Johan Gangsås Hole

Automatic Species Counterpoint

Music Generation at Five Levels Using a Guided Local Search Algorithm

Master Thesis

Supervisor: Sverre Hendseth

Trondheim, May 2021

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

NTNU

Norwegian University of Science and Technology

Master Thesis

Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

© 2021 Johan Gangsås Hole. All rights reserved

Master thesis at NTNU,

Printed by NTNU-trykk

Contents

Abstract

In this thesis, a system is developed that can generate counterpoint pieces in each of the five species as presented by Johann Joseph Fux in 1725. Existing rules of counterpoint from Fux, supplemented by Jeppesen in 1930, are formalized and quantified as a set of constraints. These constraints form the basis of the proposed constraint optimization problem (COP), for which an algorithm is designed to ensure a satisfactory melody generation. The algorithm developed is a guided local search metaheuristic. The search strategy iteratively improves a given counterpoint melody by always picking and improving upon the note in the melody that leads to the most accumulated penalty. The search algorithm is generalized and applicable to all of the five different species. The user provides input parameters such as key, scale type, vocal range and type of species. The generated results are exported symbolically in midi-format, which can be further handled by programs such as musescore.

Generated results have been made available on soundcloud. The reader is encouraged to listen to the auditory examples linked throughout the thesis. An example of a fifth species counterpoint generation is given here: `https://soundcloud.com/johan-gangsas-hole/automatic-generation-of-fifth-species`.

Chapter 1

Introduction

The question of whether musical compositions can be automated has been a topic of conversation for several centuries. Already in the baroque era of Bach, musical dice games aided novice composers to generate music from a set of pre-composed options randomly. These dice games also include variants of automated counterpoint by the well-known composer C.P.E. Bach, the son of Johann Sebastian Bach. Bach devised a game for "making six bars of double counterpoint at the octave without knowing the rules" [26, p. 36]. The computer was introduced to aid in algorithmic compositions as early as 1955. The composition was done on a symbolic level and was produced by Hiller and Issasscon from 1955-1956 with the "Illiac Suite" . Since then, a large variety of different algorithms and paradigms has been used in computer-aided compositions. These include generative grammars, rule-based systems, transition networks, genetic algorithms, and stochastic models such as Markov models [26].

In species counterpoint, a melody is composed over or below a given melody called the cantus firmus. The goal of the counterpoint is to be as independent as possible from the cantus firmus, while still creating pleasing harmonies when played together. The main difference between the different species are the rhythms. As an example: first species has one note for each note in the cantus firmus, and second species has two notes for each note in the cantus firmus. The rules regarding how to compose the counterpoint has most famously been laid out by Johann Joseph Fux.

The goal of the system presented in this thesis will be to generate all of the five species of counterpoint. Previous solutions seem to be concerned mostly with only first- or fifth species, which motivates the idea of making a contained system that can handle all of the five species. This will be achieved by implementing a guided local search algorithm that is issued on a pre-constrained system. The rules presented by Fux and Jeppesen are to be structured as system constraints in a cost function that is to be minimized. The task of the complete algorithm will be to incrementally adjust a initially randomized melody pitch sequence until the cost function is under a certain threshold. Having a well formed system structure will also be a goal in of itself. This means that the proposed algorithm should be generalized to the extent that it can be used for generation of all the five different species. To achieve this, certain care must be done during the design phase.

The choice of focusing on species counterpoint as the composition task was made because of several reasons. The pedagogical framework in which the species was first outlined makes it approachable and contained. It also supports a good balance of abstraction and generality compared to other composition styles. Its division into different levels, or species, with a concrete rule-set that is sufficiently precise, gives a well-formed system specification in the form of explicit counterpoint rules. Therefore, the system design phase has a good entry point since the system specification to a large degree is already given by formal rules. Fifth species counterpoint has also been used extensively in algorithmic composition tasks [10, 6, 20, 1, 32], making it possible to compare solutions and strategies.

The problem of algorithmic composition is interesting in itself, but we are also interested in system design, particularly scalability. Therefore, the music representation in the style of objects and how these objects can be further structured to reduce higher-order complexity is also addressed. Modern approaches in algorithmic composition has also shifted more towards machine learning methods during the last decade. Although these results have been promising - as will be discussed in section 2.2 - the issue of such "black-box" implementations is the loss of control of the generated results. A system based on a rule-based paradigm was therefore more intriguing, as this led to a more in depth study of how rules of composition could be expressed in a computer system.

The thesis is structured as follows. First, some music theory is presented to give context to the problem to be solved. This musical introduction includes a brief study of counterpoint and harmonization, in addition to different musical concepts such as intervals. The presentation of counterpoint also include the rules as presented by Fux and Jeppesen. The second part of the background chapter is a presentation of existing methods for algorithmic composition. This presentation is both in regards to general approaches and practices dealing with automatic counterpoint specifically. Based on the information regarding existing solutions, the proposed system is design in chapter 3. Chapter 4 presents the implementation, while the results are presented in chapter 5. Lastly, a discussion is made with remarks regarding the development phase and choices made during the system implementation. It is also discussed whether the implemented system was successful in generating species counterpoint, in addition to possible improvements and future work.

Chapter 2

Background

This chapter provides background to the work described in this thesis. First, we give an introduction to counterpoint and the different rules for each species. Then, an overview of other techniques for algorithmic composition is outlined from a historical perspective. We separate this preview into two main categories, AI-based and knowledge-based. Lastly, more concrete examples of systems for counterpoint generation are presented.

2.1 Counterpoint

In this section, basic concepts regarding music theory are presented. In particular, the set of rules for strict contrapuntal writing is described. The information here is based on the counterpoint book by Johann Joseph Fux, translated in 1965 by Mann [7]. Information regarding intervals and other fundamental music theory is from Steven G. Laitz's "The Complete Musician" and Catherine Schmidt-Jones' "Understanding Basic Music Theory", both books on fundamental music theory [21, 31].

Counterpoint is when more than one independent melodic line is happening simultaneously in a piece of music. The music is then contrapuntal [31, p. 85]. Independent melodies mean that whatever is happening in one line (both rhythmically and melodic) is independent (or, if possible: different) from what is happening in the other lines. One simple example of counterpoint is a *round*, where everyone sings the same melody, but starting at different times. In this way, even though everyone is singing the same tune, the fact that people will be singing different parts leads to independence between voices. An example of a simple round is given in figure 2.1. Here, the asterisk's indicate the entry points where a new part

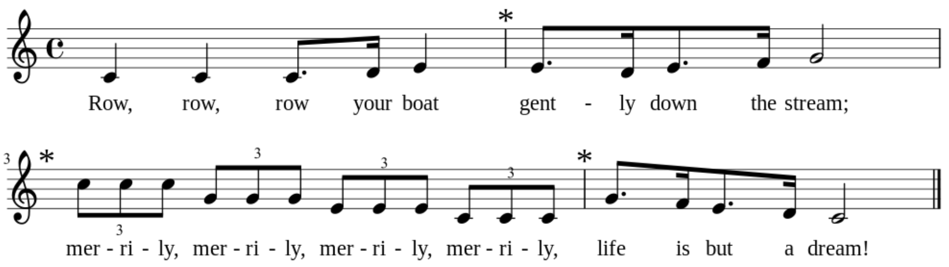

Row, row, row your boat gent - ly down the stream;
mer - ri - ly, mer - ri - ly, mer - ri - ly, mer - ri - ly, life is but a dream!

Figure 2.1: Simple example of applied counterpoint in "row your boat".[1]

can join the singing from the start. Given the three asterisks in this example, a maximum of four different parts can be sung simultaneously.

Counterpoint has since the beginning of the tenth century been a focus of western music together with polyphony and harmony [21, p. 47]. This led to an early pedagogical study by the early eighteenth centrury, mostly credited to the works of Johann Joseph Fux in his book *Gradus ad Parnassum* from 1725 [7]. The book consists of compositional exercises divided into five levels or *species*. Given a melody called the *cantus firmus* a new melody is constructed as a counterpoint based on certain rules and constraints.

To write a satisfactory counterpoint, one must first have a good understanding of intervals. Presented with a pair of pitches, the various distance between these two pitches form intervals. These intervals are, along with the rhythm, one of the fundamental building blocks of tonal music [21, p. 713], and a presentation is therefore given in the following section.

2.1.1 Intervals

As mentioned above, *intervals* are the distances between two pitches. Therefore, an interval gives a relationship between notes, where one might be higher or lower than the other. Describing smaller intervals can be done using half steps (adjacent keys on a piano keyboard) and whole steps (consists of two half steps). One example is "D natural is a whole step above a C natural". Larger intervals require a more descriptive way of naming them, typically done by an ordinal number representing the number of pitch-names that span the two notes [31, p. 136-140]. For example, an interval from C to G is a fifth since five pitch names span C to G (C, D, E, F, G). One can also identify intervals by their number of half steps. A fifth,

[1]picture from https://en.wikipedia.org/wiki/Round_(music)

for instance, correspond to seven half steps.

Intervals can further be divided into *simple intervals* and *compound intervals*. The simple intervals are limited to be within a range of an octave. Compound intervals are larger than an octave but are often expressed as their simple counterpart [21, p. 13]. Figure 2.2 shows all of the simple generic intervals with their corresponding ordinal name. A more specific representation is given in figure 2.3. Here, all of the specific simple intervals for a C major scale is shown. The prefixes "P" and "M" indicate the two basic categories of intervals: the unison, fourth, fifth and octave are *perfect* (P) intervals, and the rest are *major(M)/minor(m)* intervals, also called *imperfect intervals*. These two categories are used to distinguish the quality of the interval. In a harmonic context, perfect intervals are more stable and "at rest". Imperfect intervals might create tensions that need to be resolved.

Both imperfect and perfect intervals can be further transformed into other intervals [21, p. 13]. Increasing the intervals by a half step creates an *augmented* interval. Decreasing the interval by a half step results in a *diminished* interval. The only interval that can not be diminished is the unison, but all intervals can be augmented. It is, however, impossible for an imperfect interval to be augmented or diminished to a perfect interval and vice versa. A simple example of an augmented interval is shown in figure 2.4.

The transformation induced by augmentation and diminution means that similar-sounding intervals can have different names [31, p. 144]. One such example is the *tritone*, which gets its name from the number of whole tones in the interval, three. A tritone can be expressed as a diminished fifth or augmented fourth. Since this interval is neither major, minor or perfect, it is unusually *dissonant* and unstable and must therefore be handled with care.

Such dissonances lead us into the *quality* of the intervals, namely the level of stability. The perceived stability or instability of a given interval is highly determined by musical context. However, it is possible to categorize the intervals into two new categories, which is essential in the study of counterpoint. These categories are consonant and dissonant intervals. Consonant intervals consist of stable intervals - including unison, third, fifth, sixth, and octave. The dissonant, or unstable, intervals include the second, the seventh, and all diminished and augmented intervals. The perfect fourth can either be stable or unstable depending on musical context, but in the study of counterpoint, this interval is considered to be dissonant [31, p. 183-186].

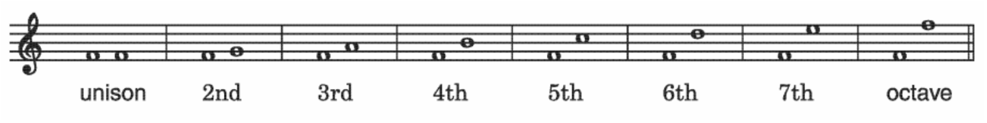


Figure 2.2: Generic simple intervals with ordinal names [21, p. 13].

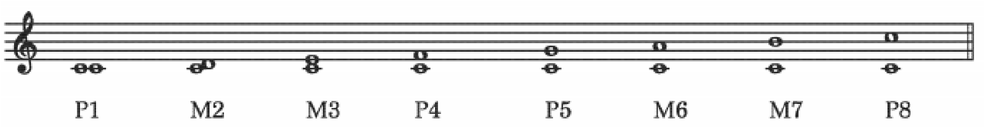


Figure 2.3: Specific intervals for the C major scale above the tonic [21, p. 13].

The consonant intervals can further be divided into two types: *perfect consonances* and *imperfect consonances*. Figures 2.5 and 2.6 summarizes the types of intervals. The interplay between the different types of intervals is a powerful tool for composers to create motion in their music; for example, a seventh that is immediately followed by a consonance, like a perfect fifth. This creates tension that is resolved when a dissonance goes back to a perfect consonance.

2.1.2 Contrapuntal Motion

When two voices move together, as is the case of counterpoint, they create different contours depending on how the voices move relative to each other. The different contours create what is known as *contrapuntal motion*, and there are in all four different motions for species counterpoint [21, p.50].

The first one is when the two melodies move in opposite directions from one another. This creates **contrary motion**. Example of contrary motion is shown in part A of figure 2.7. This is the motion that gives the most voice independence, and should therefore be preferred.

The second type of motion is **similar motion**, which means that the voices move

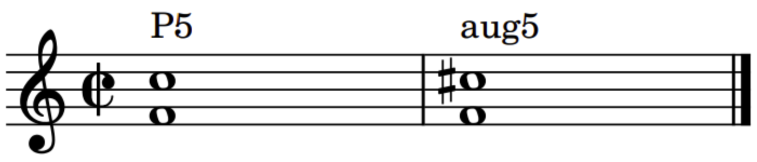


Figure 2.4: Augmentation of a perfect fifth.

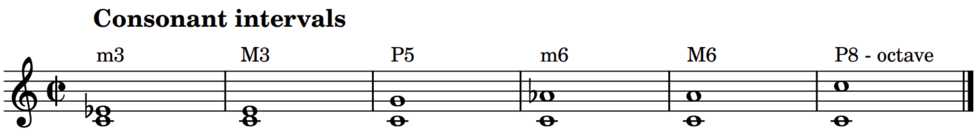


Figure 2.5: Consonant intervals.

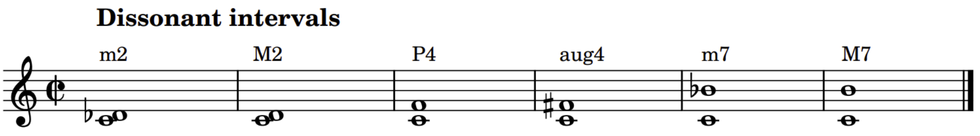


Figure 2.6: Dissonant intervals.

in the same direction but with different melodic intervals. This is illustrated in part B of figure 2.7.

When one voices stays monotonic while the other moves freely, we get the third type of motion: **oblique motion**. In oblique motion, there is only one of the voices which changes pitch value. An example is shown in figure 2.8A.

The last type of motion is **parallel motion**, which creates the most dependence between the voices and should therefore be avoided if possible in counterpoint. In parallel motion, the voices move in the same direction with the same melodic interval. This is illustrated in 2.8B.

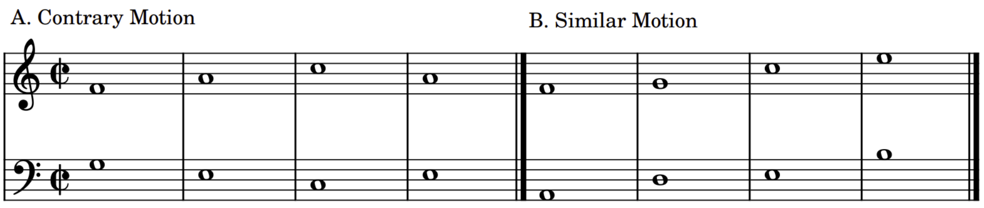


Figure 2.7: Example of contrary and similar motion between two voices.

