Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Johan Nicolaisen Brun

Algorithmic Composition of Four-Part Harmony

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

NTNU
Norwegian University of
Science and Technology

Johan Nicolaisen Brun

Algorithmic Composition of Four-Part Harmony

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Contents

Abstract

This thesis seeks to implement a compositional algorithm that generates four-part harmony, including one melody voice and three harmonizing voices. This algorithm uses rules and guidelines from western music theory to implement the generation of each voice and the chords that they create together. Additionally, grouping theory for music is used to create a hierarchical structure to assist with the generation and an abstraction of the emotional context of music is used to help make the music expressive and self-similar. The resulting generated values are converted to a MIDI format and will be playable for the reader of this thesis on Soundcloud. The complete source code written for this thesis is found on GitHub.

Sammendrag

Denne oppgaven forsøker å implementere en komposisjonsalgoritme som genererer firstemt harmoni. Dette inkluderer en melodistemme og tre harmoniserende stemmer. Denne algoritmen bruker regler og retningslinjer fra vestlig musikkteori til å implementere genereringen av hver stemme og akkordene som de lager sammen. I tillegg er gruppeteori brukt til å lage en hierarkisk struktur for å hjelpe med genereringen og an abstraksjon av den emosjonelle konteksten til musikken er brukt til å hjelpe med å gjøre musikker uttrykksfull og selvlik. De resulterende genererte verdiene blir konvertert til et MIDI format og vil være mulige å spille av for leseren av denne oppgaven på Soundcloud. Den fullstendige kildekoden skrevet for denne oppgaven er mulig å finne på GitHub.

1 Introduction

Music is often called the language of emotions [2]. This is reflected in how vital music is in movies, video games and even as parts of everyday social life to establish a common mood that permeates the experience. Music can convey being happy, sad, frightened, excited, and more. This expressiveness can be accomplished through rhythm, melodic contour, harmony and dynamics, amongst others, and parts of performance that are more difficult to quantify.

Through these manners of expression, music can elevate any sensory experience and awaken real emotion in the listener. This thesis explores the possibilities of using an emotional abstraction presented in the article *A Circumplex Model of Affect* [11], in algorithmic composition to compose music that reflects a specific mood and allows for larger scale self-similarity. This emotional context will be combined with grouping theory developed by Fred Lerdahl and Ray Jackendoff and introduced in their book *A Generative Theory of Tonal Music*[8].

The subject of 4-part vocal harmony was chosen for this thesis because of its well-defined rules and recommendations. Because of this, arranging vocal harmony is often used in the early parts of educational programs on music theory as it introduces the student to essential concepts such as voice leading, counterpoint and harmonic progression. An additional personal motivation for the author of this thesis is having spent the entire period of studying for this degree singing classical choir music. This experience has created a deep connection and appreciation for this genre.

The concept of procedural generation is also wildly engaging and exciting. Using chance to create something new every time makes for both a compelling development process and a program that piques the user's imagination. Many recent approaches to music generation based on machine learning require massive datasets and annotations done by experts. Because procedural generation does not have this prerequisite, it allows for faster development of new features. Additionally, because it requires knowledge about the underlying field that the generation is based on, which in this thesis is music theory, it creates a fun and motivating context for learning about this field. Therefore, hopefully, the reader of this thesis will have the opportunity to learn more about how music and harmony work without a background in music.

At the core of this thesis is a software development project. The design and implementation of this project constitute a significant part of the motivation for this thesis. Digitization and automation are major trends in the software development industry, and this project tackles both in a distinctive way. In order to generate music, it must first be digitized. The Pitch and duration of notes, chords, scales, and more must be represented digitally. There already exists functionality for this found in MIDI and python libraries. However, these are only used to support this project's digitization to retain complete control over the generation and processing of information. Although this information is generated, much of it is automatically processed in separate modules. Implementing these will serve as a great exercise in developing well-defined independent modules that automatically process information. Finally, harmony is often generated using an optimization strategy. Extracting the correct meta-information that will allow each voice to be generated independently while remaining coherent can lead to a system that does not require optimization. Implementing this is a difficult task that will be explored in this thesis.

Motivated by all of this, the goals of this thesis are as follows. Use algorithmic composition to compose a section of four-part harmony. Explore the possibilities of generating expressive music that reflects a given emotional context. Generate the hierarchical grouping structure presented in *A Generative Theory of Tonal Music* and apply this structure to algorithmic composition. Explore the use of common meta-information to allow for independent generation of voices. Enforce rules and guidelines based on music theory without using search and optimization strategies.

2 Background

2.1 Four Part Harmony

This chapter will present essential concepts of western tonal music theory to provide the necessary context for discussing the topic of algorithmic composition and the results of this thesis. Western music refers to music as defined by the conventions and standards developed in Europe from the end of the middle ages to today. Tonal music is music that is centered around a key center which will be further expanded upon later in this chapter. It will also present methods, rules and guidelines used when composing four-part harmony. The rules are defined in traditional writing of harmony and are supposed to be followed strictly to achieve a good sounding result and as an exercise in harmonic writing. Guidelines are softer restrictions that can help the composer write harmony that sounds good and is easy to perform, but are not necessary to be followed at all times. These rules and guidelines were established several hundred years ago and have been challenged in many ways, but remain a good starting point [12].

2.1.1 Notes, Keys, Intervals and Chords

A note defines a certain pitch and the duration for which it is held. Each note has a name corresponding to a pitch shown in figure 1. The A in figure 1 corresponds to 440Hz and is the standard tuning frequency used as a basis for finding the frequencies of all other notes [6]. The method used to find the pitch of other notes is called a tuning system. The twelve-tone equal temperament tuning system is the most commonly used in western music. In this tuning system, each pitch is a multiple of $\sqrt[12]{2}$ or $\frac{1}{\sqrt[12]{2}}$ of the previous pitch for higher and lower pitches, respectively. According to this system, starting at one pitch and moving upwards results in twelve distinct notes, each a *half step apart*. When a pitch is multiplied twelve times, the frequency is doubled, and we consider the note to be the same, only in a different octave [10].

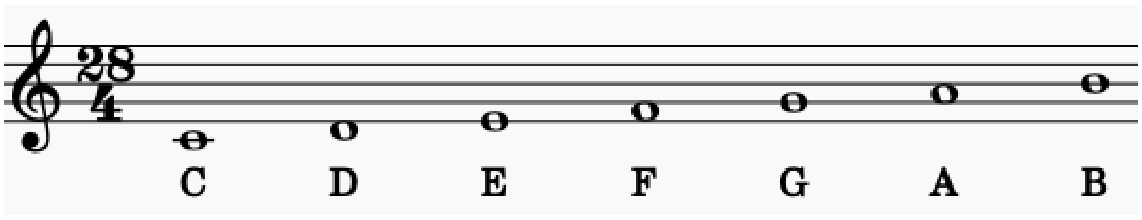


Figure 1: Natural notes and their names

Two half-steps put together is considered a whole-step. By starting at one note and then moving upwards using a sequence of eight whole-steps and half-steps, ending at the octave above, creates a set of notes which defines a scale, also known as a mode. The most well known modes are the major and minor modes, although there are several other modes with distinctive sounds. The major scale can be seen in figure 2. The starting note of the scale defines the key that the mode exists within and acts as the tonal center. This means that starting at the note C and moving upwards according to the major scale until C is reached again defines all notes used in the key of C major. The tonal center refers to both the first note and the corresponding chord and is where the music sounds "at rest". The tonal center

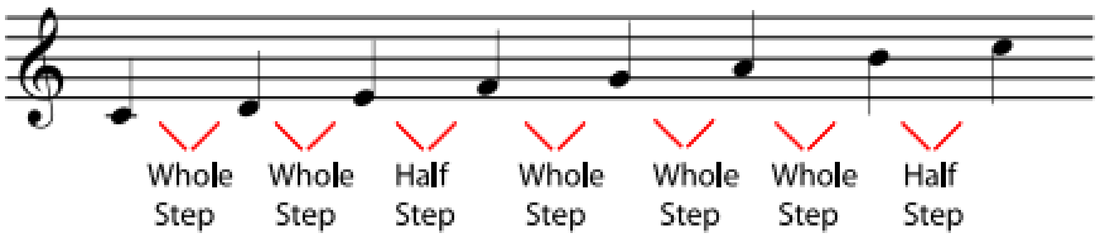


Figure 2: Major scale [12]

Interval	Mathematical ratio	Number of half-steps
Unison	1:1	0
Major second	9:8	2
Major third	5:4	4
Perfect fourth	4:3	5
Perfect fifth	3:2	7
Major sixth	5:3	9
Major seventh	15:8	11
Octave	2:1	12

Table 1: Intervals of the major scale

is also referred to as the first degree of the major scale. Each note of the major scale is assigned a degree from first to seventh and the same applies to the corresponding chords. This is very convenient because the degrees have the same functionality in melodic and harmonic movements regardless of which key they are in [12].

The relationships between the frequency of two notes, also described as the distance between notes, are called intervals. Each interval has a name based on how many half-steps the two notes are apart. For example, the relationship of 2:1 is called an octave. Figure 3 shows the rest of the intervals that have relationships smaller than the octave. Larger intervals than an octave are referred to as compound intervals. They are often described as the combination of an octave and one of the intervals in figure 3. Intervals in their natural form have exact mathematical relationships like the octave, for example, a fifth has the relationship 3:2, and a major third has the relationship 5:4. However, when using twelve-tone equal temperament, these "just" intervals are sacrificed to have all intervals sound the same across all octaves and keys, which is very important in polyphonic music.

Intervals are divided into two groups in two different ways. First, distinguish between perfect intervals and major/minor intervals. The perfect intervals are unison, fourth, fifth, and octave. These are called perfect intervals due to the close relationship between the two notes of the interval, and they can be augmented or diminished by raising or lowering the top note by a half-step. The ratio between the pitches in the perfect intervals illustrates this close relationship, as seen in table 1. The major and minor intervals are the major and minor second, third, sixth and seventh.

The second way to divide intervals is to differentiate between consonant and dissonant intervals. Consonant intervals are at rest and are easy to listen to, while dissonant intervals have tension and may even make the listener uncomfortable. It is possible to see the degree of consonance and dissonance of intervals by looking at the ratios in table 1. Simple ratios lead to consonant intervals and complex ratios lead to dissonant intervals. The consonant intervals include all perfect intervals and the major and minor third and sixth. This leaves the major and minor

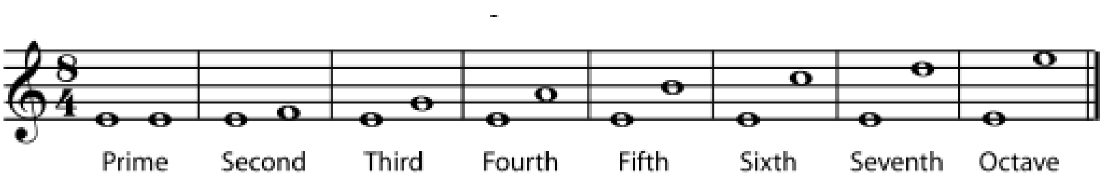


Figure 3: Simple intervals

C	Dm	Em	F	G	Am	Bdim
I	ii	iii	IV	V	vi	vii°

Table 2: Diatonic chords in C major

second and seventh as the dissonant intervals [7].

