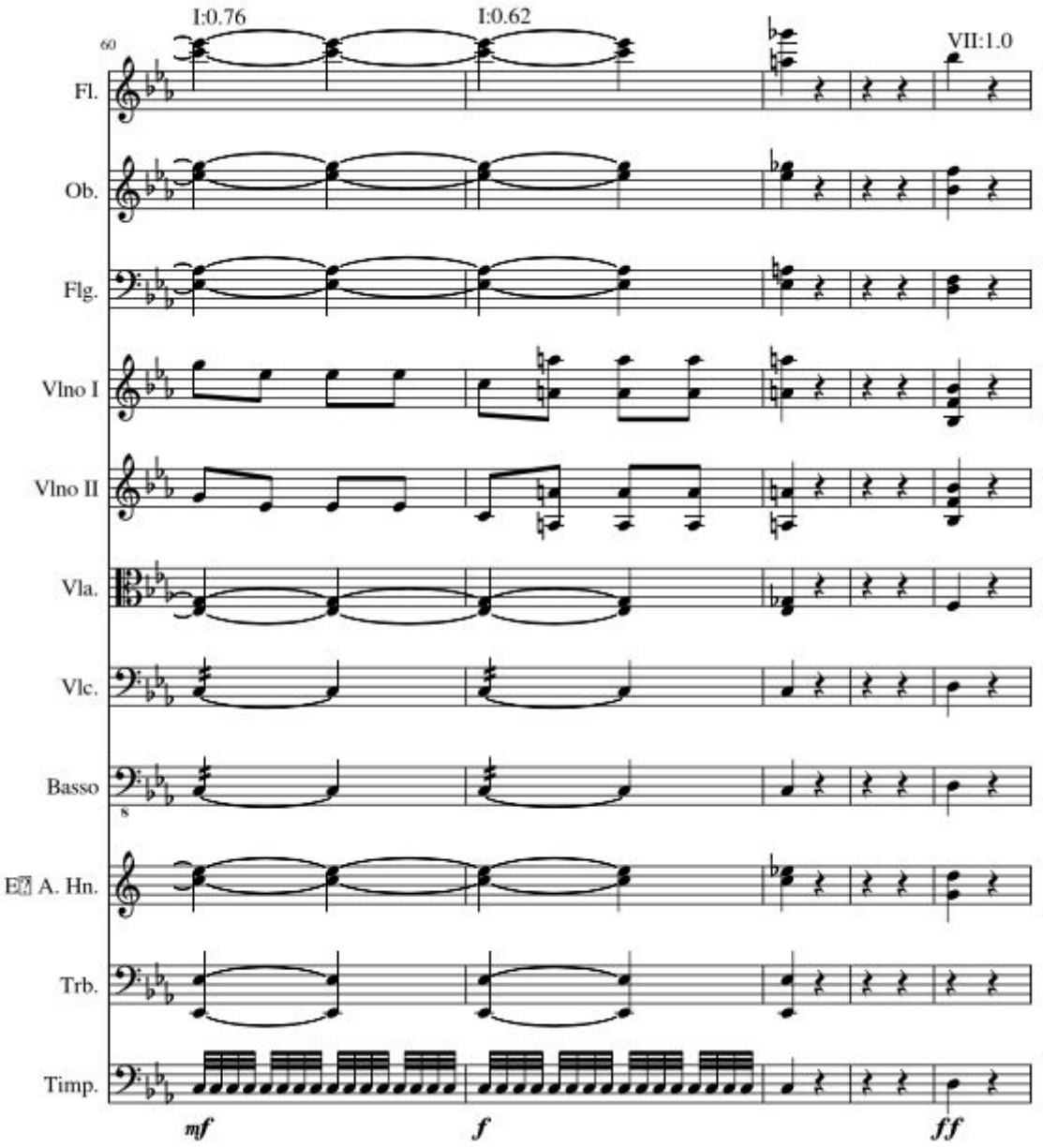

34
V+7:1.0
I:0.79
V+7:1.0
VIIdim7:1.0
V+7:1.0
I:1.0
II7:1.0
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
E♵ A. Hn.
Trb.
Timp.
p
mp


41
II7:0.86
I:0.88
I:1.0
II7:0.8
III:0.67
IV:0.95
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
E♭ A. Hn.
Trb.
Timp.
p
mp
mf


47
I+:0.69 III:0.83
IV:0.95
II7:0.73
I:1.0
I:1.0
I:1.0
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♭ A. Hn.
Trb.
Timp.
f
mf
fff


I:1.0
V+7:0.97
V+7:1.0
V+7:1.0
V+7:0.93 I:1.0
53
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♭ A. Hn.
Trb.
Timp.
p


I:0.78
I:0.68
58
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♭ A. Hn.
Trb.
Timp.
ff
mp


I:0.76
I:0.62
VII:1.0
60
Fl.
Ob.
Flg.
Vlno I
Vlno II
Vla.
Vlc.
Basso
8
E♭ A. Hn.
Trb.
Timp.
mf
f
ff