2.1.3 Cantus Firmus

Cantus firmus (plural: cantus firmi) is Latin for "fixed song" and is a preexisting melodic line used as a basis of the contrapuntal composition [26, p.48]. The cantus firmus is often abbreviated CF. Fux presents many examples of cantus firmi in all of the different modes (scales) in his study of counterpoint. These melodic lines are mono-rhythmic, meaning that the notes are of equal length (often whole notes).

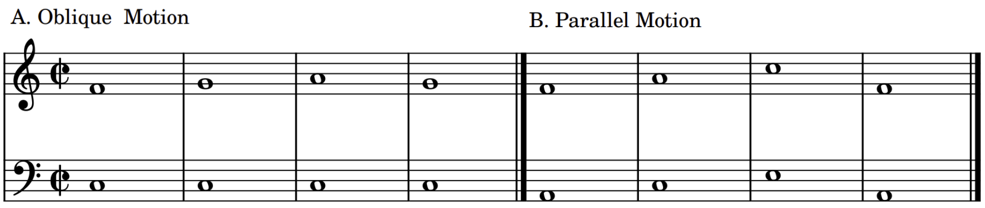


Figure 2.8: Examples of oblique and parallel motion between two voices.

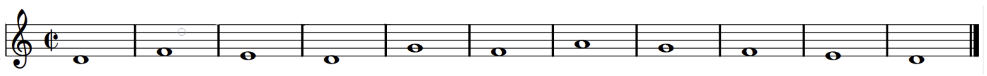


Figure 2.9: Example of a cantus firmus in the dorian mode [7].

The number of pitches is usually between 8 and 14 notes [7]. The cantus firmus is the basic structural pillar against which the counterpoint voice is added. How the counterpoint voice move in relation to the CF is what separates the different species. Each species introduces a new way of complementing the CF rhythmically, creating tension, resolution, and melodic fluency. Figure 2.9 illustrate a cantus firmus example in dorian mode given by Fux in *Gradus ad Parnassum* [7].

Rules

Since the cantus firmus is often given as an initial melody which form the basis of the counterpoint to be composed, the rules regarding how to compose cantus firmus melodies is not as formalized as that of species counterpoint. It is, however, still possible to list some rules by studying Jeppesen's preliminary exercises regarding melodies in first species and Palestrina-style melodies in general [18, p.83-97 and p. 109-112]. The rules are similar to that of first species melodic consideration, with some minor differences:

1. One must begin and end on the tonic to emphasize the key.
2. Only melodic consonances may be used.
3. The cantus firmus should have a clear climax point.
4. All notes must be whole-notes.
5. The cantus firmus should be within a range of a tenth and in a singable vocal range.
6. The penultimate note should be a major second above the tonic. A minor second below is also allowed, but this should be rare.

7. All perfect, major, and minor intervals up to the fifths are permitted in ascending as well as in descending motion, as is the octave. Only ascending and not descending minor sixths are allowed.

8. Having too many skips in the cantus firmus is bad. Likewise, having too long sequences of step-wise motion might sound trivial. One should therefore aim to find a balance between the two.

9. Large leaps should be recovered by a step in the opposite direction.

10. Care should be taken in having successive large leaps both in the same and opposite direction.

11. Successive note repetitions is not allowed, and a note should not be repeated too much across the entire cantus firmus given its short length. One must also be careful not to repeat motivic sequences, as this might come at the expense of the melodic direction.

12. Following the lengths of the cantus firmi provided by Fux, the length of the cantus firmus should lie between 8 and 14 notes.

13. Leading tones (seventh scale degree) should be resolved by the tonic.

By inspecting the dorian cantus firmus in figure 2.9, one can observe that all of the rules are satisfied. Since the cantus firmus is in the dorian mode, the tonic is *D*, which is correctly the start and end note of the cantus firmus melody. The cantus firmus is also well within the maximum allowed range of a tenth, with a good mix of both small leaps and step-wise motion.

2.1.4 First Species

First species counterpoint begins with introducing one new note for each note in the cantus firmus. This new melody can either be above or below the cantus firmus. The intervals used must also be consonant and as independent as possible. This means that they should ideally move in different melodic motion to the cantus firmus. This results in a note-against-note or 1:1 counterpoint. The added voice is also called the contrapuntal voice [21, p. 48-49]. A simple example is shown in figure 2.10A. Here, the counterpoint is given above the cantus firmus. Notice how the voices move with close to mirrored contours in contrary motion. This is to enforce as much independence as possible. The melodic climaxes are also different. The climax for the cantus firmus is in measure three, while the climax for the counterpoint voice is in measure five.

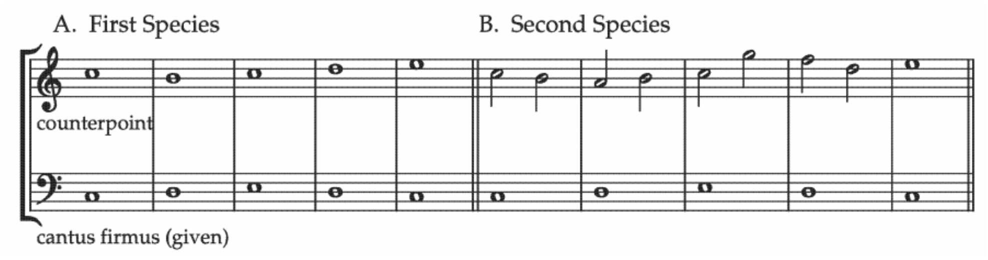


Figure 2.10: Example of first and second species counterpoint using the same cantus firmus [21, p. 48].

The notes that form the counterpoint must follow a set of rules and guidelines to ensure both harmonic and melodic fluency. One such rule, and arguably the most important according to Fux [7], is that *all harmonic intervals between the cantus firmus and the counterpoint must be consonant.* This rule creates a framework of feasible note possibilities, and other rules are introduced to ensure a well-formed global structure adherent to the form. Example: *Any motion is allowed except for the direct motion into a perfect consonance.*

Several more rules and considerations have to be made when composing first species counterpoint. These rules are now presented in the following subsection. By enforcing these rules, the number of note repetitions allowed, leaps, possible start and end notes in addition to counterpoint range is constrained.

Rules

The rules for different species of counterpoint is taken from a translated and modernized version of Fux's *Gradus ad Parnassum* [7, p.27-70]. Since the work of Fux is presented as a dialog between a teacher and a student, some of the rules are somewhat unclear. Therefore, Knud Jeppesen's work on counterpoint from 1930 [18] is used as a supplement to formalize the rules with a clearer distinction. Before the rules are presented, some additional musical terminology is explained.

Diatonic means the seven steps that an octave is divided evenly into in minor and major scales. A melody in C-major is therefore diatonic if all of the notes in the melody is one of the white keys on a keyboard; C-D-E-F-G-A-B [21].

Harmonic intervals are the vertical intervals between the counterpoint and cantus firmus voice.

Melodic intervals are the intervals from one pitch to the next in a single voice line.

Cadence is a melodic or harmonic configuration that creates a feeling of resolution. Cadences often appear at the end of music phrases as a form of musical punctuation [21, p. 43].

Bellow follows a list of all the major rules for first species composition:

1. For every note of the cantus firmus, there is one note in the counterpoint.
2. The counterpoint is diatonic except for the raised leading tone in minor. This means that all of the notes must be within the given scale implied by the cantus firmus except for occasionally the leading tone in minor.
3. All harmonic intervals must be consonant (a perfect fourth is considered a dissonance).
4. The voices should lie within their respective ranges - bass, tenor, alto or soprano. This is to keep each separate part within a reasonable and singable range.
5. The first harmonic interval between the cantus firmus and counterpoint voice must be any perfect harmony and the last an octave or unison. If the counterpoint lies in the lower part, however, only the octave or unison might be used.
6. Unisons may occur only on the first and last notes of the counterpoint melody.
7. The maximum range between the cantus firmus and counterpoint should rarely exceed the interval of a tenth.
8. The last interval must be approached by motion of a minor second upwards or major second downwards, depending on the penultimate note of the cantus firmus.
9. Jeppesen supplements Fux and specify the importance of a clear high point in the counterpoint melody that should not be exceeded or introduced more than once. The rule of having a distinct climax-point is therefore added.
10. Upper voices can sometimes cross if necessary, but avoid "overlapping" (in an overlap voices do not cross, but one moves to a position that is at or beyond the previous pitch of another voice). Examples of rule violations is shown in figure 2.12.

11. All perfect harmonic intervals must be approached by contrary motion.

12. Melodic motion can proceed by step or leap but steps and leaps of augmented and diminished intervals and leaps of any seventh are forbidden. Leaps greater than an ascending sixth are forbidden except for leaps of an octave which should be rare.

13. The counterpoint may not outline an interval of a tritone or seventh except for an augmented fourth that is fully stepwise outlined and precedes an inward step. See figure 2.11 for an example of a tritone outline. Dissonant intervals is therefore avoided both harmonically (between the counterpoint and cantus firmus), and as melodic contours.

14. No note may be repeated more than three times successively. Jeppesen seems to be more strict regarding repetitions than Fux. While Fux set the limit at three repetitions, Jeppesen strictly specifies: "The repetition of a tone is permitted occasionally in the first species, and there only." [18, p. 111], implying that the number of repeating notes should not exceed two.

15. No two successive melodic leaps in the same direction may total more than an octave.

16. While ascending, in the case of two successive melodic steps or leaps, the larger one should precede the smaller; while descending the smaller should precede the larger. Jeppesen, however, points out that this rule should not be applied too rigidly in first species counterpoint between whole notes [18, p. 109].

17. No successive melodic leaps in opposite directions; leaps should be followed by inward, step-wise motion.

18. The same harmonic interval should not repeat more than three times successively.

19. There should be no more than two successive melodic leaps.

20. The range of the counterpoint should be limited to a tenth.

21. Contrary motion should be preferred.

22. No voice should move by a chromatic interval (any augmented or diminished interval).

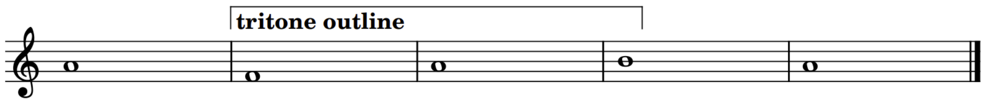


Figure 2.11: Example of violation of rule 13 for first species counterpoint. The stepwise motion F-A-B outlines a tritone, which is not allowed.

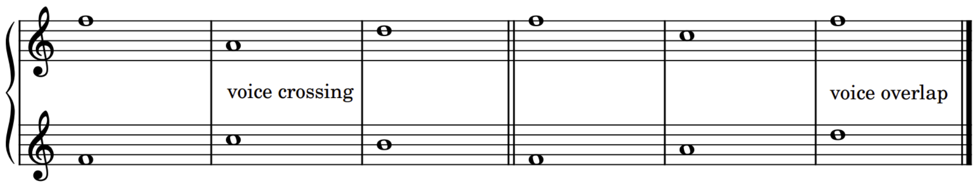


Figure 2.12: Example of violation of rule 10 for first species counterpoint. The first example contains a voice crossing, while example two illustrates voice overlap.

2.1.5 Second Species

In second species counterpoint, two notes are written for each note in the cantus firmus. This form is also called 2:1 counterpoint [21, p. 48]. In contrast to first species counterpoint, second species introduces the possibility of dissonant intervals on weak-beats. Other than that, the rules are generally the same as first species. For example, all harmonic intervals on downbeats must be consonant. An example of second species counterpoint is shown in figure 2.10B.

Rules

Second species build upon the rules of first species, adding the rules listed below:

1. The repetition of notes should now be avoided in second species and in the remaining species.

2. The counterpoint must end on a whole note.

3. The accented portion of the measure (beat 1) can only have consonances.

4. The unaccented portion of the measure (beat 2) may have either consonances or dissonances. Consonances may be introduced freely, while dissonances must be approached and left by step continuing in the same direction. In other words; it must fill in the interval of the third between the two notes on either side of it. Figure 2.13 illustrates a valid dissonance, approached and left by step in descending motion.

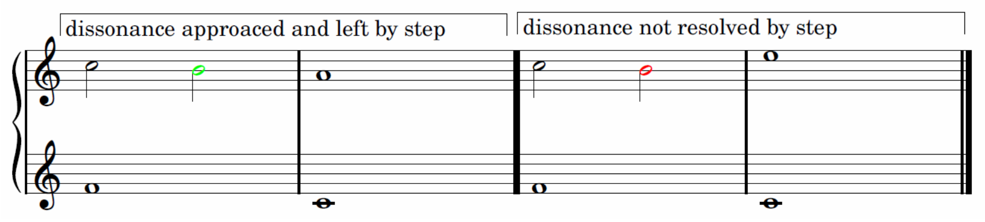

dissonance approaced and left by step
dissonance not resolved by step

Figure 2.13: Example of allowed and disallowed dissonance handling for second species. The dissonance in the first measure is properly resolved with continuous step-wise motion, while the dissonance in the third measure is followed by upward leap and is therefore not resolved.

5. As in first species; avoid unisons except at the terminals. Authorities disagree: Fux forbids unisons except at beginning and end (though he occasionally includes them in his examples). Jeppesen is less strict. But since we are interesting in staying true to the Fuxian style, we will keep Fux's objection to unisons.

6. Must begin on an up-beat (beat 2 of the measure), and the first tone must be the tonic or the fifth of the scale.

7. Avoid consecutive melodic intervals on the same pitches. That is, motivic repetitions.

8. Accented fifths or octaves following each other on successive accents should be avoided if possible, unless the intervening accompaniment note leaps by more than a third. In the second of these parallels between downbeats the leap of a fourth is thought to mask the effect of the parallel. This rule exception is somewhat unclear, so an example of two parallel octaves is given in figure 2.14. The first one is not allowed, while the second one is allowed due to the skip in the intervening note.

2.1.6 Third Species

Third species does not introduce any more types of dissonances, but it does make possible richer and more varied melodies. The added counterpoint will now be in quarter-notes, except for in the last measure which will be a whole-note to emphasize the cadence. The number of notes is therefore twice that of second species counterpoint, and is called 4:1 counterpoint.

In contrast with the other two species presented so far, third species has a lot more

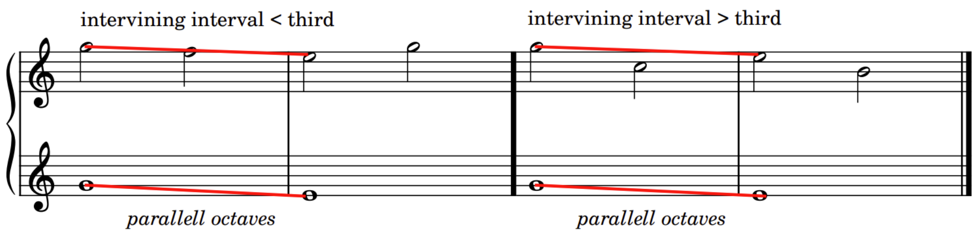


Figure 2.14: Example of violation of parallel perfect intervals for second species counterpoint. The intervening interval in measure two is too small to mask the parallel octave. The parallel octave is properly handled in measure three.