Chords are a set of three or more notes. Their characteristics come from the intervals between these notes. The most common and basic chords in tonal music are the major and minor triads. A triad is constructed with the root note which names the chord, the note a third above the root and the note one fifth above the root. The third can be either a major of minor third which defines if the chord is major or minor. Building a triad can also be described as stacking two thirds on top of each other. Adding a major third to the root and a minor third to the major third gives a major chord and the opposite order results in a minor chord. By taking each note of the major scale and stacking the thirds following the major scale above it results in a set of diatonic chords that make up the basis for writing chord progression in a major key. These chords are referred to by the degree of their root note using roman numerals. Major and minor chords are distinguished by using upper case and lower case numerals. The seventh chord is called a diminished chord because it has two minor thirds stacked on top of each other. This chord has a particularly dissonant sound and is rarely used in classical harmony [12, p. 185-186].

When the lowest note of a chord is the root note, then that chord is said to be in root position. If the lowest note is the third or fifth on the other hand, the chord is said to be in its first or second inversion respectively. These are slightly more unstable versions of the same chord that have a more temporary feeling. The first inversion is often used for voice leading purposes and creating a flow from chord to chord due to its non-permanent feeling. The second inversion however is much more rarely used [15, p. 102-123].

2.1.2 Writing for Vocals

When writing music for vocals, some considerations should be made due to the limits of the human voice and the differences between male and female voices. Women are divided into sopranos and altos for higher and lower voices. Men are similarly divided into tenors and basses. Practiced singers can usually master two octaves within the ranges shown in figure 4. While there is overlap between the male and female ranges, it is essential to consider that the same note will sound very different when sung by a man or a woman. The note in figure 5 will sound dark and relaxed and will be difficult to add power to when sung by a woman. However, when a man sings it, it will be bright and tense and while it is possible to sing it with full power, singing it at a low volume requires good technique. With this in mind, notes should generally be placed an octave apart for a man and a woman to sing them with the same characteristics.

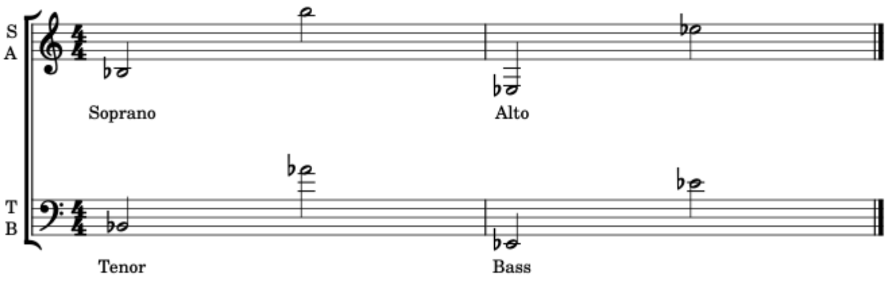
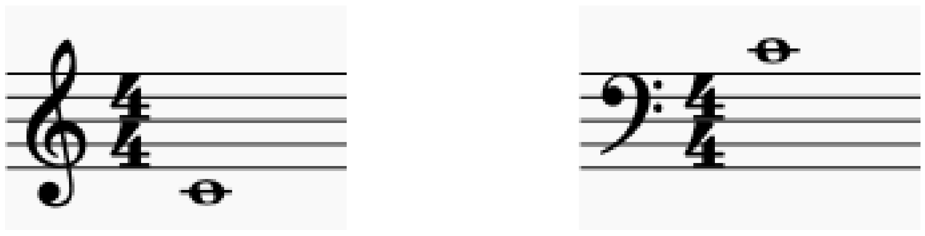


Figure 4: Vocal ranges



(a)

(b)

Figure 5: The same note sung by a female voice and a male voice

The same qualities apply to chords as well. Singing a chord in a lower part of a range will sound dark and soft, while it will sound bright and strong in a higher part of a vocal range. As a chord moves into the lower part of the male vocal range, it will start to sound muddy and unclear, especially if the pitches of the harmony are close together. Because of this, we want to place notes at a larger distance in the male voices and closer as the pitch gets higher. This distribution of voices reflects the harmonic series shown in figure 6 [14].

It is also important to strive toward good voice leading. This means writing melodies for all voices that are easy to sing. To accomplish this a composer will try to use a majority of stepwise movements, repeat notes and avoid leaps larger than a fifth expect for the octave. [15, p. 22-26].

2.1.3 Counterpoint

This thesis seeks to generate harmony, which to a large extent revolves around how chords are connected to each other and move through time. Harmony also deal with how the voices

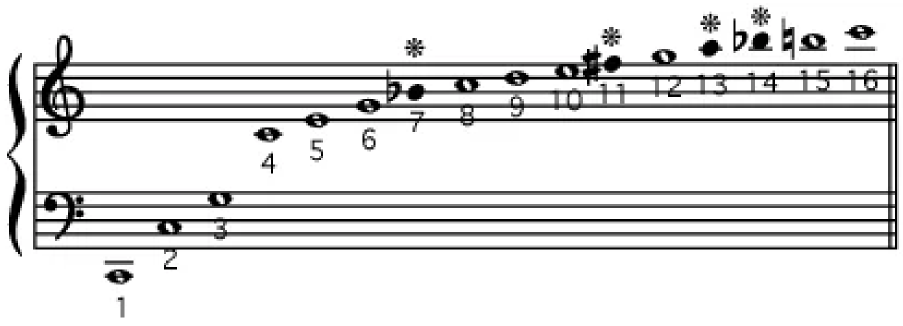


Figure 6: Harmonic series, courtesy of Encyclopedia Britannica

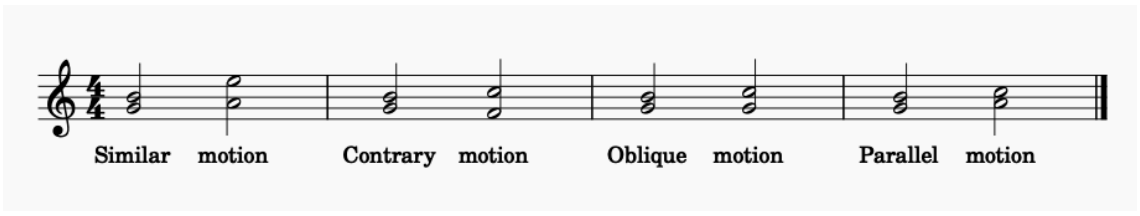


Figure 7: Four types of motion

building these chords move together. When two voices move together, independently of each other, it is called counterpoint. Writing counterpoint usually consists of taking a given melody, called a cantus firmus, and writing a second melody called the counterpoint. The counterpoint should ideally work both above and below the given melody. The traditional forms of counterpoint are called counterpoint species. Relevant to this thesis are first-species counterpoint and second-species counterpoint.

First-species counterpoint deals with a counterpoint that has one note for each note in the cantus firmus, with the same duration, giving two melodies that occur simultaneously. When multiple voices move with the same rhythm, that music is considered homophonic. Second-species counterpoint has a counterpoint melody that moves twice as fast as the cantus firmus giving a two to one rhythmic relationship. When there are varying rhythmic relationships between voices, that music is considered polyphonic.

There are four types of movements that a voice can perform in relation to the melody: similar motion, contrary motion, oblique motion and parallel motion. Similar motion occurs when the melody and second voice move in the same direction, but by different amounts. Contrary motion occurs when the melody and second voice move in opposite directions. Oblique motion occurs when one voice remains stationary while the other moves away from it. Parallel motion occurs when both voices move in the same direction by the same amount. Parallel motion is generally not wanted as it imposes a high degree of dependence between voices which is the opposite of the goal of counterpoint. In addition there are some parallel movements that are considered illegal in traditional 4-part harmony. Those parallels being parallel fifths, octaves and unisons because of their very strong consonance making the two voices sounding almost like one part [7].

2.1.4 Functional Harmony

While early harmonic music used simple chords that seemingly stood independent of each other without a particular sequence in mind, from around 1650, the concept of functional harmony began being used. In functional harmony, the chords start becoming more complex and have particular relationships with each other, each chord having a limited option of chords that could follow it. In today's modern genres, such as jazz and rock, the rules of functional harmony are not considered as much. The methods of functional harmony presented in this thesis are referred to today as classical harmony.

Primary Chord Progressions

There are three categories of primary chord progressions, each is named after the movement of the root note from the first to the second chord. The categories are *falling fifth/rising fourth*, *rising second* and *falling third*. The chord progressions are shown in table 3. The most common among these are those moving to or from I, IV and V, some even having their own name, for example V - I which is called an authentic cadence [15, p. 95-97].

Falling Fifth	Rising Second	Falling Third
I - IV	I - ii	I - VI
ii - V	ii - iii	iii - I
iii - VI	iii - IV	IV - ii
V - I	IV - V	V - iii
vi - ii	V - vi	vi - IV

Table 3: Primary chord progressions

Secondary Chord Progressions

Less common but also used are the secondary chord progressions, which are the opposite movements of the primary chord progressions, *rising fifth*, *falling second*, and *rising third*. These are seen in static progressions where the chord moves according to a primary or a secondary progression and then immediately back with the corresponding progression. Otherwise are secondary chord progressions usually used when moving from the tonic chord, as the tonic chord is a resting point without much tension giving more freedom to move in any direction. On the other hand, moving to the tonic chord with a secondary chord progression is much less common and almost exclusively done with the IV - I progression, also known as the plagal cadence [15, p. 98-101].

2.2 Music Generation

The type of music generation discussed and implemented in this thesis is algorithmic composition. This is defined as procedurally generating or composing music using algorithmic methods and rules based in music theory. The music theory was presented in the previous chapter and this chapter will present some theories for generalizing music theory to a mechanical form, relevant methods of music generation and previous work in this field.

2.2.1 History of Algorithmic Composition

The Musikalisches Würfelspiel or Musical Dice Game attributed to Mozart is often cited as an early example of algorithmic composition. In this game, players assembled precomposed snippets of a waltz according to the thrown dice. This method of randomly forming a piece of music is an example of stochastic music generation [1][13, p. 10]. However, the greek mathematician Pythagoras was already laying the groundwork in around 500 B.C. by documenting the relationship between music and mathematics. The greek studied music, arithmetic, geometry, and astronomy, and left many treaties on music theory. Yet it is not known if they used mathematical models in the composition of music. Another significant development came to be around 1000 A.D. when the framework for our conventional system of music notation was established. Guido Arezzo (around 991–1031) was a music theorist who, amongst others, contributed considerably to developing this framework. His notation allowed other musicians than the composer to perform the music, formalising a composition to a larger extent which previously had been to an extensive degree improvised. This development also started to increase the distinction between composer and performer [9, p. 21][13, p. 7].

What we think of today as algorithmic composition is usually computer-aided algorithmic composition. In 1957 Lejaren Hiller and Leonard Isaacson were able to successfully program the ILLIAC computer to create the first completely computer-generated composition[13][1]. The music was generated on a symbolic level, meaning that the output of the system, representing note values, had to be interpreted by a musician. The *Illiac Suite* scored for string quartet consisted of four movements or "experiments", each dealing with a musical task. Experiments 1 and 2 modeled two different sets of counterpoint techniques experiment 3 used a non-restricted ruleset to create experimental music and experiment 4 used markov models of varying orders [9].

2.2.2 A Framework for Comparing Methods

Rene Wooller et al. present a framework for describing and comparing different algorithmic composition methods in their 2005 paper *A framework for comparison of process in algorithmic music systems*. This framework places methods on a two-axis spectrum based on the method's function and breadth of context. The function of the method is divided into three categories: analytic, transformational and generative. A simple way to describe the difference between these algorithms is by looking at the input and output. Analytical algorithms extract specific features from a set of data and thus tend to reduce the data size from input to output. For example, any method that processes a dataset with music within a single genre to replicate that genre's characteristics. Transformational algorithms rearrange and alter the available input, so the size of its output will be similar. Generative algorithms increase the size by generating more music from their input. The other criterion, breadth of context, depends on how much a single piece of the music depends on other parts of the score. An example of low breadth would be a melody where each note is only generated based on the previous one, which appears in stochastic methods. In contrast, a higher breadth entails that generating each note is dependent on whole phrases or even the entire score. An example of a method with a high breadth of context is generative grammars [16].

2.2.3 Generative Grammars

Generative grammars, originating in the basic linguistic model developed by Chomsky in 1957, contains methods for analyzing and algorithmically composing music. The work in this field by, amongst others Lerdahl and Jackendoff led to their book *A Generative Theory of Tonal Music* first issued in 1983, which will be further looked into in a later section. Some of the more notable uses of generative grammars for musical production and analysis include the following. In jazz, this formalism is used for generating chord progressions based on jazz harmony rules and musical corpora. Music ethnology aims to use grammatical models to describe different music styles. In European art music, this is used in analysis and generation through hierarchical grammatical structures.[9]

The concepts best described by generative grammars are the ones that allow a hierarchical division of the musical material and work with the substitutions of symbols. This includes both knowledge-based and non-knowledge-based methods. The first are methods where rules are explicitly formulated, and the latter are methods where a corpus is processed, and rewriting rules are automatically generated based on the corpus.[9]

The Chomsky hierarchy of grammars is divided in four levels grammars with increasing levers of restrictions and decreasing levels of generative capacity.

- Type 0 grammar (unrestricted grammar)
- Type 1 grammar (context-sensitive grammar)
- Type 2 grammar (context-free grammar)
- Type 3 grammar (regular grammar)

When it comes to hierarchical and context-related organization of musical material for algorthimically composing sequential structures, generative grammars are an essential class of formalisms. However its weakness shows in the desciption of simultanious structures such as harmony.

2.2.4 Grouping Theory for Tonal Music

A Generative Theory of Tonal Music (GTTM) is a book by Fred Lerdahl and Ray Jackendoff that attempts to create a theory of music that can also be described as a general framework for analyzing tonal music. Its goal is: "to be a formal description of the musical intuitions of an experienced listener" [8, p. 1]. As music is a product of human activity, Lerdahl and Jackendoff argue that a piece of music is a mentally constructed entity. Therefore music theory exists among traditional areas of cognitive psychology, such as theories of vision and language. It is important to note that the theory of music presented in this study is limited to homophonic music and does not claim to be a complete description of polyphonic music, especially for contrapuntal varieties of tonal music. The complete analysis framework presented in this book consists of multiple parts. However, in this thesis, only the theory on grouping structure will be presented and used to develop the generation algorithm.

Grouping is a natural way for listeners to perceive music, similar to visual perception. Grouping in GTTM occurs according to two sets of rules. Five wellformedness rules define how groups are put together hierarchically and must be satisfied by all possible grouping structures. Six preference rules define how pitches and rhythms should be grouped to reflect a listener's intuition and are not required to be fulfilled at all times. The book presents these rules as follows [8, p. 36-51].

Grouping Well-formedness Rules

GWFR 1 Any contiguous sequence of pitch-events, drum beats, or the like can constitute a group, and only contiguous sequences can constitute a group.

GWFR 2 A piece constitutes a group.

GWFR 3 A group may contain smaller groups.

GWFR 4 If a group G1 contains part of a group G2, it must contain all of G2.

GWFR 5 If a group G1 contains a smaller group G2, then G1 must be exhaustively partitioned into smaller groups.

Grouping Preference Rules

GPR 1 Strongly avoid groups containing a single event.

GPR 2 (Proximity) Consider a sequence of four notes n1n2n3n4. All else being equal, the transition n2-n3 may be heard as a group boundary if

1. (Slur/Rest) the interval of time from the end of n2 to the beginning of n3 is greater than that from the end of n1 to the beginning of n2 and that from the end of n3 to the beginning of n4 or if
2. (Attack-Point) the interval of time between the attack points of n2 and n3 is greater than that between the attack points of n1 and n2 and that between the attack points of n3 and n4.

GPR 3 (Change) Consider a squence of four notes n1n2n3n4. All else being equal, the transition n2-n3 may be heard as a group boundary if

1. (Register) the transition n2-n3 involves a greater intervallic distance than both n1-n2 and n3-n4, or if
2. (Dynamics) the transition n2-n3 involves a change in dynamics and n1-n2 and n3-n4 do not, or if
3. (Articulation) the transition n2-n3 involves a change in articulation and n1-n2 and n3-n4 do not, or if
4. (Length) n2 and n3 are of different lengths and both pairs n1n2 and n3n4 do not differ in length.

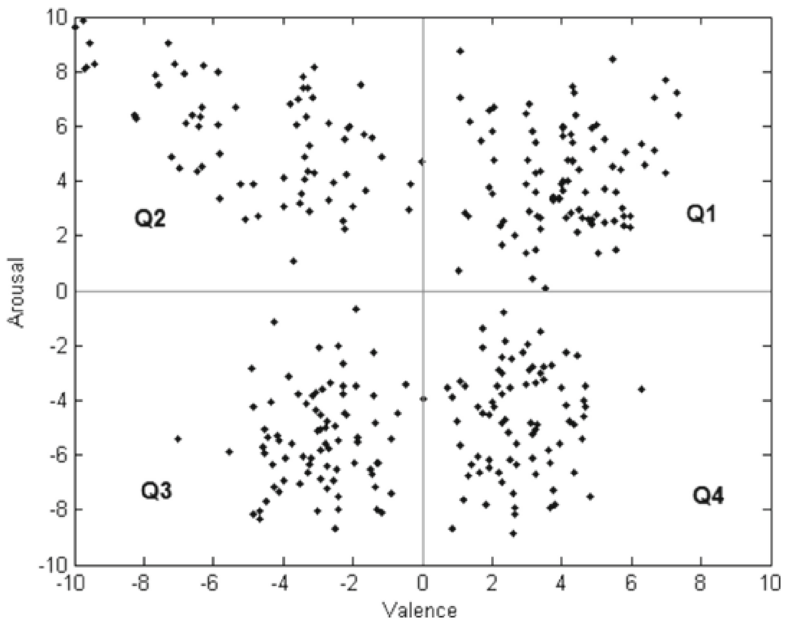


Figure 8: Distribution of the data set [3]

GPR 4 (Intensification) Where the effects picked out by GPRs 2 and 3 are relatively more pronounced, a larger-level group boundary may be placed.

GPR 5 (Symmetry) Prefer grouping analyses that most closely approach the ideal subdivision of groups into two parts of equal length.

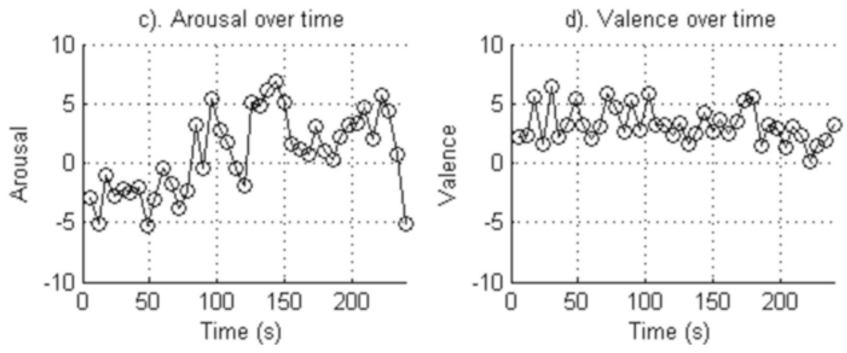
GPR 6 (Parallelism) Where two or more segments of the music can be construed as parallel, they preferably form parallel parts of groups.

In 2007, the article *Implementing "A Generative Theory of Tonal Music"* was published. In this article the automatic time-span tree analyser (ATTA), a musical analysis system, was presented. ATTA automates musical analysis based on GTTM which at the time was done manually by musical experts. The resulting system builds among others, a grouping structure for a given melody [4].

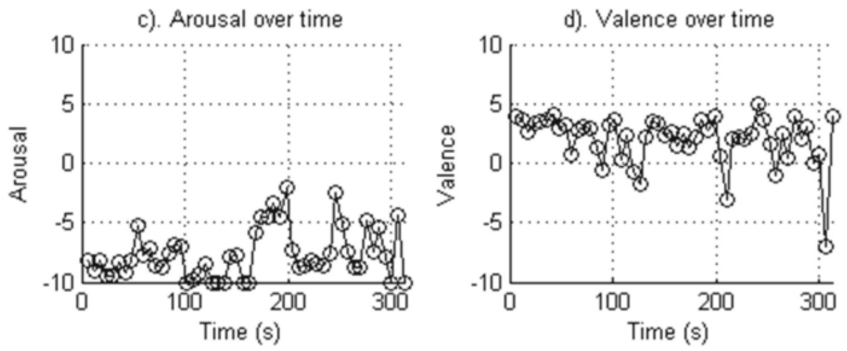
2.2.5 Emotional Mapping of Music

In the 2016 paper *Music Emotion Maps in Arousal-Valence Space* Jacek Grekow presents an approach to detecting emotion in music using regression methods. The music is to be mapped to the 2D space of valence and arousal presented by Russell in the 1980 paper *A Circumplex Model of Affect* [11]. In this project, a database of segments of music is built by having five music experts with university musical education individually annotate 324 6-second segments of music with values of valence and arousal in the range of -10 to 10. These values were averaged, and the resulting distribution was almost uniform across the four quadrants of the VA-space as shown in figure 8. These segments are used to train a regression model to predict the emotional state throughout pieces of music.

The best-obtained regression models for predicting valence and arousal were then used to analyze pieces of music. The pieces were similarly divided into 6-second segments with 3/4 overlap. The results were presented in four different ways:



(a) A-V over time maps for the song let it be by Paul McCartney (The Beatles) [3]



(b) A-V over time maps for Piano Sonata No. 8 in C minor, Op. 13 (Pathetique), part 2, by Ludwig van Beethoven [3]

1. Arousal-Valence over time;
2. Arousal-Valence map;
3. Arousal over time;
4. Valence over time.

These maps show how the values of valence and arousal change throughout the compositions. It clearly shows how different parts of songs have different affective states and how a song can gradually increase or decrease in valence or arousal during a song. Interestingly the maps shown for the two compositions in the paper show that while arousal increases gradually in mean and or variance, arousal behaves more as diverging from a mean and then returning to it [3].