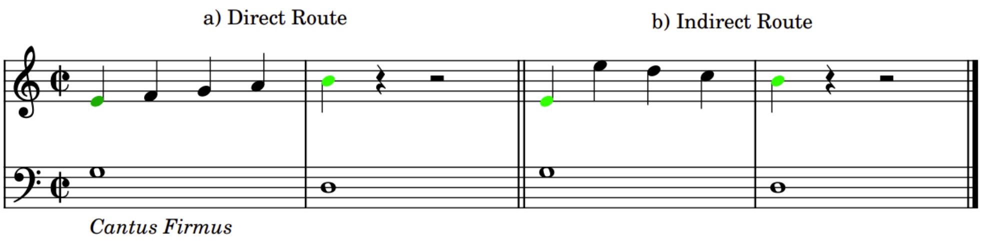


Figure 2.15: Two possible melodic routes. Notes on strong beats are indicted in green.

note repetitions across the whole melody. This is due to the limited range and number of notes. To prevent monotonic melodies, third species introduces the notion of different *routes* [7, p. 51-54]. *Direct routes* stay between the two notes on strong beats in consecutive measures. *Indirect routes* goes out of the range between two consecutive strong beats, leading to a more unpredictable and varied melodic line. Examples of routes is shown in figure 2.15.

Given the increased complexity, musical context also becomes more apparent. Parallel motion is still constrained, but is allowed if the number of beats between them are at least four and occurring on weak-beats. This results in a more goal oriented composition with a sense of direction, with a goal to peak on a consonance. The

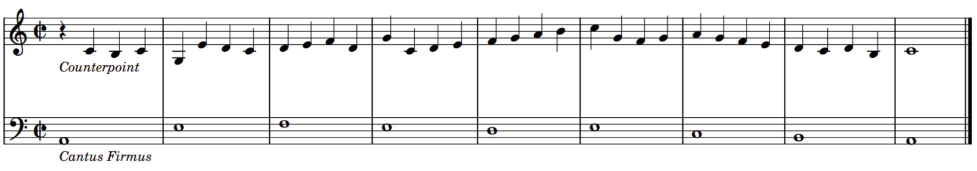


Figure 2.16: Example of a third species counterpoint. Courtesy of Alan Belkin [4]

added difficulties also mean that the whole cantus firmus must be analyzed before even beginning to compose the counterpoint voice. This is to ensure no overlapping climaxes or voice crossing. An example of a third species counterpoint above a given cantus firmus is shown in figure 2.16. Notice how the contour of the counterpoint ascends to a peak in measure 6 before descending down to a imperfect consonant in the last measure, while the cantus firmus is mostly descending after the initial leap in measure 2.

Rules

1. Four quarter-notes for each whole note in the cantus firmus, except on the start, which start on a rest, and the last note, which must end on a whole-tone.
2. The first note must, as in the preceding species, be a perfect consonance. However, imperfect consonances may be used occasionally if this leads to a better overall melodic structure.
3. The first and third quarters in each measure must be consonances.
4. The second and fourth quarters might be consonances or dissonances. The conditions are the same for that of second species. Jeppesen however, contrary to Fux, does not restrict movement to the continuation in the same direction. It might return to the tone from which one started. This is called auxiliary notes, and only lower auxiliary notes are allowed in Jeppesen's modernized counterpoint.
5. One common exception to the above-mentioned rule is the descending skip of the third following an unaccented quarter note introduced stepwise from below, see figure 2.17. Unaccented quarter notes introduced from above is treated less rigorously. Some common rule exceptions with unaccented notes introduced from above are shown in figure 2.18.
6. Aside from in the first and last measures, unisons can only appear on beats 2-4 in each measure, but only rarely.
7. No exceptions are permitted to the rule that "larger intervals must precede smaller ones in upwards movement", and vice versa, where the direction is opposite.
8. Two or more successive skips in the same direction is not permitted.
9. No upward skips from an accented quarter note is permissible.
10. Skips should be filled out immediately.

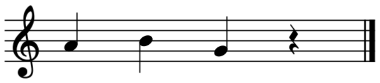


Figure 2.17: Allowed exception to rule number 5 for third species counterpoint.

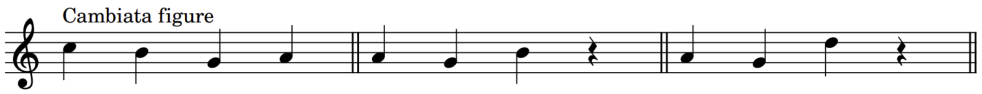


Figure 2.18: Common figures with the unaccented quarter note introduced from above. The figure in measure one is the much-liked cambiata figure.

11. Descending or ascending skips from two successive accented quarter notes are to be avoided.

12. Repetitions of notes within a bar should be avoided when the repeating note is introduced from above. Therefore, the motifs shown in figure 2.19 are rare in pure Fux style. Note repetition introduced from below is allowed, as this was a popular and common ornament in the sixteenth century.

13. Given the increased restriction due to the introduced notes, accented fifths or octave on successive accented quarters following each other may be permitted very rarely.

2.1.7 Fourth Species

Fourth species counterpoint is rhythmically different from the preceding species. Fourth species introduces a new form of dissonance, the *suspension*. Up until now, dissonances has always been placed on weak-beats and approached and left by step. The suspension however introduces dissonances on strong beats. To achieve this, the note is started on a consonance on a weak beat in the previous bar, and then, sustained, becomes dissonant in the next bar as the harmony changes around it [4]. Each suspension consists of three steps; the preparation, which must be consonant, the dissonance, which is the same note as the preparation but with another harmony surrounding it, and lastly, the resolution, another consonance to resolve the suspension. It is, however, possible that the tied note is a consonance in both measures. It is then possible to leap in stead of moving by step. Figure

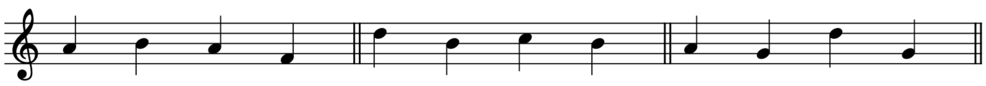


Figure 2.19: Examples of note repetitions in measures that should be avoided in Fux style.

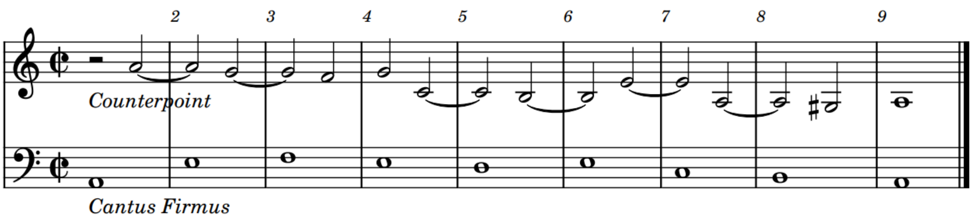


Figure 2.20: Example of a fourth species counterpoint. Courtesy of Alan Belkin [4]

2.20 shows an example of fourth species counterpoint with the cantus firmus given below. The suspensions are illustrated with tied notes across the bar-lines. Notice how the chain of suspension is broken in measures three and four. According to Fux, this may only happen maximum one time each exercise. Notice also how the dissonances is mostly resolved by step downward. This makes it hard to create convincing melodic lines in fourth species [4].

Rules

1. As in second species, there are two half notes for each note in the cantus firmus.

2. The rhythm is now syncopated, meaning that the unaccented half note in each measure is tied to the accented one immediately following.

3. Dissonances may only be used on accented half notes (beat 1), such that the dissonant tone is tied over from the unaccented part of the preceding measure, where it must be a consonance with the cantus firmus. In fourth species such dissonances are preferred to consonances.

4. Each dissonance must be followed by a step-wise movement downwards to an imperfect consonance. Therefore, when the counterpoint is in the upper voice, only the seventh and fourth may be used as a suspension dissonance. When the counterpoint is below the cantus firmus, only the second and ninth can be used. Note that this is only necessary when the syncope is dissonant.

5. It is possible to break the chain of syncopation, which should be done very rarely. This give rise to second species movement that must be handled according to the rules of this species. Figure 2.21 illustrates this, where the break in syncopation in measure two leads to one measure of second species in measure three. Notice that since a syncopated consonance occurs on the strong accent in measure three, it is permissible to take a passing dissonance on the following weak-beat.

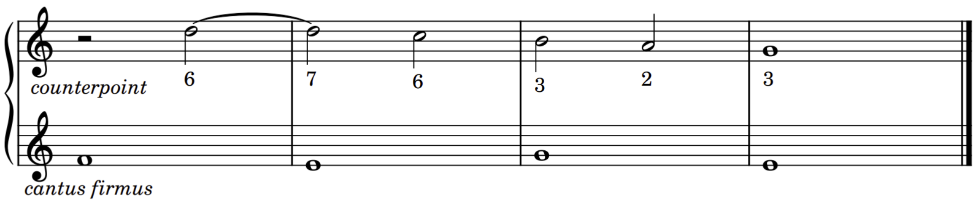


Figure 2.21: Example of allowed chain break in fourth species counterpoint.

6. Must begin on a up-beat (beat 2 of the measure) which forms a perfect consonance to the cantus firmus.

7. If the counterpoint is in the upper voice, the dissonance in the penultimate measure should be a seventh. With the cantus firmus in the upper voice, the suspension of the second is the rule.

2.1.8 Fifth Species

Fifth species combine the rhythms off all the preceding species. This results in a more complex contrapuntal voicing, with the goal being a fluid and smooth result. That increased freedom means that there is more care that has to be made in how the melody flows. As Jeppesen writes; "When melody and rhythm unite the relation becomes very complex and subtle. It becomes increasingly difficult to formulate impressions into rules; they must be held fluid within certain broad limits" [18, p. 135]. Jeppesen still manages to formalize some generalizations, presented in the following rule section.

In fifth species counterpoint, we have access to eight-notes for the first time. Figure 2.22 shows how a perfect fourth, from C to F, can be elaborated with eight-notes in fifth species. The added notes are D and E, and works as double passing notes that fill in the ascending gap. This is similar to how the interval of a third was filled in with a passing note in earlier species. Eight-notes used in this way will always occur on the weak beats of the measure (beats two and four). Example 2.22 uses eight-notes in beat two.

An example of fifth species counterpoint is shown in figure 2.23. Notice how the counterpoint now uses rhythms found in the preceding species, including tied notes and eight-notes.

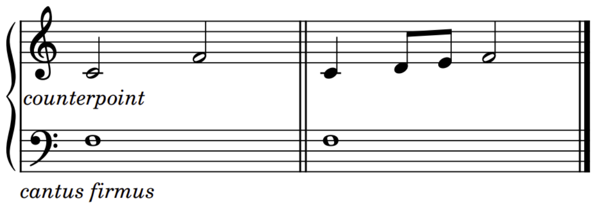


Figure 2.22: Example of melodic elaboration in fifth species using eight-notes.

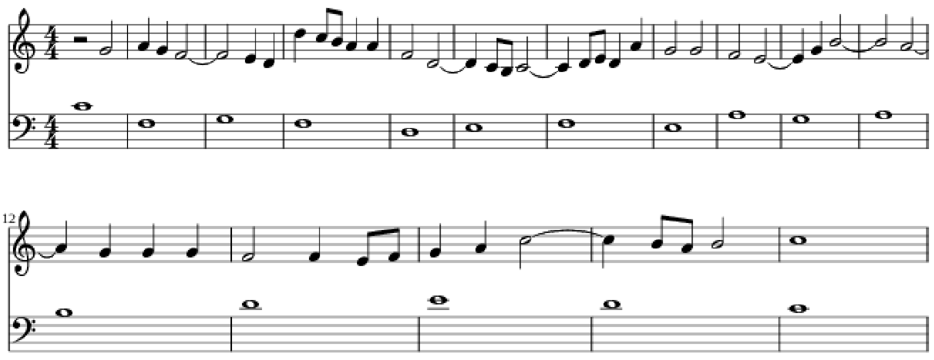


Figure 2.23: Fifth species example with eight-note elaborations and tied notes [10].

Rules

1. No longer limited to one specific rhythm.
2. Rhythms require compensations just like leaps; rhythmic fluency should develop in a continuous way, with slower rhythms developing into faster rhythms, and faster rhythms developing into slower rhythms.
3. Syncopations require some extra care because of its halting effect on the melody. It is therefore common that shorter note values are put immediately before the syncope, and that the syncopated note is followed by eights.
4. The higher tones often have longer note durations, as to emphasis the climax of the counterpoint.
5. In ascending motion, it is common to begin with the quicker notes. The opposite is true for descending movement, where it is more common that the longer note values precede the smaller.
6. Treatment of quarter-notes:
   - Quarter-note movement should, if possible, begin on an unaccented half-note. This rule applies especially to descending movements.
   - A leap followed by step-wise movement of quarter-notes is a natural progression and is often used.
   - The quarter-note movement should continue up to an accented half-note or suspension.
   - The total number of quarter-notes in succession should not be longer than eight.
   - Two quarter notes should, if possible, not stand isolated in the place of an accented half note in a bar.
7. Treatment of eight-notes:
   - Assuming that the cantus firmus moves in whole-notes, no more than two eight-notes should be in succession and in each measure. They can also only appear in metrically weak positions (the second or fourth beat of a 4/4 measure).
   - Eight-notes must be introduced and left by step-wise movement.
   - Since eights may only occur on unaccented beats, they cannot come after a note value greater than a dotted half.
8. Treatment of syncopation:

- The note of least value to be syncopated with another note of equal value is the half-note. This means that a quarter-note can never be tied to another quarter-note.
- It is not allowed to tie notes of less value to subsequent notes of greater value. The opposite may take place, but only in 2:1 relation.
- In the use of dotted half notes the rules for the third species apply to the part of the note that is tied over to the next measure. This means that this part of the note must be treated as either a passing dissonance or a dissonant auxiliary note, and may never proceed upward by skip.

9. Treatment of dissonances:

    - An unaccented half-note that follows after a tie can form a dissonance when the dissonance is treated according to the rules of the second species. The same applies to quarter-notes after ties or dotted half notes. The keyword here is that the note must be unaccented to be able to form dissonances.
    - Tied quarter notes should rarely be used as dissonances if it is not on a weak beat.

2.2 Review of Existing Methods and Software

In general, procedures for music generation can be divided into two main categories; knowledge-based and non-knowledge-based methods [26, p. 270]. The non-knowledge-based methods are characterized by being learning-based. This means that the structure and musical content is not generated based on a predefined rule-set, but learned from training on a data set. This type of music generation has become increasingly more popular over the last decade concurrent with the development of more sophisticated neural networks. These modern, state of the art models incorporate memory, and can therefore represent more abstract musical structures and generate longer, structurally sound musical pieces.