2.3 Software Development Tools

2.3.1 Python

The project in this thesis was developed using Python version 3.10. Python was chosen because of its flexibility and simplicity in implementation, and because of the authors familiarity with it. Python is a programming language that does not enforce typing of variables, parameters, return types, ect. However, as of version 3.5 python supports type hinting which is highly advantageous. Typing is an important part of error prevention in software development which is reflected in the appearane of TypeScript, an alternative to JavaScript with strict type enforcing[1]. Typing also helps the IDE with recognizing types across functions and classes which helps prevent syntax errors. Python was also chosen because it has the option for object oriented programming. This approach is very useful when developing a system with a hierarchical structure like in this thesis[2].

2.3.2 Midi and MuseScore

Midi is a language, or protocol, for instructing a device or program how to create, playback or alter sounds. It is not something that creates sound on its own, but rather allows for communication between devises that create sound. [5]

MuseScore is a free, open source, professional music notation software. MuseScore can use fileformats such as MusicXML and MIDI to transfer notation between programs. It is a great tool for listening to and reading the scores generated in this thesis, as well as altering them or adding notes to examine how potential results in the future should be. Additionally it has been used to create some of the figures used in this thesis. [3]

2.3.3 Music21

In order to produce midi files from the values generated in this project, a library for producing midi files was needed. The library chosen was music21. Music21 is a toolkit for symbolic analysis and generation of music and is for the most part developed at the MIT section for Music and Theater Arts. [4]

Music 21 has dedicated objects for the many components that make up a score. A note can be created and shown easily. Music21 objects can be output as MusicXML which is a file format created to share musical scores across programs. This allows the notation written using music21 to be shown in MuseScore.

```python
n = note.Note('C#3')
n.duration.type = 'half'
n.show()
```

[1]https://www.typescriptlang.org/
[2]https://www.python.org/about/
[3]https://musescore.org/nb
[4]https://web.mit.edu/music21/doc/index.html

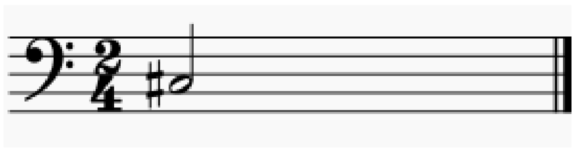


Figure 10: C♯ shown in MuseScore

The Note object can also be initialized with the standard midi values for notes in the range 0 - 127, therefore the string "C#3" could be replaced with the integer 49 to achieve the same result. Chords can be built similarly.

```
c = chord.Chord(['G4', 'B-4', 'D5'])
c.duration.type = 'half'
c.show()
```

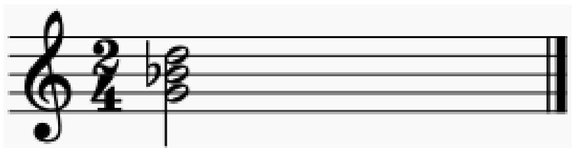


Figure 11: G minor chord shown in MuseScore

Music21 uses a stream-object to collect notes and chords into the same score. Objects can be added to the stream at initialization or appended afterwards. Streams can also be added within other streams either sequentially in a single voice, or simultaneously to create multiple voices.

```
tenorStream  = stream.Stream(note.Note('G3'))
tenorStream.insert(clef.Treble8vbClef())
bassStream = stream.Stream(note.Note('E3'))
bassStream.insert(clef.BassClef())
s = stream.Stream([tenorStream, bassStream])
s.show()
```

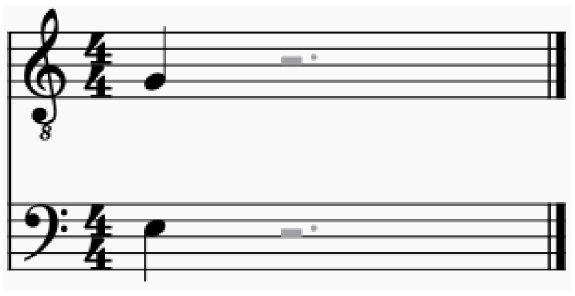


Figure 12: Two nested stream used to create a score with two voices

The clef object used in the code above is an example of the large amount of functionality for writing scores that is available through music21. The library also supports setting different tempos, chosing different instruments for streams, adding spanners and expressions to notes and much more. [5]

[5] https://web.mit.edu/music21/doc/moduleReference/index.html

3 System Design

Considering the background presented in the previous chapter, a system that can accomplish the generation goals of this thesis can be designed. This chapter presents the method and resulting code of the development process. First a set of specifications for what is required of the system is laid out. Then the following section presents the decisions on how such a system can be designed. Finally the implementation of this system is explained and the most relevant source code is shown.

3.1 Spec

This chapter will outline the requirements for the generation algorithms and the midi files produced by them. It is therefore divided into two parts. The first part will present the musical goals for the resulting midi files, based in the theory presented in chapter 2.1. The second part will present the functional requirements of the algorithm to accomplish these goals. Using the framework presented in chapter 2.2.2 we can describe the sought after algorithm as generative and with a medium to high breadth of context.

3.1.1 Specification For the Generated Music

The melody generated by the system should have a clear melodic contour. It can include non chord tones, but the start and end of phrases and following large leaps should use chord tones. The melody should also exhibit a degree of self similarity.

Each chord should include at least a root note, a third and a fifth. Chords can be in root position and first inversion which implies that the bass voice will consist of root notes and thirds. The sequence of chords used should follow progressions in functional harmony.

Each voice should be easy to sing, and therefore include repeat notes, a majority of step-wise movements and no leaps larger than a fifth. When voices are moving similarly, they should avoid moving in parallel fifths, octaves and unisons.

While the melody may contain durations from eights notes to whole notes, the shortest duration of the harmonizing voices should be quarter notes.

Additional limitations

- Each composition will be in 4/4 time.
- Each composition will be in C major.
- Each composition will be in 80 beats per minute.
- Each voice will be built using only diatonic notes.
- Each voice should stay within its respective range.

3.1.2 Functional Requirements

The system developed as part of this thesis requires the following in order to generate 4-part harmony.

Generating grouping. The system should generate a hierarchical structure that can be used to separate parts of the music into naturally divided segments based on the grouping theory presented in chapter 2.2. In order to produce self similarity horizontally and establish a context for each voice to work together, information for the meta model should be generated for each group.

When generating the voices, this system will take a melody-first approach and then follow by adding harmony to the melody. This may allow it not only to generate a melody, but to also take a melody as input and add harmony to it. Therefore the melody generation will use the grouping structure generated previously and populate it with a melody while adding relevant information to the grouping structure that can be used when adding harmony such as the chords that can be used together with the current note. The algorithm should select a chord from the set made available after generating the melody. This selection process should be should be based on functional harmony and reflecting the current values for valence and arousal.

When generating the harmonic voices, the algorithm will use the grouping structure and the meta model generated with the grouping and melody to generate each voice. Each voice should have its own grouping object that can be populated with notes that are selected so that the voices corresponds with the others rhythmically and that the current chord is realized. Each voice including the melody should be in the same format so that they can be used by the same function to build the midi file.

By taking the information stored for each voice a corresponding music21 stream should be produced so that all four can be combined to produce a midi file and score.

3.2 Design

The system developed in this project will be divided into one main module and four submodules. The main module will direct the flow of information between submodules which will each have a task related to generating the final composition.

3.2.1 Grouping Module

The grouping module aims to build a hierarchical grouping structure that can be populated by notes and contain information used to align all four voices together. This goal can be achieved by building a tree using classes for each level of grouping so that each level can contain the information relevant at that level of abstraction. The size of the composed piece of music determines how many levels of abstraction are needed. Currently, the goal is to compose a section of a piece. A section can be divided into multiple phrases and a phrase can be divided into subgroups that for example contain a rising and falling motion. The subgroups are finally divided into base groups. This makes up a four-level hierarchy. The lowest level base groups contain a set of notes that naturally fit together according to grouping rules presented in chapter 2.2.4. The phrase groups contain information about the melody's contour, valence

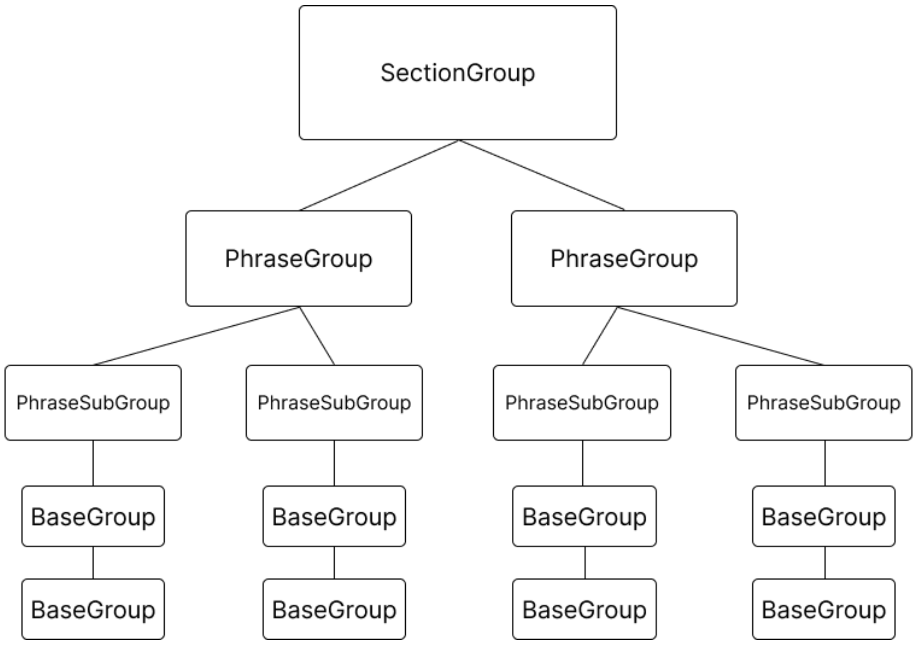


Figure 13: Example of a possible grouping structure

and arousal, which is divided amongst the phrase subgroups. This is in turn distributed down to the base groups. Multiple phrases make up a section of music which is currently the highest level of abstraction in this project. The resulting hierarchy is shown in figure 13.

3.2.2 Melody Module

The melody module populates each base group with notes. For each note a set of chords are selected that fit together with it. With these two goals in mind there are many possible approaches.

This thesis' goal is to explore the possibilities of using a generated set of values to generate the melody. Therefore, each base group will use the following:

- A direction that defines part of the overall contour of the phrase.
- A value of arousal that will be used to find
- The number of units within the group (notes or rests)
- The number of beats the units are divided amongst
- A value of valence that will be used to find
- The amount of minor and major chords available to the group.

Decisions that are taking place within this structure will be done using a uniform distribution so that the results will reflect how well the structure is built rather than a well tuned stochastic method.

The melody will be generated by combining pre-made blocks of movements. Choosing between step-wise movement and counter-step movement. These will move upwards or downwards

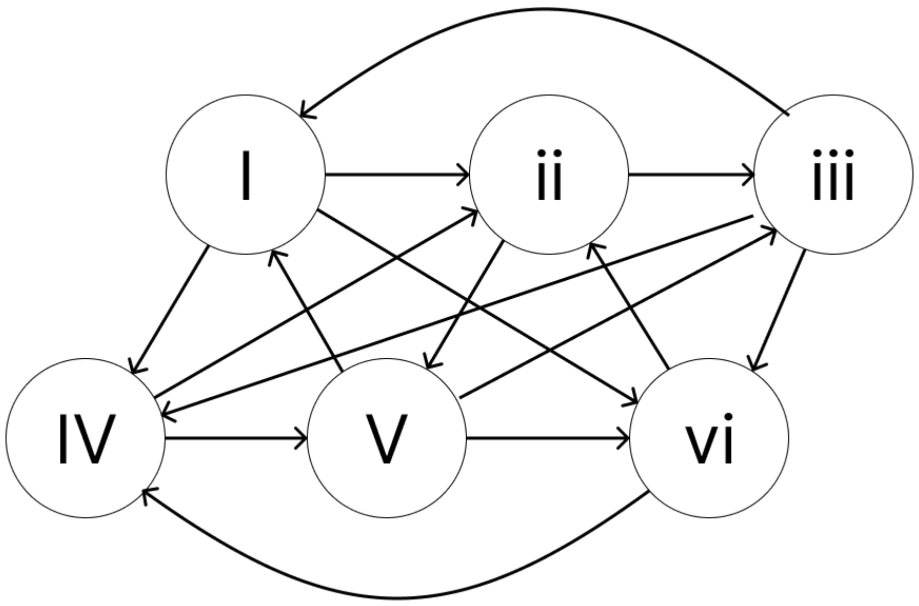


Figure 14: Directed graph for primary chord progressions

according to the direction of the contour which will be flipped if the melody approaches the end of the soprano voice range.

Using the value for valence, the module will choose which chords are available in each base group. Each base group will always have access to the primary triads, I, IV and V because these are the most important and most used triads and because collectively they use every degree of the major scale. The lower the value of valence is, the more secondary triads will be added to the available chord for that base group.

Choosing chords will be done using a directed graph of the primary progressions. When choosing the next chord the algorithm will only include the edges to available chords and choose randomly amongst these or to repeat the current chord. If the only available possible progression is to repeat the current chord, or if there are no possible primary progressions, then the algorithm will chose randomly from the secondary progressions. Because the secondary progressions are the opposite movements of the primary progressions, this is equivalent to reversing the directions of the edges of the graph for one iteration.

3.2.3 Harmony module

Using the information generated by the grouping module and the melody module, the harmony module seeks to realize the chords chosen by the melody module while staying within the many conditions of voice leading and harmonic writing. The chords will be built using the three harmonizing voices bass, tenor and alto. These voices will have relationships with the melody as described in chapter. 2.1.3

This module will treat the voices differently because the bass voice has different constraints placed upon it. Due to all chords being in their root position or first inversion, the bass voice will be forced to only use the root note or third of each chord. This is in contrast to the soprano, alto and tenor voices which are also allowed to use fifths and notes outside the triad. Because of this, the design of the bass algorithm will be relatively simple. It will employ a greedy algorithm that chooses between the root and the third of the next chord, always choosing the one that is closest to achieve good voice leading. To guarantee that the bass voice has these options available it will always be generated before the other two harmonizing voices. This will naturally create all four types of motion according to the relationship between the melody and chord progression.

The the alto and tenor voices will use a similar, but slightly different approach. These will also use a greedy algorithm to generate their voices. However, instead of always choosing the closest possible next degree, which would cause all three voices to always move similarly, they will choose between a similar motion or counter motion and be limited to this within each base group as long as voice leading and voice range conditions are met. The motion will be flipped in the case where this is not possible. Because of the greedy approach, oblique motion will occur where it is possible, often due to repeat chords or falling third movements because of the two common notes.

3.2.4 Midi module

Finally, when all values for all voices are generated, the midi module will use these to generate midi values and build a midi file. This module will use the library music21 to generate midi files. Additionally, it will sett the midi file to use midi voices as instruments and make sure that the score is appropriately built with the correct clefs and accidentals.

3.3 Implementation

This chapter will present the implemented methods and functions meant to satisfy the spec presenter previously. To accomplish this, each module will have its own section where solutions are explained and then presented as source code from the project or as pseudocode when it is more appropriate.

3.3.1 Representing notes, chords, scales and keys

In order to generate notes, it is favorable to have a good, abstract representation. The generation algorithm in this project uses the degrees of the scale of the key that each correspond to a specific note. These are octave-independent values, in that a degree can represent one note, but in different octaves. Therefore the octave for each degree must also be generated. This representation is convenient because it guarantees that only diatonic notes are generated. It also makes it easy to expand into different keys and scales. Lastly, it is easy to expand upon so if the algorithm should generate non-diatonic notes in the future, it simply requires to add values for the accidental for each note.

A very similar approach can be used when representing chords as they can also be identified by the degrees of the current scale. Scales can be represented as a set of how many half-steps each degree is above the root and keys can be represented by the amount of flats or sharps in its key signature. As this project limits itself to the key of C major the scale will be `(0, 2, 4, 5, 7, 9, 11)` and the key will be `0`.

3.3.2 Generating a Grouping Structure

In order to generate the grouping structure, the grouping module consists of a class for each level of the hierarchy. The classes are SectionGroup, PhraseGroup, PhraseSubGroup and BaseGroup.

SectionGroup
This class exists mostly to contain the lower level groups, therefore the constructor takes a number of phrases as its parameter, initializes and adds that many PhraseGroups to its attributes.

```python
class SectionGroup():
    def __init__(self, numPhrases: int) -> None:
        assert numPhrases > 1
        self.phrases: list[PhraseGroup] = []
        for i in range(numPhrases):
            self.phrases.append(PhraseGroup(2))
```

The class also contains two methods for going through the entire grouping structure, flatten and groupDescent. Flatten returns a tuple of lists with the relevant base group attributes used when generating streams or figures. GroupDescent takes a method baseGroupMethod as its parameter and calls it for each BaseGroup in the grouping.

PhraseGroup
The constructor of this class generates the full contour for the phrase and the mean values for valence and arousal for its subgroups. In the same way as earlier, the numGroups parameter decides how many PhraseSubGroups should be added to the phrase. The constructor also initializes the amount of number of BaseGroups in each PhraseSubGroup. Although it currently has the same value as numGroups, this is added with the possibility of varying amounts of base groups in the future.

```python
class PhraseGroup:
    def __init__(self, numGroups: int) -> None:
        assert numGroups > 1
        self.numGroups = numGroups
        self.numBaseGroups = numGroups
        self.groups: list[PhraseSubGroup] = []
        contour = self.buildContour()
        meanVs, meanAs = self.generateMeanVAs()
        for i in range(self.numGroups):
            self.groups.append(PhraseSubGroup(
                self.numBaseGroups,
                (meanVs[i], meanAs[i]),
                contour[i]
            ))
        ))
```

This constructor uses two methods to generate the first values of the meta model. The buildContour method generates a nested list with values for either ascending or descending contours. First, it declares the appropriate amount of ascending and descending values, that is one pair of each for every new pair of base groups within the phrase. Then for each PhraseSubGroup a list of randomly selected contour values is added to the contour list and removed from the available directions. Removing the randomly selected values ensures that the melodic phrase with return to approximately the same place as where it starts.

```python
def buildContour(self):
    contour = []
    availableDirections = []
    for i in range(math.ceil((self.numGroups * self.numBaseGroups) / 2)):
        availableDirections += [Direction.ASCENDING, Direction.DESCENDING]
    for i in range(self.numGroups):
        subPhraseContour = []
        for j in range(self.numBaseGroups):
            subPhraseContour += [availableDirections.pop(
                random.randrange(0, len(availableDirections))
            )]
        contour.append(subPhraseContour)
    return contour
```

The generateMeanVAs method uses a normal distribution to generate values for each PhraseSubGroup. A distribution is used to get values centered around a mean, but with the possibility to use different variances. The mean values used in the normal distributions, however, are picked from a uniform distribution.

```python
def generateMeanVAs(self) -> tuple[list[float], list[float]]:
        meanValence, meanArousal = rng.uniform(0, 10), rng.uniform(-10, 10)
        return (
            rng.normal(meanValence, 0.1, self.numGroups),
            rng.normal(meanArousal, 0.1, self.numGroups)
        )
```

PhraseSubGroup

The PhraseSubGroup's job is mostly to distribute the values generated by its parent group. In addition to this the constructor also generates a new set of valence and arousal values, again with a normal distribution, but using the means supplied by the parent group instead of generating new ones. A value diffV is also picked from a uniform distribution which will be used in the BaseGroup.

```python
def __init__(
self, numGroups: int, avgVA: tuple[float, float], contour: list[Direction]
) -> None:
    assert numGroups > 1
    self.numGroups = numGroups
    self.contour = contour
    self.groups: list[BaseGroup] = []
    self.valenceMeans, self.arousalMeans = self.generateVAMeans(*avgVA)
    self.diffV = rng.uniform(-2, 2)
    for i in range(self.numGroups):
        self.groups.append(BaseGroup(
            self.contour[i],
            (self.valenceMeans[i], self.arousalMeans[i]),
            self.diffV
        ))
```

BaseGroup
The final group exists mostly to store the information needed to generate the stream objects and subsequently the midi files.

```
def __init__(
self, dir: Direction, meanVA: tuple[float, float], diffV: float
) -> None:
    self.dir = dir
    self.degrees = []
    self.octaves = []
    self.availableDegrees = []
    self.valence, self.arousal = meanVA
    self.numUnits, self.numBeats, self.durations =
        self.generateRythm(meanVA[1], diffV)
```

This class does start to make decisions based on the value of arousal. The method generateRythm chooses the number of units, or notes, and number of beats in the group. Then calculates durations to evenly distribute the notes across the beats. This is where the diffV attribute is used to split one note into two with half the duration or assimilate two notes into one with twice the duration.

```
def generateRythm(self, arousal: float, diff: float):
    numUnits = 0
    numBeats = 0
    durations = 0
    if arousal < -5:
        numUnits = 1
        numBeats = 4
    if arousal < 0:
        numUnits = 2
            numBeats = 4
    if arousal >= 0:
        numUnits = 4
        numBeats = 4
    if arousal > 5:
        numUnits = 4
        numBeats = 2
    durations = numUnits * [numBeats / numUnits]
    if diff < -1 and arousal > -5:
        numUnits -= 1
        durations.pop(-1)
        durations[-1] *= 2
    if diff > 1 and arousal < 5:
        numUnits += 1
        durations[0] /= 2
        durations.insert(0, durations[0])
    return numUnits, numBeats, durations
```

3.3.3 Melody Generation

The melody is generated by populating the base groups with degrees and octaves. The melody start at the first degree, and then moves upwards or downwards in pitch according to the contour. It also chooses chords for every degree, or every beat in the case where the melody moves faster than in quarters. The melody module is made up of one class with the same name. The MelodyModule class takes a grouping structure and a voice as parameters and uses the attributes currentDegree, currentOctave, currentChord and hasTension to keep track of the melody and chord progression at all times, also between base groups.

```
class MelodyModule():
    def __init__(self, grouping: SectionGroup, voice: Voice) -> None:
        self.grouping = grouping
        self.voice = voice
        self.currentDegree = 1
        self.currentOctave = voice.startOctave
        self.currentChord = 1
        self.hasTension = True
        self.grouping.groupDescent(self.populateBaseGroup)
```