It will also become evident that the diversity of musical dimensions leads to an unavoidable excessive list of different methods and approaches which are possible to use when generating music procedurally. Nonetheless, various approaches are listed in the following two sections, focusing on each of the two main categories: knowledge-based and non-knowledge-based. Since the trend in algorithmic composition in recent years have been dominated by machine learning approaches, we will begin by presenting different approaches within this sub-field of music generation.

2.2.1 Neural Networks and Artificial Intelligence

The main advantage of (artificial) neural networks is its enabling of problem solving by changing a number of weights in a structure of interconnected components. In this way, the network can learn from data sets of different content and structure, resulting in artificial networks being applicable to a number of different sub-fields, such as natural language processing (NLP), image recognition and in our case music generation. The name is derived from its biological model, the interconnection of the neurons in brains [26, p. 205].

Early Stages

The earliest examples of artificial neural networks started to form as early as 1943, when neurophysiologist Warren St. McCulloch and Walter Pitts started to develop models for connectionist structures. This dealt with the reaction patterns in nervous systems. Due to the limitation with computational power during this time, the connectionist model only allowed for the calculation of simple logic functions [24]. Although the first results were quite limited, it did spark the interest in artificial neural networks, leading to the development of the first perceptron in 1958 by Frank Rosenblatt [30]. Early development within the field of artificial intelligence did however come to a halt with the publication of the Lighthill report in 1973, which presented the limitation of perceptrons and current models and criticized AI's failed realisation of its main objective [23]. The publication of this report in addition to a general declining lack of enthusiasm led to a cut in funding for AI research, resulting in a decade long quiet era known as the first AI-winter[2].

Transitional Period (1980s-1990s)

A new wave of enthusiasm and funding happened in the 1980s. This period also saw new applications of artificial intelligence. The classical NLP approach was broadened, and the first examples of music generation using neural nets were realised [26, p. 213]. The first system for music generation using AI was a hybrid approach, developed by Hermann Hild et al. in 1991 using methods based on neural networks and a rule-based system. The system was called HARMONET and its task was to harmonize melodies in the style of Bach chorales [12]. They also explored different options for harmonization, using decision trees and nearest neighbour classification. However, all of these alternative approaches were all outperformed by the neural networks.

The development of music generation using neural nets stagnated after an initial boom during the 1990s. This might be related to the second AI-winter which occurred after the development of expert systems from 1980-1990. The initial ANN approach developed during the late 80s became somewhat of a standard approach up until the deep learning boom from 2006 to 2009. Research during this period include the feed-forward ANN by Todd and Loy in 1989 [33], CONCERT by Mozer (1994), a recurrent autopredictive connectionist network [25], and MELONET I by Hörnel (1997), a music generator in the baroque style of Bach [14].

Current Methodology and Research

The use of deeper neural networks and more computational power has led to an increase in more creative applications using computer-aided music generation. There is no longer a focus on only symbolic generation of creating MIDI or sheet music, but also on performance generation like the new wave2midi2wave system [8]. This system is unique in the way that the symbolic representation being generated is further passed into a performer network that maps the symbols to sound using a model trained on actual live piano performances. In this way the output is much more realistic and human-like, with inferred dynamics in the piano play-

[2]https://www.investopedia.com/terms/a/ai-winter.asp

ing. The training set is the MAESTRO (MIDI and Audio Edited for Synchronous TRacks and Organization) data set, consisting of close to 200 hours of piano performances in both MIDI and audio. By training the model on actual performances, the generated music is more expressive and sound more like human performances, with different timbres and acoustic textures based on microphone placement of the training set recording. This means that sounds like the performer breathing, pedal presses and turning of sheet music paper is also modelled. This further enhances the piano synthesis, creating a realistic rendering of the symbolic representation.

The above-mentioned wave2midi2wave system is one of the latest system by the Magenta project, a google brain research group exploring the role of machine learning as a tool in creative musical processes[3]. Magenta was started in 2016, and has since then published 27 papers in the field of music synthesis, sequencing, audio-to-MIDI and MIDI-to-audio translation and music generation. The research started with an initial basic RNN model to test the Magenta code framework[4].This was further fine-tuned and a technical report was published in 2016 by Natasha Jaques et al. [17]. This model purposed a simple note-RNN for monophonic melodic structure, and served as a good starting point for further research. The model combined machine learning with reinforcement-learning, in an approach to further develop the LSTM structure purposed by Eck and Schmidhuber mentioned above. The reinforcement learning model was used on the LSTM model to try to capture some music theory constraints. The results were satisfactory in that the application of reinforcement learning was able to correct almost all of the targeted "bad behaviours" of the LSTM model, such as note repetition. But even though the results were promising on an objective level (as seen by statistics based on 100,000 compositions from the model) the subjective interpretation of the melodic lines were more varied. The LSTM melodies were more monotone and conjunct, while the melodies generated by the LSTM+reinforcement-learning model were more disjunct and "off". This illustrates that objectively good results does not necessarily mean that the music generated sounds better.

Advancements in polyphonic piano music transcriptions were made in 2018, as seen in the paper published by C. Hawthrones and others [9]. Here, a deep convolutional and recurrent neural network is presented to jointly predict onsets and frames, which means to create a symbolic music representation based on raw audio. The transcription was done on piano music and trained on the MAPS data set, consisting of piano audio and corresponding annotations of symbolic representa-

[3]https://magenta.tensorflow.org/

[4]https://magenta.tensorflow.org/2016/06/10/recurrent-neural-network-generation-tutorial

tion of the piano piece. Hawthorne concludes that the system does a good job in capturing harmony, melody rhythm and dynamics, but that further improvements are limited by the need of a more expansive data set. This work was further developed in the more recent (and above-mentioned) article on wave2midi2wave [8], using the larger MAESTRO data set of 200 hours of virtuoso, live-recorded piano music.

A new breakthrough was made with the emergence of *Transformers*, a new attention-based neural network with increased memory capabilities. Note that it is based solely on attention mechanisms, dispensing with recurrence and convolutions. The initial tests were presented in a paper by A. Vaswani et al. in 2017 [34], with the task being translation from English language to both French and German. Attention-mechanisms was used to better capture global dependencies between input and output. The Transformer architecture also allowed for significantly more parallelization, which is better suited for modern GPUs. The transformer saw an improvement in both computational time and outperformed the best previously reported models, which illustrates the strength of such a model architecture for sequence transduction.

This method was adopted by Magenta in another attempt to capture long-term structure[15]. The reader is encouraged to examine Google Magenta's latest Transformer network [5], to get a feel of the current state-of-the-art music generation using neural networks. Although the results are good in terms of performance and local structure, the generated pieces still lack a global structure, and better resemble piano improvisations rather than isolated pieces.

A more song-based approach has been done recently with Jukebox: a generative model for music in the raw-audio domain [5]. What distinguishes Jukebox from other models is the direct use of audio and not a symbolic representation like MIDI or sheet music. First the audio is processed through a VQ-VAE (a type of variational autoencoder that uses vector quantisation to obtain a discrete latent representation)[27] to compress the audio to discrete codes that is further passed to an autoreggressive transformer model, similar to the one used by Magenta. Since the model is based on pure audio, it manages to represent melody, composition, timbre and human voice singing all in one system. Because of its vast data set with over 1.2 million songs, it also manages to imitate many different styles and artists. What separates this model is the ability to generate pieces that are multiple minutes long. But due to the system being frequency-based, the generated songs

[5]Generated samples of the latest Wave2Midi2Wave architecture can be found here: `https://magenta.tensorflow.org/maestro-wave2midi2wave`

are often somewhat muddled, especially with the synthesized lyrics.

Based on the presentation of existing work some pros and cons with a neural network approach is worth mentioning. The greatest advantage, which to some degree has been mentioned implicitly in this chapter, is that you do not need a deep musical understanding to be able to create decent results. The emphasis is in many cases on constructing a well formed model that captures features from the data set, rather than developing an expert system based on music theory. Using ANN's is also good for performance synthesis (e.g. Wave2Midi2Wave [8]) and translation from audio to symbolic representation and vice versa, as explored in the later works of Magenta.

The main disadvantage is that you often loose some of the control of the generated result. It is difficult to capture heuristic and using meta-information in the music generation. Such information include time signature, key signature and the hierarchical composition of different element that constitutes a musical piece. Since music generation requires models of many different timescales, it is difficult to capture all temporal dependencies within a musical piece with only using neural networks. It is, however, common to decompose the music into discrete note events, which represent different notes in a musical score. The generated music might sound good, but it is hard quantify *why* the model generated the results it did. Another problem is modelling long term structures. As mentioned above, this is an issue that has been in the forefront when developing new models. Creating music with long-term structure (e.g. more than several seconds of structure) is still a very challenging problem, even with modern sophisticated methods [28].

2.2.2 Knowledge-based Methods

The other field of computer-aided music generation can be described as knowledge-based methods or, alternatively, algorithmic composition. Note that ANN's are often also mentioned as part of algorithmic composition, and the distinction between these two fields are therefore somewhat blurry in literature.

In his book *Algorithmic Composition*, Gerhard Nierhaus presents multiple approaches to computer-aided music generation, including historic perspective and a thorough presentation of different algorithms and paradigms used in procedural music generation. Based on the survey of Gerhard Nierhaus, this section presents popular knowledge-based methods and their application in modern systems, starting with a short historic outline.

Historic Perspective

The first music that was generated with the aid of the computer was the *ILLIAC suite* in 1957 by the professors and composers Lejaren Hiller and Leonard Issacson [26, p. 63]. The output was on a symbolic level, consisting of four movements were each movement was a different experiment. The first movement or "experiment" was used to generate a *cantus firmus*, a start-melody to be further harmonized through the means of counterpoint. The latter three movements was for harmonization and further playing instructions by the means of Markov models and other stochastic principles and constraints [13].

Knowledge-based systems has also seen a broader application in the field of computer-assisted composition. In contrast to artificial neural network models, which are primarily used for symbolic generation, performance synthesis and audio to symbol conversion, knowledge-based methods are more applicable to compositional aid given rule-constraints [26, p. 64]. This has led to the emergence of specialized languages for computer music, to aid composers that do not have an extensive technical background. This began with the language of MusicN (first language for audio synthesis). Others include Csound (written in C and still being used today) and SuperCollider [26, p.64]. More modern programs include that of DAW's - Digital Audio Workstations - that are meant to provide a working environment for composers utilizing multiple libraries and programs for computer-aided music composition [19].

The ILLIAC-suite provided a start for computer-aided music generation, and new systems soon followed throughout the 1960s and 1970s. Concurrent with the development of the ILLIAC suite, Iannis Xenakis also developed a system based on Markov models [26, p. 72], were he uses Markov chains to arrange segments or "screens" of different musical density and dynamics. But in contrast to the work of Hiller and Issacson on the ILLIAC-suite, Xenakis was more interested in using his models as tools. That is, to realize computer assisted compositions [26, p. 81]. The output could be used or discarded by the composer as seen fit. Hiller and Issacson however wanted to model the entire compositional process from start to finish (although on a symbolic level) [2, p. 2].

Hiller continued his work on algorithmic composition and developed together with Robert A. Baker MUSICOMP in the late 1950s, the first computer-assisted composition *environment*. The system consisted of a number of subroutines, similar in design to the program that generated the ILLIAC-suite. Given that it was developed as more of a composition environment, it made the process of writing the compositions much easier. Having well defined smaller structures put together to form a larger hierarchy of different musical entities gave the program and the interface with the user much more flexibility, resulting in a rich variety of different generated results [2, p. 3].

Later models include the works of David Cope. He began his work in 1981 with EMI - Experiments in Musical Intelligence. EMI uses transitional networks which are represented in a graph to represent and process musical information. First, the system is tasked to analyse a corpus of a particular musical genre [26, p. 4]. Cope's system first decompose the given composition, before a complex recombination of musical segments at different timescales and levels is applied. Since the system is a transitional network with an exhaustive analysis model based on decomposition, the generated results are classified as style imitations of the style of the corpus.

The advantage of using knowledge-based methods is that the generated results reflect the granularity of the implemented music theory. In this way, the generated results can be evaluated based on what was expected from the knowledge represented within the system. One clear disadvantage, however, is the difficulty in quantifying musical concepts within a program. This often requires a thorough understanding of the musical domain the system should explore and potentially realise. Such systems might therefore be less approachable for developers with limited knowledge regarding music theory.

2.3 Species Counterpoint Used In Automatic Harmonization

One problem that seem to reappear in literature on algorithmic composition is that of autonomous harmonization of a given melody [10, 6, 20, 1, 32]. The reason might be that the rules of Fux [7] can be expressed formally, and can therefore be employed in algorithmic composition without introducing too much non-determinism and ad hoc solutions.

Given this fixed rule-set, there are a number of ways to implement this. Ignoring a rule hierarchy, Komosinksi et al. [20] employs a method of dominance relationship. This allows for analysis of evaluation criteria deduced from the rules of counterpoint, without making any assumptions on the importance of each rule. In this way, aggregations of criteria that would lead to loss of information are avoided. Komosinksi defends this approach by stating that a rule hierarchy often becomes non-specific, with "some rules are more important than others" [7] being the only way to quantify the importance of one rule over another. This quantification is also especially difficult since contrapuntal writing to a large degree is driven by audible preferences over structural preferences. That is; a properly structured counterpoint that satisfy all of the rules might still sound bad to the listener, while a ill structured counterpoint that fails to satisfy some of the constraints might sound better. To avoid this problem, Komosinski and Szachewicz propose a method of dominance relation in order to find the set of best counterpoint for a given cantus firmus, without imposing an importance-hierarchy of the different rules [20]. They are also concerned with the relationship between all of the different possible counterpoints for a given cantus firmus, so their research is also an analysis of the solution space of counterpoints. Their system manages to compose first species counterpoint using this dominance relation. The generated counterpoint melodies is a set of best counterpoints to the given melody, but given the dominance relation they are mutually incomparable.

Another solution that in contrast with the system of Komosinski et al. does impose an importance hierarchy on the rules, is the genetic algorithm of Acevedo used for fugue generation [1]. A fugue subject is the input of the system, and the generated counterpoint melody is the countersubject. The fitness evaluation was based on a sum of weighted features. These features were deduced from the ruleset of Fux. One such feature was for instance that the generated melody had to be in the same key as the input melody. The generated results proved satisfactory. The generated results were evaluated by an external musician, which gave an average grade of 2.94 out of 5, with 5 corresponding to a melody being the work of a musical expert. Despite the results being promising, Acevedo points out a few important steps of improvement. Firstly, the set of features must be expanded to include more of the rules of Fux. Secondly, data-representation of both notes and melodies should be more detailed, making explicit the concepts regarding both measures and beats.

Acevedo also references the system of Bill Schottstaedt, who was one of the first to implement automatic fifth species counterpoint [32]. This was done in 1984 in his thesis on computer research in music and acoustic. Schottstaedt structures his program as a rule based expert system, where the knowledge is encoded as a list of IF... THEN statements. This was done purely procedural and not object-oriented, which led to a large list of checks and rules. To better illustrate the functionality, Schottstaedt also provides the code in its entirety. In contrast with the abovementioned systems, Schottstaedt implements fifth species counterpoint, not only first species. Despite only having a subset of the rules of Fux, the results were satisfactory despite the limited computational power.