The populateBaseGroup method, which is called for each base group, generates a piece of the melody that fits within the group. It start by choosing between four different types of motion that are either step wise or a variant on counter step motion. Then a check is made to see if the resulting motion will move the melody outside of the range of the voice singing the melody, and if this is the case then the direction of the contour is flipped. The method then goes through each move in the motion and adds the new degree. When all degrees are added the method finds chords that will be harmonizing the melody.

```
function populateBaseGroup
input baseGroup
    moves ← getMoves(number of units)
    if melodyEndsOutsideRange
        direction ← opposite direction
    foreach move
        baseGroup.degrees ← currentDegree
        baseGroup.octaves ← currentOctave
        currentDegree, currentOctave ← currentDegree + move
    availableChords ← chooseChords(valence)
    foreach min(unit, beat)
        nextChord ← getNextChord(availableChords, melodyDegree)
        if melodyDegree not in nextChord
            hasTension ← True
        baseGroup.availableDegrees ← getAvailableDegrees(nextChord)
        currentChord ← nextChord
```

As mention earlier, the chord selection process uses the base group's value of valence to choose a set of chords that reflect the valence. A high valence leads to more major chords and a low valence leads to more minor chords. This is implemented by starting with a list of the primary triads and adding more of the secondary triads to it based on how low the valence is.

```python
def chooseChords(self, valence: float, diff: float):
    chords = [1, 4, 5]
    secondaryTriads = [2, 3, 6]
    if valence > 7.5:
        return chords
    if valence >= 5:
        chords.append(random.choice(secondaryTriads))
        return chords
    if valence < 5:
        chords.append(random.choice(secondaryTriads))
        chords.append(random.choice(secondaryTriads))
        return chords
```

These chords are then used to find the next chord in the progression which is chosen from the intersection between the available chords and the chords that fit in the primary chord progressions from the previous one. If the previous note of the melody was not part of its chord, then the available chords are limited to only chords that include the current note in order to resolve the dissonance from having a non chord tone. If there is no overlap between the available chords and the primary chord progressions then the next chord will be chosen from the intersection between the available chords and the secondary chord progressions.

```python
def getNextChord(self, availableChords: list[int], melodyDegree: int):
    if self.hasTension:
        availableChords = intersection(
            self.getChordToneChords(melodyDegree),
            availableChords
        )
        self.hasTension = False
    inter = intersection(
        primaryChordProgressions[self.currentChord],
        availableChords
    )
    if inter == []:
        inter = intersection(
            secondaryChordProgressions[self.currentChord],
            availableChords
        )
    return random.choice(inter)
```

Lastly for each chord the degrees that are available for the harmonizing voices to use is determined. This is accomplished with two goals in mind. Guaranteeing that all three degrees of the triad is included in each chord and giving the voices the most amount of freedom available. Therefore the available degrees will be chosen from one of two sets depending on the melody degree. If the melody degree is a chord tone then the remaining voices should be able to choose between non chord tones or doubling another voice, and if the melody degree is not a chord tone then the remaining voices should only choose from the degrees in the triad.

```python
def getAvailableDegrees(self, chord: int, melodyDegree: int):
    availableDegrees = []
    if melodyDegree in chordDict[chord]:
        availableDegrees = copy.copy(extendedChordDict[chord])
        availableDegrees.remove(melodyDegree)
    else:
        availableDegrees = copy.copy(chordDict[chord])
    return availableDegrees
```

3.3.4 Generating Three Harmonizing Voices

When the melody and associated chords are generated, it is the job of the harmony module to populate the base groups of the remaining three voices. Just like the melody module, the harmony module has one class with the same name and which takes a grouping structure and a voice as its parameters.

```python
class HarmonyModule:
    def __init__(self, grouping: SectionGroup, voice: Voice) -> None:
        self.grouping = grouping
        self.voice = voice
        self.currentDegree = -1
        self.currentOctave = voice.startOctave
        self.grouping.groupDescent(self.harmonize)
```

The method harmonize is called for every base group. As the base groups are copies of the melody's base groups, the method start by setting the correct number of units in case the melody moves in eighths, updating the durations and resetting values of the degrees, octaves. Then, is the bass voice is being generated, the rootMotion method is called which adds the root note of each chord. Otherwise the algorithm randomly chooses between similar or counter motion as a basis for generating the voice. However this may not end up being final motion of the base group as will be presented further on in this section.

```python
def harmonize(self, baseGroup: BaseGroup):
    baseGroup.numUnits = min(baseGroup.numUnits, baseGroup.numBeats)
    baseGroup.durations = baseGroup.numUnits * [1]
    baseGroup.degrees = [0] * baseGroup.numUnits
    baseGroup.octaves = [0] * baseGroup.numUnits
    if self.voice.voiceGroup == VoiceGroup.BASS:
        self.rootMotion(baseGroup)
    else:
        motion = random.choice([Motion.SIMILAR, Motion.COUNTER])
        self.findDegrees(baseGroup, motion)
```

The rootMotion method works by adding the root note of each and finding the octave that places the root note as close to the previous one without going outside the voice's range. In the case of adding the very first root note when there is no previous root note the algorithm uses the starting octave stored in the voice object.

```python
def rootMotion(self, baseGroup: BaseGroup):
    availableDegrees = baseGroup.availableDegrees
    for i in range(baseGroup.numUnits):
        baseGroup.degrees[i] = availableDegrees[i][0]
        baseGroup.octaves[i] = self.findClosestOctave(
            degrees[i],
            self.currentDegree,
            self.currentOctave
        )
        availableDegrees[i].remove(degrees[i])
        availableDegrees[i] = self.reduceAvailableDegrees(
            availableDegrees[i],
            degrees[i]
            )
        self.currentDegree, self.currentOctave = degrees[i], octaves[i]
```

While root motion is not a concept found in voice leading, it is necessary to satisfy the specification of having the bass voice always singing the root note of each chord. Therefore the similar, counter, parallel or oblique motion of the bass voice will be somewhat incidental. On the other hand for the tenor and alto voices, the method findDegrees seeks to implement all of these.

This method finds a potential next degree using the findNextDegree method which also returns flags for the leap being to large and for the leap putting the voice outside its range. If the voice ends up outside its range, the motion for the rest of the base group is flipped and the current iteration is immediately restarted from the top. On the other hand if the leap is too large, then an alternative degree is found by moving in the other direction. If this alternative does not move the voice outside its range it is used because it is guaranteed to be a smaller leap. If the alternative is outside the voice's range, then all constraints cannot be satisfied and staying within range must be prioritized on the expence of voice leading. This has the benefit of moving the voice away from the edges of its range.

```
input baseGroup, motion
function findDegrees
    i ← 0
    while i < numUnits
        nextDegree, nextOctave, leapTooLarge, isOutside ← findNextDegree
        if isOutside
            motion.value ← motion.value * −1
            continue
        if leapTooLarge
            oppositeMotion ← Motion(motion.value * −1)
            altDegree, altOctave, leapTooLarge, isOutside ← findNextDegree
            if not isOutside
                nextDegree, nextOctave ← altDegree, altOctave
        baseGroup.degrees[i] ← nextDegree
        baseGroup.octaves[i] ← nextOctave
        removeItem(baseGroup.availableDegrees[i], nextDegree)
        baseGroup.availableDegrees[i]  ← reduceAvailableDegrees()
        currentDegree, currentOctave ← nextDegree, nextOctave
        i ← i + 1
```

3.3.5 Producing A MIDI File From Integers

The midi module is the smallest module and its objective is to take the grouping objects that have been populated with degrees, octaves and durations and produce a midi file from them. Because this module only processes data and does not need to store any it does not have a class and consist of a set of functions. The main attraction being getStream.

The function getStream takes a grouping and voice group as parameters and returns a stream object for the entire voice. It begins by setting up the stream object, adding a voice as the midi instrument, setting the correct clef for the voice and setting the tempo to 80 bpm. Then each degree and octave is converted to note objects via midi values. The duration is added to the note and the name of the current chord is added as a lyric to be shown in the final score. The note is then appended to the stream.

```python
def getStream(grouping: SectionGroup, voiceGroup: VoiceGroup):
    s = stream.Stream()
    s.insert(instrument.Tenor())
    s.insert(getClef(voiceGroup))
    s.insert(tempo.MetronomeMark(number=80))
    degrees, octaves, durations, _, _ = grouping.flatten()
    roman = ['I', 'ii', 'iii', 'IV', 'V', 'vi']
    ks = key.KeySignature(0)
    for degree, octave, duration in zip(degrees, octaves, durations):
        n = getNoteFromDegree(degree, octave, ks)
        n.quarterLength = duration
        if voiceGroup == VoiceGroup.BASS:
            n.addLyric(roman[degree-1])
        s.append(n)
    return s
```

When all groupings have been converted to stream objects then are all added to a final stream, which is shown in MuseScore and stored as a midi file.

4 Evaluation of Generated Music

This chapter will present scores generated by the algorithm developed in this thesis. First a few different results will be shown to give a feeling of what kind of music is generated and then two examples will be analysed slightly more in depth.

4.1 Generated Music

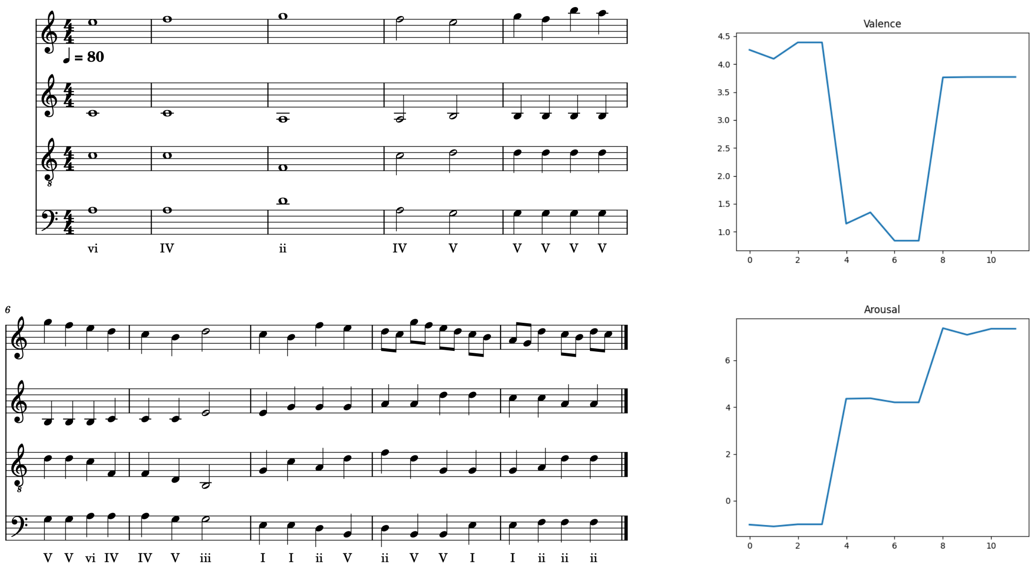
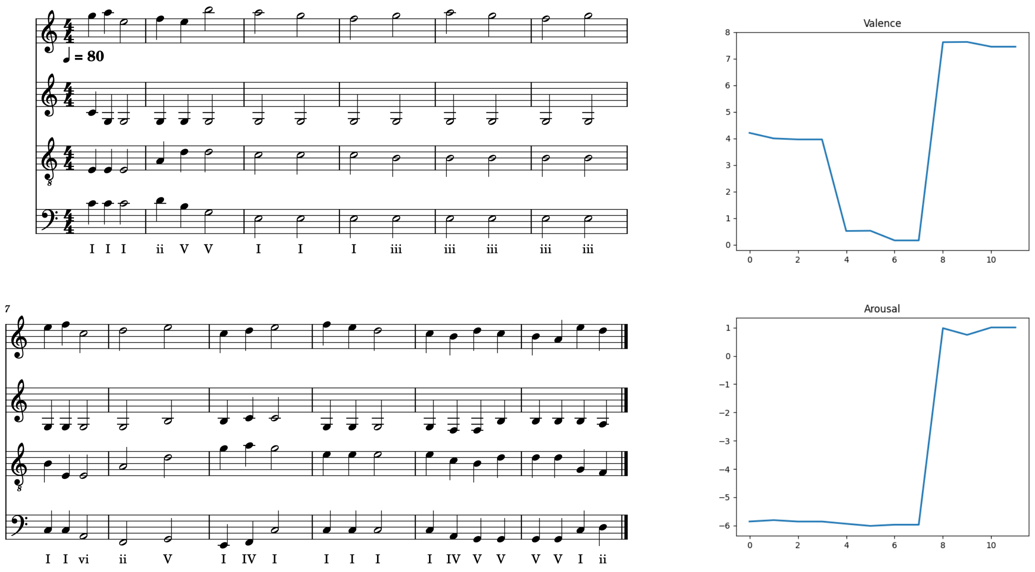