A more modern approach to fifth species generation was done by Herremans et al. using a variable neighborhood search algorithm [10, 11]. The system includes a multitude of rules and checks similar to Schottstaedts system. It is, however, structured somewhat differently. All the rules of fifth species counterpoint is encapsulated in an objective function consisting of a set of sub-scores. Each sub-score relates to one rule. These sub-scores are part of one of two categories, where the melodic aspects (horizontal relationship) is grouped into one subclass, and the harmonic aspects (vertical note relationship) is grouped into another subclass. Each subscore were quantified into a value between 0 and 1. The rules that they represents are rules such as “each large leap should be followed by step-wise motion in the opposite direction” and “the climax should be melodically consonant with the tonic”. The search algorithm used was a local search strategy with three different neighborhood searches. First, an initial random counterpoint melody fragment is generated that satisfy all of the hard constraints, before three different local searches with different neighborhoods are performed. The melody fragment is therefore iteratively improved based on the three different local searches, until a valid solution is found.

In addition to search methods presented above, stochastic models can also be used in counterpoint generation [6]. In her analysis and synthesis of Palestrina-style counterpoint, Farbood uses Markov chains to express the relationship between successive notes. This is done both in horizontal and vertical direction. To realize this, transition tables were constructed for each of the different rules. One example of this is the harmonic interval table, with consonant intervals (third, sixth and tenth) having higher probability. The results were satisfactory, but given the multitude of different transition tables, some inter-dependencies between the tables appeared. This resulted in a small change in one table leading to a large change of the quality of generated results. A lot of time therefore went into weighing the probabilities properly in order to get musical results. The end results were however good and comparable to that of student compositions.

This concludes the presentation of methods for automatic counterpoint generation. As can be seen, there are multiple algorithms and paradigms that can be applied to realize systems for generating counterpoint.

2.4 Constraint Satisfaction Program

Constraint satisfaction programming introduces ways to solve problems where the property of the solution can be expressed as a set of rules that it must satisfy. A CSP consists of three main parts [3]:

1. **Variables** - The set of variables to be constrained. These variables can be limited to a domain, or be part of an infinite domain such as all the real numbers in some interval. The values in the domain is usually of the same

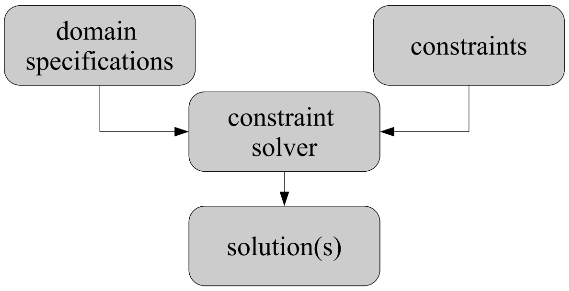


Figure 2.24: Layout of the CSP architecture used in PWConstraints. From [3, p. 18].

type.

2. **Constraints** - Expression of the mathematical relations between the variables. These may in theory be arbitrary mathematical relations, such as logical relations or set relations.

3. **Solver** - The solver finds a valid value to the variable that satisfy the given set of constraints. Since the search space may be huge, having an efficient solver has a great impact on the computational time of a CSP.

The CSP paradigm often appears in systems for modeling music theories and composition. In fact, system such as the above-mentioned fifth species generator by Schottstaedt [32] and the VNS-approach by Herremans et al. [11] is implemented as a form constraint optimization problem. In these examples, the variables are sequences of pitches, the constraints are the quantification of the rules of Fux, and the solver is the different algorithmic approaches to find valid solutions.

What is advantageous with this approach compared to others in algorithmic composition is how compositional rules can so easily be expressed in the world of computer programming. The expression of music theory in the system becomes declarative and modular [3]. By implementing constraints on the solution space instead of defining how to achieve this outcome makes the problem more approachable. CSP also provides a intuitive system architecture which gives a clear modular relationship between the different components. An example of such an architecture is shown in figure 2.24, which is based on the constraint programming language PWConstraints used to solve complex musical problems [22]. In this architecture, the variable domain specification and the constraints are handed independently to the solver, which then returns valid solutions.

2.5 Software Development Tools

This section gives some context to the different choices of software development tools used in this project.

2.5.1 MIDI

MIDI (Musical Instrument Digital Interface) is an industry-standard that defines pitches and pitch durations in a digital music format. This symbolic representation is precise and consistent and has been a standard for the vast majority of electronic audio interfaces since the mid-1980s. MIDI works as a communication protocol that allows computers and synthesizers to control each other and exchange information. This means that an event triggered by, for instance, a key-press on a keyboard can instantiate a predefined drum pattern or sequence. MIDI has also become an integral part of many digital audio workstations (DAWs) since MIDI is just a set of commands that can be manipulated in ways prerecorded audio cannot. This allows for easier multitrack recording and composition and gives the composer more freedom. DAWs with MIDI support, therefore, can quickly change the key, tempo, or instrumentation of a MIDI arrangement [16].

We are, however, more interested in the information stored in its file format, the Standard MIDI File (SMF). SMFs give a standardized way of representing symbolic music information. This information includes the note values, timing and track names. The MIDI-files are also compact, since they do not contain any real audio data. Also, since MIDI has become an integral part in many computer audio system, using SMFs as the file format means that the music generated can be exported and opened in a large variety of different softwares. One such software is *MuseScore 3*, a free music composition and sheet music notation software.

2.5.2 MuseScore 3

MuseScore provides methods for automatic sheet music transcription from MIDI files. Such a transcription is essential for the implemented system since it gives a way to express the generated results, which to a large degree is audible, in a neat and professional printed score. MuseScore can also export the MIDI as audio, sampled with high-quality sound fonts. Despite its many features for musical notation, MuseScore will only be used in this thesis as a utility program to convert the generated MIDIs to either sheet music (.pdf) or audio files (.wav) [6].

[6] https://musescore.org/

2.5.3 Python

The proposed system is implemented in the Python language using the PyCharm IDE. The reason for choosing Python is primarily due to personal preference, as this is the programming language that the author is most familiar with. However, Python does have some advantages over other languages. It has simplified syntax, meaning that the code can be easily written and executed. There are also additional libraries that provide useful extended functionality.

Some of these libraries are used in this project; mainly **matplotlib** for plotting charts and graphs, **random** for generating pseudo-random numbers, and **pretty_midi** to handle the MIDI data[7]. **pretty_midi** proved to be an invaluable part of the MIDI I/O of the system. It is therefore further presented in the following section.

pretty_midi

pretty_midi is a Python library that provides functionality for easy MIDI modification and extraction. The library contains utility functions and classes for handling the MIDI data in a simplified format, with usages in analyzing, manipulating and synthesizing a MIDI file. Examples of such functions include loading a MIDI file into a PrettyMIDI object or instantiating an empty pretty_midi object that can further be loaded with instruments [29].

To better illustrate the functionality, some example code is given below[8]. The code illustrates how to create and export a simple MIDI file:

[7]https://www.python.org/about/
[8]Curtesy of Colin Raffel, https://github.com/craffel/pretty-midi

```
import pretty_midi
# Create a PrettyMIDI object
cello_c_chord = pretty_midi.PrettyMIDI()
# Create an Instrument instance for a cello instrument
c_p = pretty_midi.instrument_name_to_program('Cello')
cello = pretty_midi.Instrument(program=c_p)
# Iterate over note names,
# which will be converted to note number later
for note_name in ['C5', 'E5', 'G5']:
    # Retrieve the MIDI note number for this note name
    n_n = pretty_midi.note_name_to_number(note_name)
    # Create a Note instance for this note,
    # starting at 0s and ending at .5s
    note = pretty_midi.Note(pitch=n_n, start=0, end=.5)
    # Add it to our cello instrument
    cello.notes.append(note)
# Add the cello instrument to the PrettyMIDI object
cello_c_chord.instruments.append(cello)
# Write out the MIDI data
cello_c_chord.write('cello-C-chord.mid')
```

Chapter 3

Software Design

3.1 Early Phase - Formalizing the System Structure

The development phase was a mixture of bottom-up programming and modular implementation. This meant that having a clear system architecture in place before beginning to implement the different modules was crucial, as not to lose vision of the program's scope. As a result, considerations had to be made early in the design to keep the modules as independent as possible, imposing a modular hierarchy in line with the main objective; *to generate counterpoints in each of the five species*.

Previous solutions to the problem as illustrated in section 2.3 shows a large number of different approaches. Schottstaedt [32] structures his program as a knowledge-based system, using IF.. THEN.. statements to formalize the rules and assigning penalties according to the severity of the broken rule. The modern approach of Herremans et. al. [10, 11] uses a similar structure to that of Schottstaedt, with the rules formalized in an objective function used by a search strategy to converge to valid solutions. The approach of Acevedo [1] uses a genetic algorithm as the search strategy, with the rules being formalized as weighted features. Common to all these approaches is the set of three main system components; namely a *cost function*, *search strategy* and *constraint representation*. These three system components were therefore chosen as the main modules in the proposed program in this thesis.

Having decided on the main structure of the program, it was possible to adopt a design paradigm to help formalize the rest of the neseccary system components. By comparing the three proposed modules with different paradigms, one good match was the conventional optimization strategy, consisting of a *objective* *function* and *search strategy*. However, the standard optimization task does not include, formally, a constraint formalism. Therefore, a hybrid design paradigm consisting of both optimization and constraint satisfaction programming was adopted. Representing the system as a *constrained optimization problem* introduced the possibility of including a *constraint formalism* also seen in the above-mentioned systems. The constraint formalism is concerned with how the cost function should be structured based on the underlying constraints on feasible solutions.

While the constrained optimization problem helped to outline the main components necessary, some were still left to be defined. For the solver and cost function to be able to communicate, they needed a mutual representation of the input value and search domain. A formal representation of the search strategy's search space led to the next module: *the music representation*. As the program is tasked with generating both a cantus firmus and a counterpoint melody, two additional modules must be included at a higher abstraction than the music representation, containing the data structures for different species of counterpoint and the cantus firmus.

Lastly, the user must have a way to communicate with the system, giving parameters such as what species to generate and what instruments to be used. There is also the question of the format of the generated counterpoint to be exported. This was chosen to be the general format of a symbolical midi-representation, as this is usable by a large number of external music programs. The last module is therefore tasked with communicating with the other modules to get a generated cantus firmus and counterpoint melody based on user input. The information contained in the music representation must then be converted to midi-format, which is then exported as system output.

Gathering all of the proposed system components, the high level system structure is outlined in figure 3.1. Below follows an overview of the main functionality of each component and how they communicate, in addition to a high level spesification of the neseccary functionality of each module.

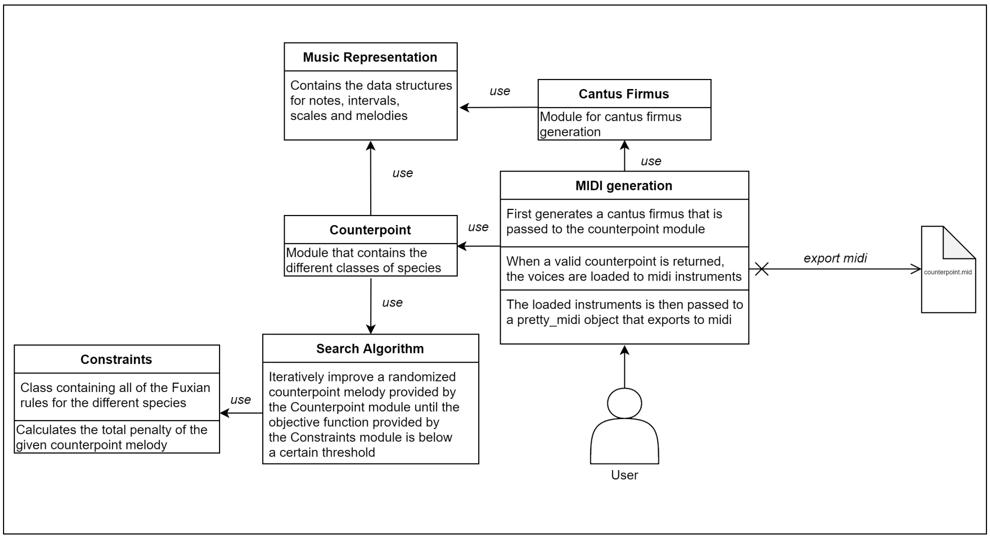

Music Representation
Contains the data structures for notes, intervals, scales and melodies
use
Cantus Firmus
Module for cantus firmus generation
use
use
MIDI generation
First generates a cantus firmus that is passed to the counterpoint module
When a valid counterpoint is returned, the voices are loaded to midi instruments
The loaded instruments is then passed to a pretty_midi object that exports to midi
Counterpoint
Module that contains the different classes of species
use
export midi
counterpoint.mid
use
Search Algorithm
Iteratively improve a randomized counterpoint melody provided by the Counterpoint module until the objective function provided by the Constraints module is below a certain threshold
Constraints
Class containing all of the Fuxian rules for the different species
Calculates the total penalty of the given counterpoint melody
use
User

Figure 3.1: Diagram of the system architecture. The different boxes indicate the different modules. The arrows between the boxes represent the module hierarchy and how the different modules call each other. E.g. the MIDI-generator is the user interface, which uses the cantus firmus generator and counterpoint generator. The cantus firmus and counterpoint modules are therefore at a lower modular hierarchy than the MIDI-generator.

3.2 Choosing the Granularity of the Music Representation

With the outline of the main system structure in place, initial testing and implementation could begin. To formalize the format of the cost function and search strategy variables, the first module to be tackled was the musical description. Music has many dimensions, as is apparent also for algorithmic composition. Choosing an appropriate musical representation is, therefore, crucial for the further scalability of the system. Because of this, the music module was the first to be outlined and design.

The music module must encapsulate the data structures necessary to extract the information required to compose contrapuntal melodies. Care must also be taken in finding the right granularity of music representation. Having too much information can make the musical objects unnecessarily complicated, while having too little information might lead to the need to add help functionality in the other modules. Choosing what information to include is difficult, mainly because the representation must reflect the generation of first to fifth species. As presented in chapter 2, the different species are structurally different. They *traverse different paths of the musical dimensions*. While first species has a trivial rhythm similar to the cantus firmus, one note for each measure, fifth species might have a florid rhythm with tied notes across bar lines. So while rhythmic considerations can almost be abstracted away from first species, instead expressing the melody as a sequence of pitches, this approach is not feasible for fifth species. Therefore, the music representation must *express the melodies of cantus firmus and accompanying counterpoint in a format that is usable for all the different species.* This melodic structure is the primary goal of the music representation module. Such melodic objects are to act as the information passed between the different modules. In this way, the information necessary to represent the music is contained in classes that the various system modules can interpret.

To get a better grip of the necessary granularity of the music module, an analysis of the structure of the different species of counterpoint must be made. Figure 3.2 shows an excerpt of the various species for a given cantus firmus, meant to illustrate the difference between the species. Below follows a brief overview of what information regarding the musical dimensions that must be included in the music representation module.