Figure 15: soundcloud.com/user-363908311/harmony-1



♩ = 80
I I I ii V V I I I iii iii iii iii iii
7
I I vi ii V I IV I I I I I IV V V V V I ii
Valence
Arousal

Figure 16: soundcloud.com/user-363908311/harmony-2

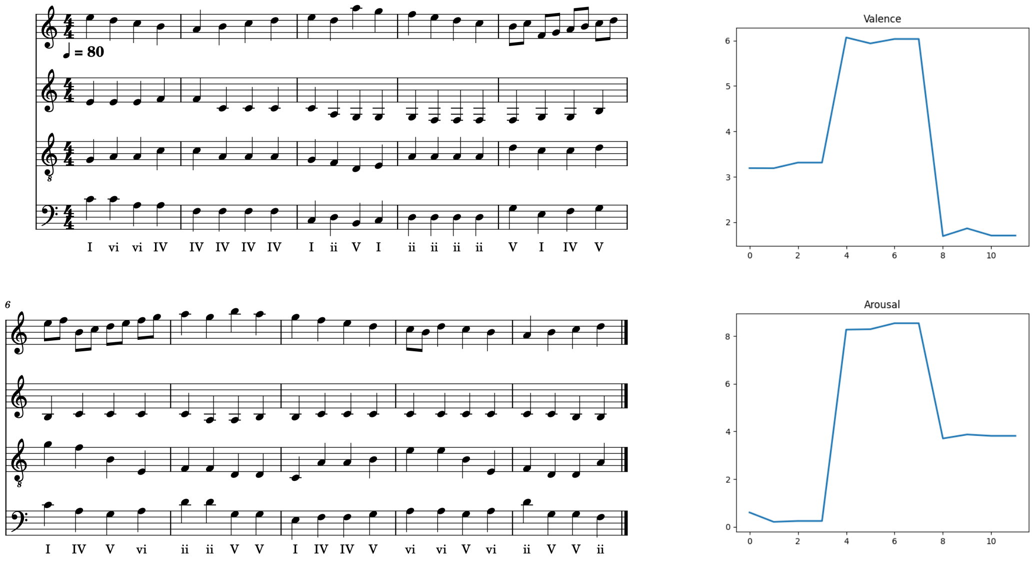

♩ = 80
I vi vi IV IV IV IV IV I ii V I ii ii ii ii V I IV V
6
I IV V vi ii ii V V I IV IV V vi vi V vi ii V V ii
Valence
Arousal

Figure 17: soundcloud.com/user-363908311/harmony-3

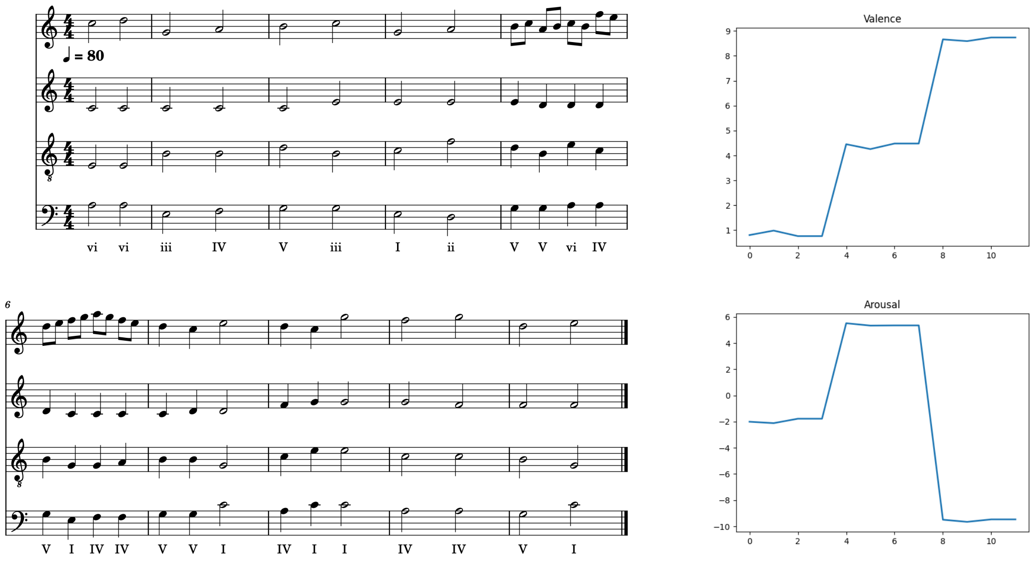

♩ = 80
vi vi iii IV V iii I ii V V vi IV
6
V I IV IV V V I IV I I IV IV V I
Valence
Arousal

Figure 18: soundcloud.com/user-363908311/harmony-4

4.2 Valence and Arousal in a Score

The score in figure 19 can be listened to at soundcloud.com/user-363908311/early-highlow-va. This score is used to show an example of a generated section with low-high-low phrase structure in both valence and arousal. The first phrase, comprised of the first four measures, uses half notes and after three I chords it moves back and fourth between the ii and the iii chords giving the phrase a minor sound. The next phrase replaces the half notes with eight notes and has a much more energetic and bouncy feel. It also only uses the primary triads, I, IV and V, which are all major chords. This creates a contrast to the minor sound of the previous phrase and makes the high valence of this phrase very apparent. The last phrase moves back into using half notes and more minor chords that include a noticeable amount of non-chord tones adding some dissonance which enhances the feeling of low valence.

The dissonance in the minor chords is resolved in the last measure that includes a IV chord followed by a V chord consisting only of chord tones with doubled root notes. This sets up a perfect cadence and is a great example of why an improved algorithm would finish this phrase with a I chord that resolves the perfect cadence. In its current state after being generated, the score ends in a very unsatisfying way, and by manually adding a I chord it is clear that this is what is lacking as can be heard here soundcloud.com/user-363908311/early-highlow-va-with-1. This resolution is even further enhances by the fact that the last note of the melody is the seventh degree of the scale which is also known as a leading tone because it leads naturally into the first degree and home base of the key.

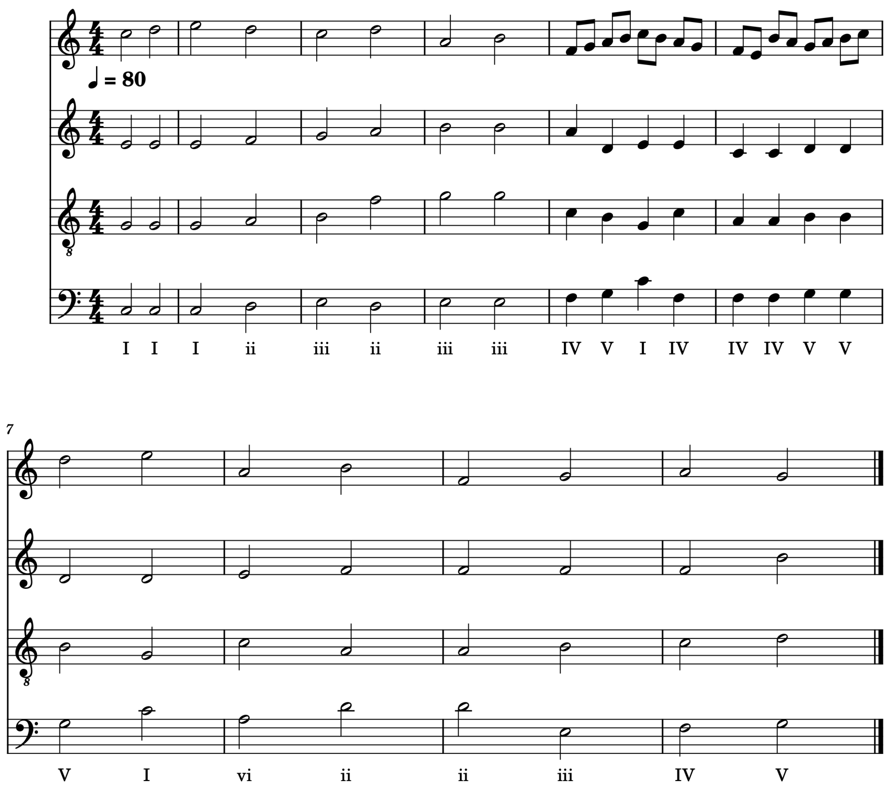
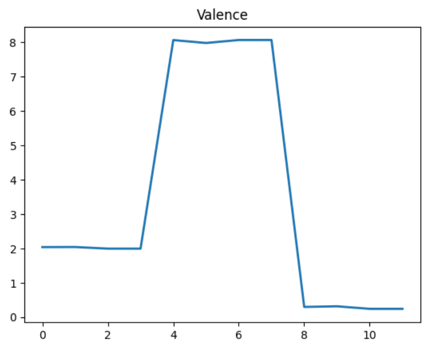
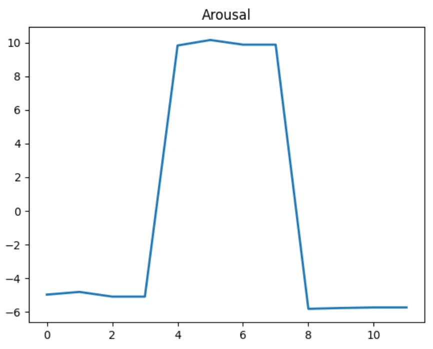


Figure 19



(a)



(b)

Figure 20: Valence and arousal plots for figure 19

4.3 Rules of Writing Harmony

The score in figure 21 can be listened to at soundcloud.com/user-363908311/harmony-rules. This score is an example of the algorithm almost being successful in generating a section without illegal parallels. Not until measure 10 when the chord moves from I to ii and the bass and tenor voices stay a fifth apart during the move. This score also exhibits good voice leading in that each voice is easy to sing, with few large leaps and unintuitive notes. It does however include some noticeable voice crossing between the bass and tenor, putting the tenor far below the bass voice and creating a slightly disturbing sound. The tenor and bass should probably be placed and octave above and below respectively in several measures.