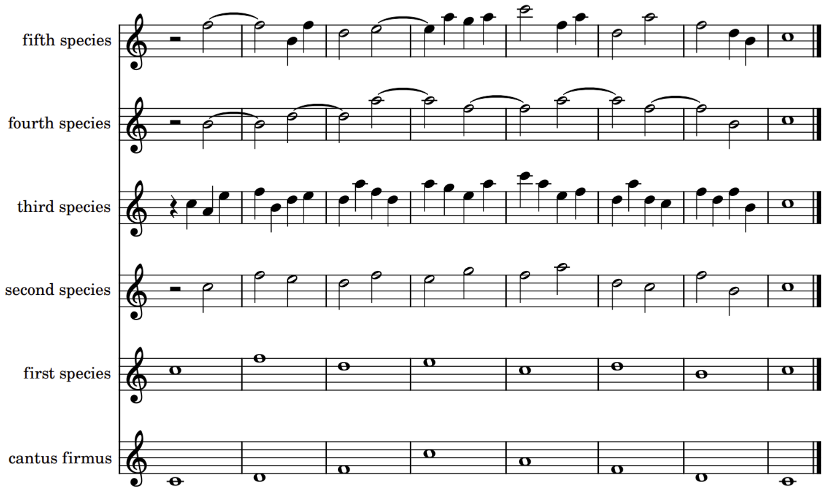


Figure 3.2: Illustration of the rhythmic form of the different species of counterpoint over the same cantus firmus.

Rhythmic dimension

One immediate observation from figure 3.2 is the relationship between the rhythm of first to third species. Second species has, except for the terminals, double the amount of notes as first species. Third species has twice as many notes as second species. Therefore, the *rhythms for species one to three is related by a factor of 2*. The rhythms of fourth and fifth species are, however, more complicated. Fourth species is syncopated, with notes crossing bar lines. But, similarly to species 1 to 3, the note durations are mostly equal across the melody. The last species is more florid, resulting in a more free rhythm with note durations of different lengths and possible ties across bar lines. The music representation must, therefore, *include support for tied notes and varied rhythms*.

Melodic dimension

The pitches of the various counterpoints in figure 3.2 all have something in common; *They are all in the same key and diatonic scale as the cantus firmus*. The melody representation of the system must therefore include some form of pitch constraint given the key and scale in which the counterpoint or cantus firmus melody should be in. Having a constraint on the possible pitches as early as in the music representation itself can also help in reducing the search space of the search algorithm, as the list of possible notes are limited by the fact that they must be the same tonality as the cantus firmus.

Instrumental dimension

As can be observed from the purely symbolical representation in figure 3.2, information regarding what instrument the melodies should be played in can be abstracted away from the musical representation entirely, and rather be handled by other modules such as the midi-generator. In this way, the information contained in the music module is as lightweight as possible, only containing the necessary information needed by the other modules. Therefore; *the musical representation should only be on a symbolical level*.

3.3 Music Module

Specifications

Gathering all of the key points from the preceding section, the music module has the following high level specification:

- The representation should only be on a symbolical level.
- Must include support for tied notes and varied rhythms.
- The possible pitches must be constrained to lie within a given key and range.

- The format of the melody object must be usable by the cantus firmus AND counterpoint representation.

- The information in the music module should not be excessive and overly complicated.

Design

A natural way to structure the music module according to the listed specification is to use an object-oriented approach. In this way, it is possible to express the musical components as mutable containers usable by the other modules, where each object is tasked with containing information regarding one musical concept. Adopting an objective approach also mean that it is possible to structure a musical hierarchy of objects, accumulating in the melody-object, which contains the data structures to represent the cantus firmus and counterpoint.

As to stay true to the proposed granularity of the representation, as well as being expressive enough to be used by the other modules, the following musical classes are outlined. At the lowest level we have a **Note**, which contains information regarding the pitch and duration of a singular note. The next abstraction is a **Interval** class, which consists of two notes and is tasked with identifying differences between two given pitches. As a way to constrain the possible melody pitches to lie within a key and scale, the proposed next abstraction is the **Scale** class, containing functionality for building a list of note objects that lie within a scale. In this way, unwanted non-diatonic pitches can be omitted. Lastly, we have the **Melody** class, consisting of a scale and appropriate data-structures to contain the relevant information regarding a melody. A figure illustrating the high level music topology is shown in 3.3.

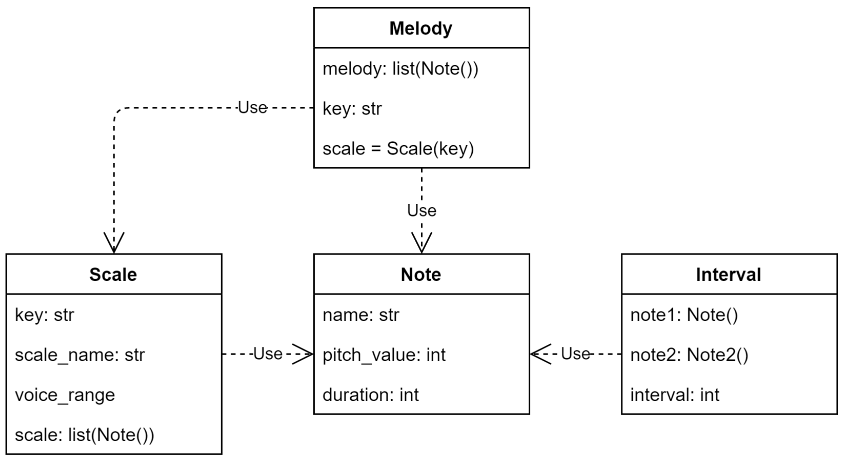


Figure 3.3: High level class diagrams of the different objects proposed to represent the music.

The high level design presented here was done purposely to give flexibility during the implementation of the module. Since the music representation is such a key part of the system, it was excepted that it would be prone to minor alterations throughout the development, concurrent with the implementation of the other modules. The presentation of the implementation in 4.2 is, as an effect, therefore quite thorough. The implementation presents how the different classes are outlined and structured, giving both code examples and more detailed class diagrams.

3.4 Cantus Firmus

The cantus firmus is designed as an extension of the music representation and adds functionality for cantus firmus generation. Figure 3.4 illustrates the overall design. The methods marked with "-" are intended to be private. Methods marked with "+" are public. As shown in the figure, the class contains three main parts; melodic constraints, a generator algorithm, and an initialization method.

The reason for this module structure was a result of keeping the functionality regarding cantus firmus and counterpoint generation clearly divided. Since the cantus firmus works as an input to the counterpoint, having the cf generation as part of the search solver and constraints module would lead to modular interdependencies that should be avoided. We did, however, perform some minor tests to explore the possibility of incorporating the cantus firmus generation into the constraints and search strategy modules. Although possible, it was discovered that it would mean writing the rules of generation with several exceptions since the building of counterpoints and cantus firmi, while similar, have some subtle differences that require different approaches. While the cantus firmus must be generated from scratch, the counterpoint is more comparative given its dependency on the given cantus firmus. A deliberate design choice was therefore made to keep the counterpoint and cantus firmus generation contained within separate modules.

This design choice meant in turn that all of the needed functionality for representing, constraining and solving the COP task for a cantus firmus must be contained within the module. As a result, an object-oriented approach is proposed to subdivide the needed methods into their respective categories without making the functionality accessible to the other modules. An object-oriented approach also meant that the melody class could be easily extended, avoiding the need to restate already implemented functionality. The design of the constraints, generator algorithm, and cantus firmus initialization is now presented separately.

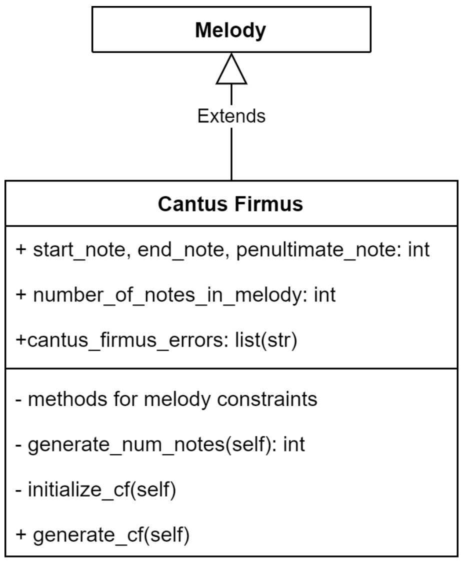

Melody
Extends
Cantus Firmus
+ start_note, end_note, penultimate_note: int
+ number_of_notes_in_melody: int
+cantus_firmus_errors: list(str)
- methods for melody constraints
- generate_num_notes(self): int
- initialize_cf(self)
+ generate_cf(self)

Figure 3.4: Main functionality of the cantus firmus class.

Cantus Firmus Initialization

By analyzing the rules presented in section 2.1.3, one can observe that the list of initial note possibilities for a cantus firmus can be limited. There are mainly four points that is worth addressing; the possible start notes, end notes, penultimate notes and length of the cantus firmus. Given the importance of emphasizing which key the cantus firmus is in, the start and end note can only be the tonic (the first scale degree). The penultimate note can only be one of two possibilities; a minor second below the tonic or a major second above. When these notes are identified, the length of the cantus firmus is generated. As by rule nr 12 in section 2.1.3, the number of notes must be between 8 and 14. The start, end and penultimate notes are pre-set in the initialization of the cantus firmus, while the notes between the start note and penultimate notes are randomly initialized. For a cantus firmus in C-major alto range of length 8, a initialization might look like this;

```
cf = ["C",random,random,random,
       random, random, "D","C"]
```

Here, *C* is the tonic and must therefore be the start and end note. *D* is a major second above *C*, and is therefore acceptable as the penultimate note.

Constraints	**Penalty**
Octave leap	severe
Large consecutive leaps	minor
Large leaps in opposite direction	severe
Large leap not recovered	minor
Too many large leaps	severe
Note repetition	severe
Is not within valid vocal range	severe
Repeats motifs	minor
No clear climax point	severe
Dissonant intervals	severe
Leading tone not resolved	severe

Table 3.1: List of all the constraints issued on the generated cantus firmus, based on the list of rules presented in 2.1.3.

Constraints

The cantus firmus is rarely good enough after initialization, and must therefore be constrained and iteratively improved until a valid solution is found. To achieve this, a constraint formalism is proposed. Initial tests were, however, made to generate these melodies from scratch. Although possible, this approach did lead to an overly detailed constraint description, since a lot of temporal information about the CF had to be known pre-generation. This information included a rough melody contour, and how the melody should approach and leave the climax point. However, by issuing the constraints *after* generation instead of *before*, the problem became much more approachable.

Based on the rules presented in section 2.1.3, the initial set of constraints to be implemented is shown in 3.1. The associated severity of each constraint is a result of the wording of the corresponding rules as presented by Fux and Jeppesen. If the rule states that it "should be avoided if possible", it is defined as a minor penalty. The minor penalties may be accepted if it means that a severe constraint gets satisfied as a result.

Generator Algorithm

The generator algorithm iteratively improves upon the randomized solution until the global penalty given by the constraints is under a certain threshold. Originally, this was implemented as more of a brute force approach given the limited constraints. However, under stress testing it was discovered that this approach in

some cases could halt the run-time quite considerably, so the following change was made to optimize the algorithm.
A simple best local optimal search was added between the randomization and constraint check. This algorithm is outlined in figure 3.5. While simple in design, it proved sufficient for the given task since the maximum allowed penalty could be set to 0. In this way, for each generated cantus firmus that is passed to the MIDI-generator, all of the implemented constraints are satisfied. The best local search works by for each possible note in the cantus firmus, picking the solution which results in the minimum amount of accumulated penalty. The generator starts with the first note, traversing the randomized cantus firmus until reaching the last note. If the local search does not converge to a valid solution, the cantus firmus is randomized and another local search is executed.

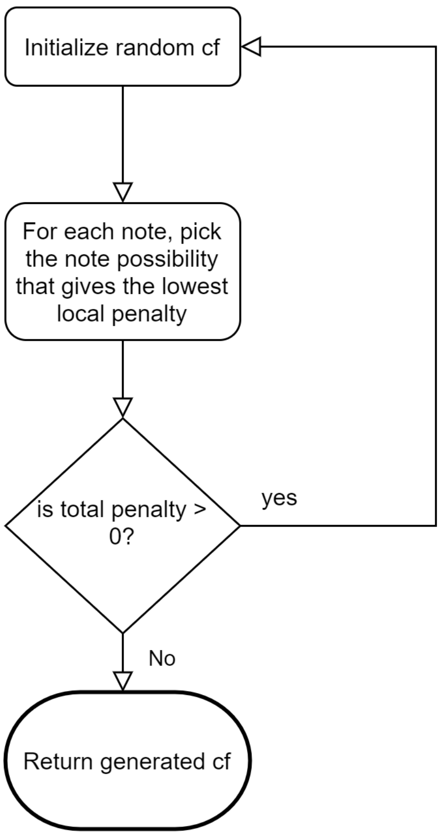

Initialize random cf
For each note, pick the note possibility that gives the lowest local penalty
is total penalty > 0?
yes
No
Return generated cf

Figure 3.5: Main loop of the designed cantus firmus generator.

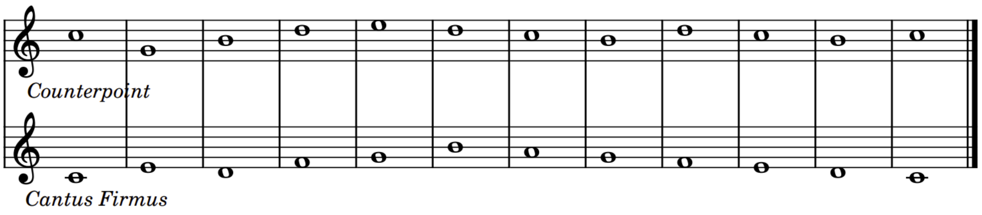


Figure 3.6: Input and output of the counterpoint module. The cantus firmus is given from the MIDI-generator as input, and the counterpoint is returned as output.

3.5 Counterpoint

The counterpoint module must contain information containers for all of the different species. The proposed strategy to achieve this is again by using a object-oriented approach, structuring the different species as classes. The functionality of the various species is to generate a randomized counterpoint draft and a list of possible note values that is passed to the search algorithm. The cantus firmus to be harmonized is provided as input from the MIDI-generator. The cantus firmus is then scanned using methods in the species class, and a list of possible note values is returned. The list of possible notes together with a randomized counterpoint draft is then sent to the search algorithm, which returns a counterpoint with an accumulated penalty below a certain threshold. In other words, **the counterpoint module is to declare for the solver the domain of possible melody pitches for the different species**. By having this module as a interface between the music representation and the solver, the search space for the solver is drastically reduced. If one were to pass the list of possible scale pitches directly to the solver, the total search space of a counterpoint with 12 notes would be $12^{15} = 1.54e16$ different melodies.

Here is an example of how this can be reduced by the counterpoint module. Lets say that the MIDI-generator module provides the generated cantus firmus of length 12 in C major alto range as shown in the lower voice in figure 3.6. The counterpoint module is tasked with finding possible notes for the first species counterpoint above this CF. The counterpoint module then scans the notes in the cantus firmus, and constructs a list of possible values for each of the notes in the cantus firmus. For first species this is trivial, as all of the notes in the counterpoint must be consonant intervals of their respective cantus firmus notes. Further limitation can be done to the start, end and penultimate notes. The start note must be a perfect interval above or below the cantus firmus, while the end note must be a unison or octave. The penultimate note must be a major second above or minor second below the end note, as was the case for cantus firmus. A high level pseudo-code illustration of this functionality (for first species) is shown below, which results in the search domain shown in 3.1.

```
def generate_search_domain:
    search_domain = [None for /...
            all note durations in species rhythm]
    for each note duration in rhythm:
        if start_note:
            search_domain[0] = get_start_notes()
        if penultimate_note:
            search_domain[-2] = get_penultimate_notes()
        elif end_note:
            search_domain[-1] = get_end_notes()
        else:
            search_domain = get_consonant_intervals(/...
                                  corresponding_cf_note)
    return poss
```

Code 3.1: Search domain of first species counterpoint using the cantus firmus presented in 3.6

```
possible_notes = {[C4,G4,C5],[G4,B4,C5,E5,G5],
                  [G4,A4,B4,D5,F5], [A4,C5,D5,F5,A5],
                  [B4,D5,E5,G5,B5],[D5,G5,B5],
                  [C5,E5,F5,A5,C6],[B4,D5,E5,G5,B5],
                  [A4,C5,D5,F5,A5],[G4,B4,C5,E5,G5],
                  [B3,B4],[C4,C5]}
```

which reduces the number of possible solutions to $3^2 * 2^2 * 8^5 = 1179648$. This list of possible notes, together with an initial randomized counterpoint, is then passed to the search algorithm. When a valid solution is found, the first species is returned from the solver and sent as output from the counterpoint module to the MIDI-generator. A proposed solution is shown in the upper voice of figure 3.6.
To achieve the functionality illustrated above, an inheritance hierarchy is proposed in which the interface with the other modules is contained in a super-class, while how the different species of counterpoint is initialized are contained in extended classes which inherits from the super-class. The inheritance hierarchy of the proposed Counterpoint module structure is shown in figure 3.7. In this way, the functionality similar for all the species are contained in the super-class, while isolated species functionality is handled within its respective extended class.

The different classes of species are also tasked with finding the rhythm of the counterpoint to be generated. For first species the rhythm is equal to the cantus firmus since the harmonization is 1:1. Since the fastest note allowed in species counterpoint is an eight note, this is set as the lowest rhythmic resolution. The counterpoint rhythm is therefore expressed as the number of eight-notes that the corresponding note consists of. A whole-note is eight eight-notes, a half-note is four eight-notes etc. For a first species counterpoint with a cantus firmus of length 8, the counterpoint rhythm is simply

```
first_species_rhythm = [8,8,8,8,8,8,8,8]
```

The task of rhythmic generation is, however, not as trivial for the other species. The search algorithm needs note possibilities for each note in the cantus firmus, but for all the species except the first, the number of notes in the counterpoint is higher than the number of notes in the cantus firmus. This issue is handled by ordering the rhythm by *measure*, with each measure containing a set of note durations for that measure. For second species, there are two notes for each note in the cantus firmus. This can be interpreted as two notes for each measure. The rhythm of a second species counterpoint with a cantus firmus of length 8 is therefore:

```
second_species_rhythm = [[4,4],[4,4],[4,4],[4,4],
                         [4,4],[4,4],[4,4],[8]]
```

Notice that the last note is a whole-note. This is to stay true to the rhythmic rules of Fux. By generating a list of possible notes based on each entry in this 2D-rhythm list, it is possible to abstract away the rhythm from the search strategy, since the representation given is only a sequence of pitches. The same search strategy can therefore be used in all of the different species.

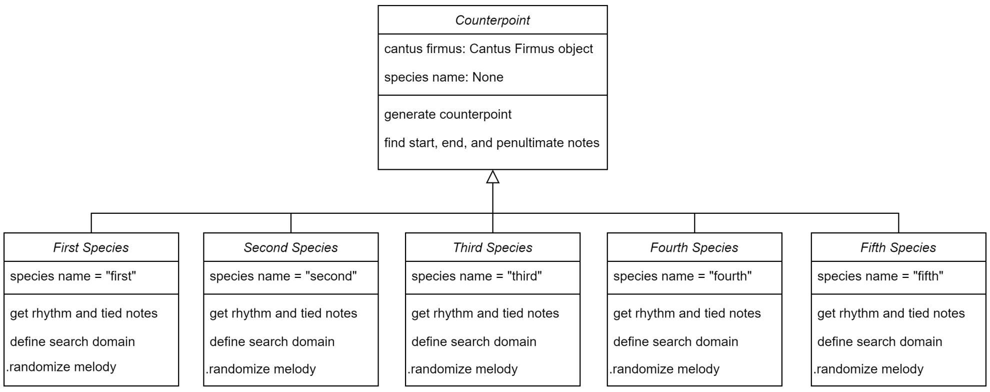


Figure 3.7: Counterpoint class hierarchy.

3.6 Constraint Formalism and Cost Function

The design of the constraint formalism is concerned with two main objectives: quantifying the constraints and choosing the sub-set of Fux rules to be implemented. By inspecting the rules for the various species as presented in section 2.1, the list of rules is quite extensive. There are in all 59 different rules divided among the five species, not including the various rhythmic treatments for fifth species. Implementing all 59 rules would be too immensive for a system of this scope, and a restriction of the rule-set must therefore be done as to limit the complexity of the constraint module. Luckily, some of the rules have already been considered implicitly by how the search domain of possible pitches is defined in the counterpoint module. These are the rules regarding rhythm and tonality, e.g. "there are two half notes for each note in the cantus firmus", and "the counterpoint is diatonic except for the raised leading tone in minor". By removing the already considered constraints, we are left with a total of 41 constraints, which is a reduction by 30%. These 41 explicit constraints are still to excessive, so a sub-set has to be picked out based on their relative importance.

By inspecting the remaining rules for the different species (that it, the rules not considering rhythm), one can observe that many are common for all of the five species. These common rules are the same as those listed for first species, and include constraints such as "no parallel fifths allowed" and "no large leaps except octave and ascending minor sixth". Since the rules of first species forms the basis for the remaining species as well, *all the rules of first species is to be quantified.* For the remaining species, additional rules are added according to the relative importance deduced from the implied severity of breaking the rule as presented by Fux and Jeppesen.

There are in all 22 rules listed for first species in section 2.1.4, in which 19 must be explicitly stated. To keep the constraints organized and maintainable, some further categorization of the rules should be made. This will also aid in the consideration of other species. Many of the rules seem to consider the melodic fluency of the counterpoint. Such rules include "there should be no more than two successive leaps" and "unisons may only occur on the first and last notes of the counterpoint melody". The first defined rule category is therefore the **melodic rules**.

Another sub-set of rules which can easily be categorized are the ones considering the interplay between the cantus firmus melody and the counterpoint melody. Examples of such rules are "upper voices can sometimes cross if necessary, but avoid overlapping" and "all perfect intervals must be approached by contrary motion". Such rules are defined as **voice-independence rules**.

The third rule category is defined based in identifying the main difference between the species. Other than the rhythm which is handled by the counterpoint module, how the dissonances are handled is what seem to separate the different species the most. While first species have zero dissonances, both second, third and fifth species introduces the possibility of having dissonant intervals on weak-beats. Fourth species can have dissonances on strong-beats, which must be resolved by downward step. As a result, the penultimate rule category is **dissonance handling**.

The last category is **harmonic rules**, which consists of possible structural rules that was not handled implicitly in the counterpoint module. As an example, "no outlined dissonance allowed" is one such rule.

For each rule, there must be assigned a penalty relative to the severity of breaking said rule. To quantify the rules in an importance hierarchy, each rule is penalized by how Fux and Jeppesen express the gravity of the rule not being satisfied. Rules expressed as being "forbidden" are categorized as *severe*. Example:

> Rule 12 (2.1.4): *Motion can proceed by step or leap but steps and leaps of augmented and diminished intervals and leaps of any seventh are forbidden.*

Other rules can be interpreted as being unwanted, but acceptable if it means that a severe rule, as a result, gets satisfied. An example of such a "bad" rule is:

> Rule 14 (2.1.4): *following Jeppesen: The repetition of a tone is permitted occasionally in the first species, and there only.*

Rule Type	Penalty
"Forbidden"	severe
"Should be avoided"	bad
"Allowed, but rarely"	minor
"Preferred"	preference

Table 3.2: A list of the penalty categorization based on the wording of the associated rule.

The second to last penalty is the "minor" penalty, which is more prone to subjective preference compared to the preceding penalties. As to guide the generated results to be more lively with a balance of both leaps and step-wise motion, breaking rules regarding how leaps should formally be compensated are categorized as "minor":

Rule 17 (2.1.4): ... *leaps should be followed by inward, step-wise motion.*

The last category is "preferences", which is used to guide the generation in the right direction when given the choice between two almost equal states. One example of a "preference" penalty is the one regarding contrary motion:

Rule 21 (2.1.4): *Contrary motion should be preferred.*

To summarize, the penalty categorizes are listed in table 3.2. Below follows a presentation of the chosen rules and how they are divided among the categories.

Melodic Rules

The melodic rules are very similar to that of the constraints in the cantus firmus module. They include rules such as what leaps that are allowed and not allowed, how successive leaps should be handled and if the melodic range is valid. A complete list of the melodic rules a long with associated penalties are shown in table 3.3.

Species	Melodic Rule	Penalty
All	Large melodic leap	severe
All	Octave leap	minor
All	Leap not compensated	bad
All	Octave not compensated	minor
All	Successive leaps in same direction	minor
All	Invalid successive leaps in same direction	severe
All	Chromatic step	severe
All	Is not within the range of a tenth	severe
All	Repeats pitches	bad / severe
All	No unique climax	severe
>1	Motivic repetition	severe

Table 3.3: List of all the melodic constraints issued on the generated counterpoints.

By inspecting the rules presented in section 2.1.4, the proposed melodic constraints satisfy rules number. 9, 12, 14, 15, 16, 17, 19, 20 and 22. Notice how there is only one additional rule needed to express the melodic rules for the remaining species. This is the rule forbidding motivic repetitions, which is used in second, third, fourth and fifth species.

Voice-Independence Rules

The voice-independence category contains rules regarding the interplay between the cantus firmus and the proposed counterpoint. Examples of voice-independence rules include the law of no parallel fifths or octaves, and if the voices are overlapping. A table of the voice independence rules are shown in 3.4. These rules satisfy rules nr 6, 10, 11, 18 and 21 from 2.1.4, in addition to including the rule forbidding parallel intervals on successive downbeats. This rule is issued on all the species except first.

Species	Voice-Independence Rule	Penalty
All	Perfect interval not properly approached	severe
All	Consecutive perfect intervals not valid	severe
All	Parallel fourths	bad
All	Voice overlappping	severe
All	Voice crossing	bad
All	Not contrary motion	preference
All	Too many equal consecutive intervals	severe
All	Unison between terminals	bad
>1	Paralell perfect intervals on downbeats	severe

Table 3.4: List of all the voice-independence rules issued on the generated counterpoints.

Harmonic Rules

Most of the harmonic rules are already considered in the initialization of the possible notes in the counterpoint module. These pre-constrained rules include that the notes must be diatonic and that the start and end notes must be perfect intervals. As a result, the harmonic rules nr 1-5 and 7 from 2.1.4 are already considered. The only fundamental harmonic rule remaining is therefore nr 13, which forbids the outline of dissonant intervals. Table 3.5 shows the associated penalty of the harmonic rule.

Species	Harmonic Rule	Penalty
All	Outlined dissonant interval	bad

Table 3.5: Table of the harmonic rule and associated penalty.

Dissonance Handling

The last rule-category is dissonance handling, which for the first species is empty. This is due to dissonances not being valid at all for first species, and since the note possibilities are pre-constrained to just be consonant intervals, the dissonance handler has no penalties. For the remaining species, some simplifications are made to the rules presented in section 2.1. For second species, rule 4 is the only one concerning how dissonances are handled. Rule 4 states that the dissonance must be approached and left by step in the same direction. Rule 4 in third species is similar to rule 4 in second species: dissonances are allowed on weak beats. But for third species, the movement is not restricted to be a continuation in the same direction. Also, the figure known as the cambiata is allowed in third and fifth species, which is a special kind of dissonance. The dissonance handling in third species also have many exceptions, such as the figures shown in 2.18. Such rule exceptions are not considered in this system. For fourth species, there are two rules regarding dissonances. Rule 3 states that dissonances may be accented, but rule 4 specify that accented dissonances must be resolved by downward step.

Fifth species, however, have an extensive list of rules regarding how dissonances may possibly be handled. Many of those rules are also concerned with how the rhythm and melody is tied together, which is not the focus of the system proposed here. This is because the rhythm to a large degree have been abstracted away from the search strategy and constraint formalism, as to make the solver usable for each of the different species. As a result, some simplifications are made. The treatment of quarter notes (rule 6 in section 2.1.8) is not considered beyond how quarter notes are handled in third species. Rule 7 (also from 2.1.8) is included in its entirety. This means that eight notes are handled according to the rules of Fux.

After omitting the rules of fifth species most prone to exceptions, the dissonance handler gets the form outlined in table 3.6. Notice how some rules do not have an associated penalty. This is because they check possible dissonance states. For example, if the cambiata figure is recognized, the dissonance is accepted.

Species	**Dissonance Handling**	**Penalty**
2,3,5	Dissonance not properly approached or left	severe
2,3,4,5	Is dissonant interval	N/A
3,5	Is cambiata	N/A
5	Eight notes not handled properly	severe
4,5	Tied notes not properly resolved	severe

Table 3.6: List of all the voice-independence rules issued on the generated counterpoints.

This concludes the presentation of the constraints to be included in the formalism. As can be seen, there are in all 24 that are implemented. All the rules for first, second and fourth species are considered. However, to ensure convergence to solutions under the error threshold, some rules in third and fifth species are relaxed. For third species, these include rule 5, rule 10, rule 11 and 12 presented in section 2.1.6. These have rule exceptions, and since the counterpoint already has limited possibilities, a choice was made to limit the set of initial constraints. However, adding further restrictions can be done under the implementation if the results achieved with the proposed rule-set is unsatisfactory. In fifth species, the rules regarding rhythmic considerations are simplified or omitted. These include rule 2, 4, 5 and 6 from 2.1.8. In effect, therefore, out of the 41 explicit rules listed by Fux and Jeppesen, only 8 are either simplified or not considered.

Cost Function

The cost function is similar in design for all the different species. Each rule category has an associated accumulated penalty for unsatisfied constraints. The penalties are integer values based on the severity of the broken rule. The cost function simply adds all these penalties together, resulting in the *total penalty score* for the proposed counterpoint.