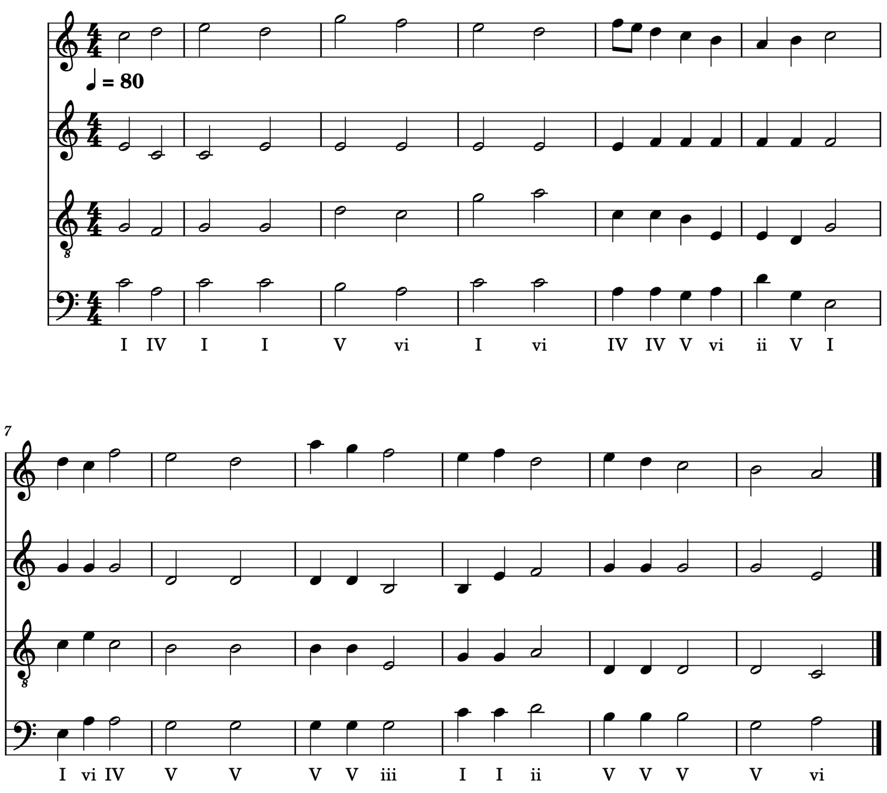
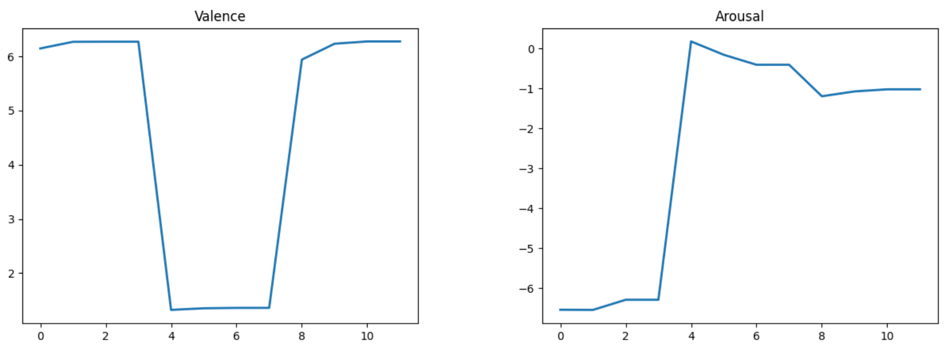

♩ = 80
7
I IV I I V vi I vi IV IV V vi ii V I
I vi IV V V V V iii I I ii V V V V vi

Figure 21



(a) (b)

Figure 22: Valence and arousal plots for figure 21

5 Discussion

This chapter will discuss the strengths and the weaknesses of the resulting system and the scores generated by it. I will also present some ideas for further development.

5.1 Software development

The implementation chapter shows that the sub phrase groups generate a value named diff from a uniform distribution. This value was originally intended to represent local jumps in valence or arousal extracted from the values from the normal distribution. This feature was added so that the base groups within the sub phrase could exhibit some varying traits while maintaining a general similarity to the rest of the phrase. This substitution is an example of half-finished functionality used in the algorithm that should be finished and expanded upon in further development.

Currently, one of the harmonizing voices is generated at a time. Then the available degrees are updated and copied over to the next voice, which is then generated, and so on. The current order is bass, alto, and tenor. This order makes it so that the tenor voice has the least options and must sometimes move around a lot, while the alto voice remains very static because it often has many options. By generating these voices, at least partially, simultaneously, the order of the generation could be switched more often, giving the voices more variability. Simultaneous generation could also allow for more detailed information to be shared amongst the voices more easily. For example, a maximum distance each voice should move, which could help avoid voice crossing.

One change that would most likely improve the system's design and allow for better scalability would be to separate the generation of chord progressions into a separate module. The choice to generate chords after the melody was based on the typical process for arranging harmony, which usually starts with a given melody and then adds harmony to it. However, limiting the generation process to how music is composed elsewhere is unnecessary. By moving the chord progression generation to a separate module and generating it before the melody, the melody could be generated with more precise knowledge about the overall harmonic context within which it exists within. Combined with a more detailed application of valence and arousal, this approach would help the algorithm place dissonance and consonance, large and small leaps and rising and falling contours in more deliberate parts of the composition.

On the other hand, the advantage of generating chords after the melody is the possibility of using the same algorithm to harmonize a given melody. This order is how, as previously stated, harmony is usually written. Therefore, giving the system this possibility would increase its practical applicability. In any case, both of these approaches should be used with a separate chord module. Another argument in favor of this is that it could increase the independence between modules. Between the melody and harmony modules and between the different instances of the harmony module, because a chord module could produce a more sophisticated meta-model for harmony generation.

5.2 Music

There is not much connection between the note durations and how each chord sounds at the current state of the generated music. In classical harmony, some voicings of chords are written to sound temporary. These chords are suitable in passing, while other voicings are more suitable for long, drawn-out durations. The algorithm could generate good-sounding harmony more consistently if it took into account these considerations.

Oblique motion can sound great, but there are cases when a chord is repeated many times, and all voices except for the melody are repeating notes. In this case, a possible improvement could be combining these notes into a single note with a longer duration, which could likely sound better and create a better contrast to the moving melody.

Each voice is now staying within its range throughout each generated composition. However, a composer would usually try to limit the range each voice has to sing throughout a piece to avoid spanning a range that is too large in too short of a time to sing it. This issue could be mitigated by adding local ranges to sections, phrases or base groups according to what is wanted from the generation algorithm.

The scores generated at this time contain illegal parallels, parallel fifths, unisons and octaves. The greedy algorithms that generate the voices in this thesis are not enough to avoid these parallels, and a more thought-out strategy is needed. One possibility could be to generate the voices more simultaneously and redo parts where illegal parallels occur but generate the voices in a different order. The voices would have to choose different degrees from the next chord. However, this starts resembling an optimization strategy and does not guarantee that the goal is accomplished. Another approach could be to detect when two voices create one of these intervals and then make sure that both do not move to the same degree of the next chord.

5.3 Future work

The melody generated by this algorithm is not fully generative and works by combining premade blocks. In order to improve it, a more sophisticated method should be used that is capable of generating more varied and more expressive melodies. By connecting specific movements of a melody to equally specific combinations of valence and arousal, the melody generation should be able to accomplish something like this. Additionally, the current grouping structure is very symmetrical, which potentially does not lead to great expressiveness. Building a grouping with more varied sized could improve this quality.

Another expansion that can be made would be into more counterpoint species. This would require the algorithm to keep track of strong and weak beats, which is something that alone could improve the current generation. It would be very interesting to see if different types of counterpoint could be interpreted as interactions between different grouping structures and if this could allow this theory to expand into polyphonic music.

An exciting possibility would be to combine this type of generation with analysis tools such as the ATTA presented in Implementing "A Generative Theory of Tonal Music" [4] and the valence and arousal detector presented in Music Emotion Maps in Arousal-Valence Space [3]. This method could allow for using a melody as input, analyzing and harmonizing it.

6 Conclusion

The compositional algorithm developed in this thesis generated a melody and three harmonizing voices that stay within each of their ranges, generally have good individual voice leading and realize all chords. The algorithm still lacks the enforcement of rules such as illegal parallels and avoiding voice crossing. However, the resulting scores in this thesis show the potential for creating an algorithm that can create expressive and coherent music. Although somewhat limited in application at this time, by increasing the granularity of the treatment of valence and arousal, this algorithm should be able to produce more expressive and self-similar music.

The modules generated in this did not end up being as independent as originally intended as some values are generated or updated in one module and passed on to another that should be at the same level of abstraction. However, restructuring the modules to have an independent chord module and introducing some more values such as how many degrees the voices should move per base group has potential to further develop this independence.

This type of algorithm could be applied in many settings. It could generate music for video games with varying emotional contexts based on what is occurring. It could help composers write choir parts or arrange harmony to a melody. With the proper application around it, it could potentially create an endless stream of music as ambient noise.

References

[1] Adam Alpern. ‘Techniques for Algorithmic Composition of Music’. In: (1995).

[2] Kathleen A. Corrigall and E. Glenn Schellenberg. ‘Music: The Language of Emotion’. In: *Handbook of Psychology of Emotions*. Nova Science Publishers, 2013. Chap. 15, pp. 299–325.

[3] Jacek Grekow. ‘Music Emotion Maps in Arousal-Valence Space’. In: (2016).

[4] Masatoshi Hamanaka, Keiji Hirata and Satoshi Tojo. ‘Implementing “A Generative Theory of Tonal Music’. In: (2007).

[5] David Miles Huber. *The MIDI Manual*. Routledge, 2021.

[6] *Acoustics — Standard tuning frequency (Standard musical pitch)*. Standard. International Organization for Standardization, 2017.

[7] Steven G. Laitz. *The Complete Musician*. Oxford University Press, 2003.

[8] Fred Lerdahl and Ray Jackendoff. *A Generative Theory of Tonal Music*. MIT, 1996.

[9] Gerhard Nierhaus. *Algorithmic Composition: Paradigms of Automated Music Generation*. SpringerWienNewYork, 2009.

[10] Kara Rodgers. *Equal temperament*. Accessed: 2022-03-03. URL: https://www.britannica.com/art/equal-temperament.

[11] James A. Russell. ‘A Circumplex Model of Affect’. In: *Journal of Personality and Social Psychology* (1980).

[12] Catherine Schmidt-Jones. *Understanding Basic Music Theory*. OpenStax CNX, 2013.

[13] Mary Simoni and Roger B. Dannenberg. *Algorithmic Composition: A Guide to Composing Music with Nyquist*. The University of Michigan Press, 2013.

[14] *The Production of Sound*. Accessed: 2022-05-29. URL: https://www.britannica.com/art/wind-instrument/The-production-of-sound.

[15] Sigvald Tveit. *Harmonilære fra en ny innfallsvinkel*. Universitetsforlaget, 2008.

[16] Rene Wooller et al. ‘A framework for comparison of process in algorithmic music systems’. In: (2005).