The structure of the cost function is illustrated in figure 3.8. As presented above, each constrain can be divided into one of four categories; melodic, voice-independence, dissonance handling and harmonic. The melodic constraint consists of 12 rules, voice-independence has 9, dissonance handling, including the dissonant state checks, has 5 and harmonic 1. The rest of the rules are considered by the limitations of the search space done in the counterpoint module. The cost function therefore is the accumulated penalty for all of these 27 constraints. Note that there are differences between the rules for the given counterpoints. These differences must be handled in the implementation of the rules.

$$cost\_function = \sum_{i=1}^{12} melodic\_rules + \sum_{i=1}^{9} voice\_independence\_rules + \sum_{i=1}^{5} dissonance\_handling\_rules + harmonic\_rule$$

Figure 3.8: Simple illustration of how the total penalty of a given counterpoint is calculated. The four summations compute the accumulated penalties for each of the different sub-categories.

3.7 Search Algorithm

By adopting a design paradigm based on a *constraint optimization problem*, a high level specification of necessary functionality of the search algorithm can be outlined:

> *The search algorithm must find valid solutions of counterpoints with the accumulated penalty being below a certain threshold. The data-structure to be minimized is given by the counterpoint module as an object. The search algorithm calls the constraints module which returns the total penalty of the given counterpoint object. the search algorithm must perform suitable alterations to the counterpoint pitch*

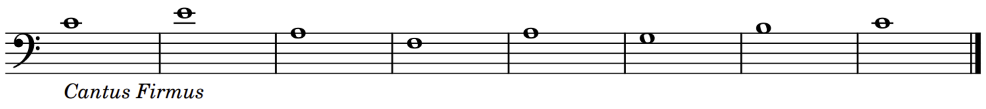


Figure 3.9: Example cantus firmus melody to which a counterpoint should be generated.

sequence until the cost function is below a threshold equalling the integer value of a severe penalty.

With this high level specification in place, we can begin by identifying the different components needed in the search strategy. The counterpoint object given as input contains a randomized pitch sequence of a feasible counterpoint. It is this randomized pitch sequence that must be iteratively improved by the solver. The search algorithm must use derivate-free methods, since the search space is discrete. The possibilities for each pitch slot is also given by the counterpoint object, which in total equals the search domain. To get a better grip of problem domain, the search space for a first species counterpoint in C major is illustrated as a graph in figure 3.10.

Example

an example is given to illustrate the search space and the difference between good and bad sequences of pitches. The counterpoint module is given a cantus firmus in C major as shown in figure 3.9. The cantus firmus has the following pitch representation:

```
cf_pitches = [60, 64, 57, 53, 57, 55, 59, 60]
```

The counterpoint module then establishes a list of possible notes of counterpoint pitches for each corresponding note in the cantus firmus. The search domain is as follows:

```
search_domain = [[60, 67, 72], [67, 71, 72, 76, 79],
                  [60, 64, 65, 69, 72], [60, 62, 65, 69],
                  [60, 64, 65, 69, 72], [62, 64, 67, 71],
                  [62, 74], [60, 72]]
```

The search domain representation and randomized initial sequence of counterpoint notes is passed to the search strategy. For illustrative purposes, the search domain is shown as a directed node network in figure 3.10. The depth of the graph equal the length of the counterpoint. The width of each graph-layer equals the

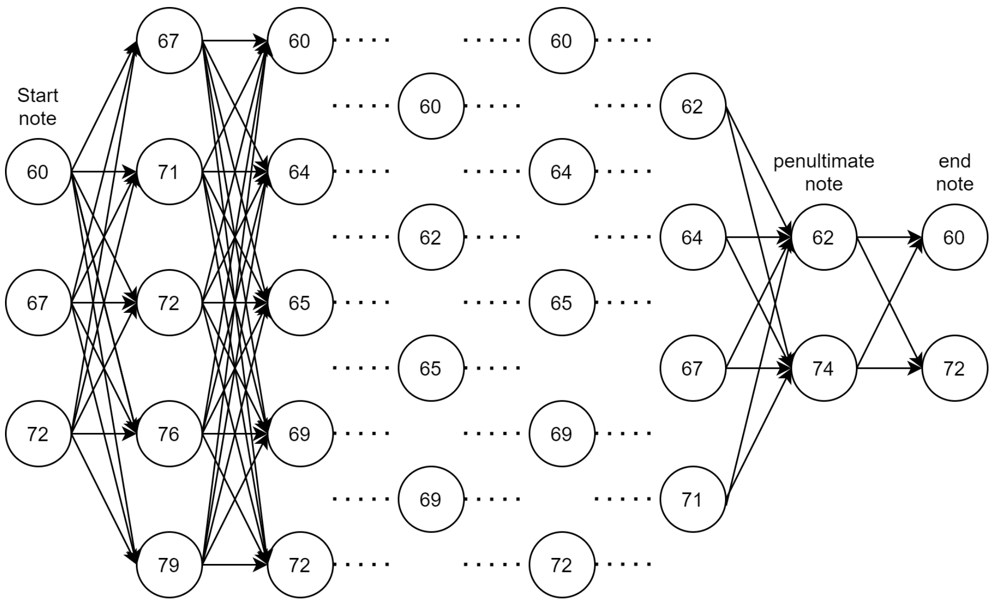


Figure 3.10: The search domain of the counterpoint given the cantus firmus in figure 3.9, illustrated as a directed node network. The node values represent the MIDI-number of the note represented.

number of possible pitches for that pitch slot in the counterpoint melody. To illustrate: the start note can be one of three options given the three nodes at the first vertical layer. The network is fully connected between the layers, with the arrows indicating possible paths from one pitch value to the next. The value of the node indicate the MIDI-number of the represented note.

The randomized initial pitch sequence is a feasible path through the search domain. One valid randomization is therefore:

```
random_counterpoint = [67, 72, 69, 69, 64, 62, 62, 60]
```

This randomized pitch sequence contains errors, such as the pitch repetition between note nr 3 and 4. The search algorithm must therefore *traverse alternative paths in the search domain* until the sequence of counterpoint pitches gives a penalty below the threshold. As the number of paths for first species is limited (24000 in this example), one novice approach would be brute forcing. This is, however, not tractable for the other species, as the number of paths increase exponentially by the number of notes in the counterpoint. The search strategy must therefore utilize more local optimization, as to avoid having to search an excessive amount of different paths.

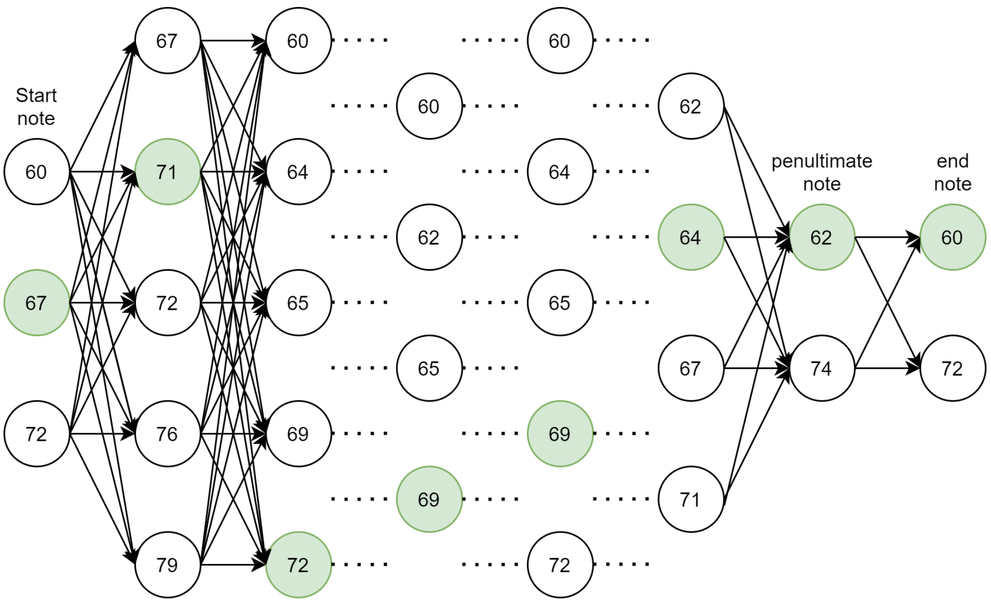


Figure 3.11: The search domain of the counterpoint given the cantus firmus in figure 3.9 with the best first guess highlighted in green. As can be seen, the best first guess might still contain errors.

Instead of randomly iterating through each possible solution, one can observe that each layer in the search domain (see 3.10) has a possible *best local option*. Therefore, by iterating through all of the layers starting with the start note, choosing for each layer the pitch that gives the lowest overall accumulated penalty in the cost function, one can get a best first guess of a possible pitch sequence. Each locally best option replaces the note currently at that position in the given randomized pitch sequence. As an example, a best first search of the search domain illustrated above is done. The best option for each layer, starting with the start note, is marked in green in figure 3.11. The final best first guess option is therefore the path traversed by following the green nodes:

```
best_first_guess = [67, 71, 72, 69, 69, 64, 62, 60]
```

The best first guess may still contain errors, as can be seen in this example. The most noticeable is the pitch repetition between note nr 4 and 5. The search strategy must therefore include functionality for adjusting the pitch sequence by identifying which layer accumulates the most amount of error. In this case, this is layer 4, which leads to the note repetition. By changing the note from 69 to 65, one avoids the note repetition but other errors might have appeared. The search must then again identify the layer with the most accumulated penalty, and adjust the corresponding note in the pitch sequence. By always adjusting the worst note, the search strategy will either eventually converge to a valid option or return the best possible pitch sequence found for the given cantus firmus.

The Proposed Strategy

The proposed search strategy is therefore a *guided local search algorithm*. An initial path-traversing is made through the search domain, for each layer picking the note which results in the minimum amount of accumulated global penalty for the pitch sequence. This best first guess is then adjusted by randomizing the notes which has the highest local penalty. The different layers are then visited in another order than the initial pass, and a new local search is done. If the accumulated global penalty is the same as the previous local search, the number of notes to be randomized according to the worst local penalty is increased. This continues until the global penalty decreases, in which the number of notes to be randomized is again set to one. The proposed strategy is illustrated in the following pseudo-code:

Code 3.2: Pseudocode of the search strategy

```
def guided_local_search(Counterpoint):
    penalty = math.inf
    pitch_sequence = Counterpoint.randomized_ctp_melody
    best_pitch_sequence = pitch_sequence
    lowest_penalty = math.inf
    search_order = [i for i in range(len(pitch_sequence))]
    notes_to_randomize = 1
    while penalty >= ERROR_THRESHOLD and elapsed_time < 5 seconds:
        penalty, pitch_sequence,worst_notes = local_search(/...
                                  pitch_sequence,search_order)
        if no decrease in penalty:
            for note in notes_to_randomize ordered by worst_notes:
                pitch_sequence[note] = random_choice(/...
                                    search_domain[note])
            randomize(search_order)
            if notes_to_randomize != length of pitch_sequence:
                randomize_idx += 1
        if penalty < lowest_penalty:
            notes_to_randomize = 1
            best_pitch_sequence = pitch_sequence
            lowest_penalty = penalty

    return lowest_penalty, best_pitch_sequence,
```

The local search algorithm can also be further specified and outlined in high-level pseudocode:

Code 3.3: Pseudocode of the local search

```
def best_first_search(ctp,search_order):
    for i in search_order:
        local_error = math.inf
        for note in search domain layer:
            pitch_sequence[i] = note
            error = constraints.cost_function(pitch_sequence)
            weighted_worst_notes = constraints.sort_indices()
            if error <= local_error:
                best_note = note
                local_error = error

        pitch_sequence[i] = best_note
        if local_error < best_global_error:
            best_global_ctp = pitch_sequence
            best_global_error = local_error
            best_global_weighted_indices = local_weighted_indices
    return best_global_error, best_global_ctp,weighted_worst_notes
```

As a way to compare the search strategy of the counterpoint melody with the search strategy of the cantus firmus melody, a flowchart is also presented in figure 3.12 to showcase the main search functionality.

3.8 MIDI Generator

The last module is the MIDI generator, which generates the system output and works as the user interface. The MIDI-generator coordinates the generation of a cantus firmus and corresponding counterpoint. The counterpoint species, key, vocal range and if the counterpoint should be above or below the cantus firmus can all be defined by the user.

The MIDI-generator was together with the music representation the first modules to be designed. This was due to the importance of having the possibility of exporting to MIDI early on in the design- and implementation-phase.

Since the module is only tasked with being an interface between the user and the rest of the system, it is quite simple in design. The user provides a set of wanted parameters, and the module does the rest. The main functional flow of the proposed module is shown in figure 3.13. First, a set of parameters is set by the user. Then, the cantus firmus is generated before the accompanying counterpoint is generated.

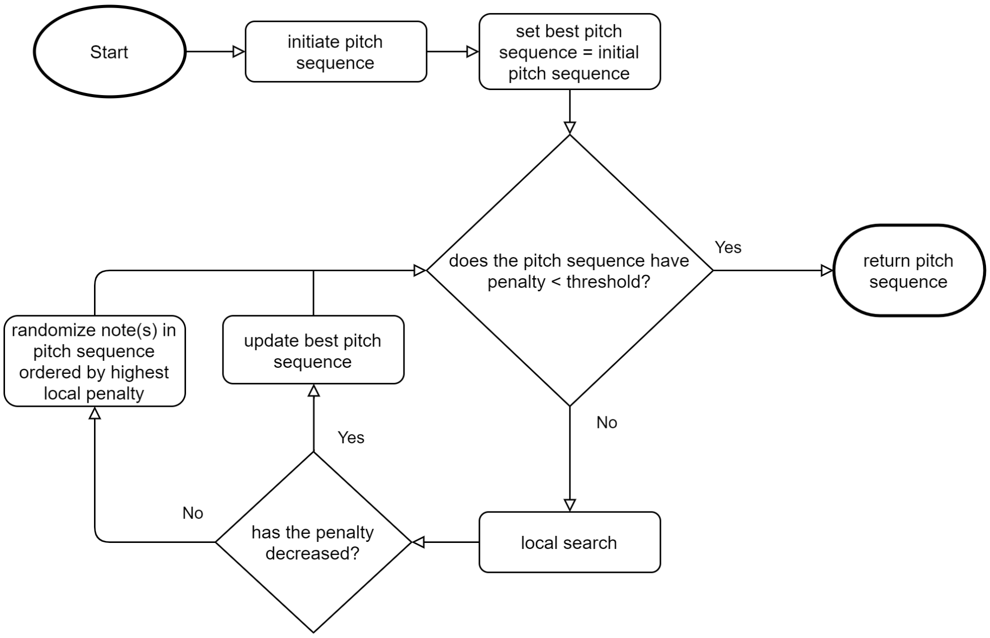

Start
initiate pitch sequence
set best pitch sequence = initial pitch sequence
does the pitch sequence have penalty < threshold?
Yes
return pitch sequence
No
local search
has the penalty decreased?
Yes
update best pitch sequence
No
randomize note(s) in pitch sequence ordered by highest local penalty

Figure 3.12: The proposed main flow of the search strategy. The local search scans the search possibilities in each layer of the search domain following a given search order. The possibility that leads to the least amount of global penalty is chosen, and the pitch sequence is updated with this value.

Lastly, the information contained in the data-structures of the cf and counterpoint melodies is loaded to pretty_midi objects, which then exports the combined cf and counterpoint to a midi-file.

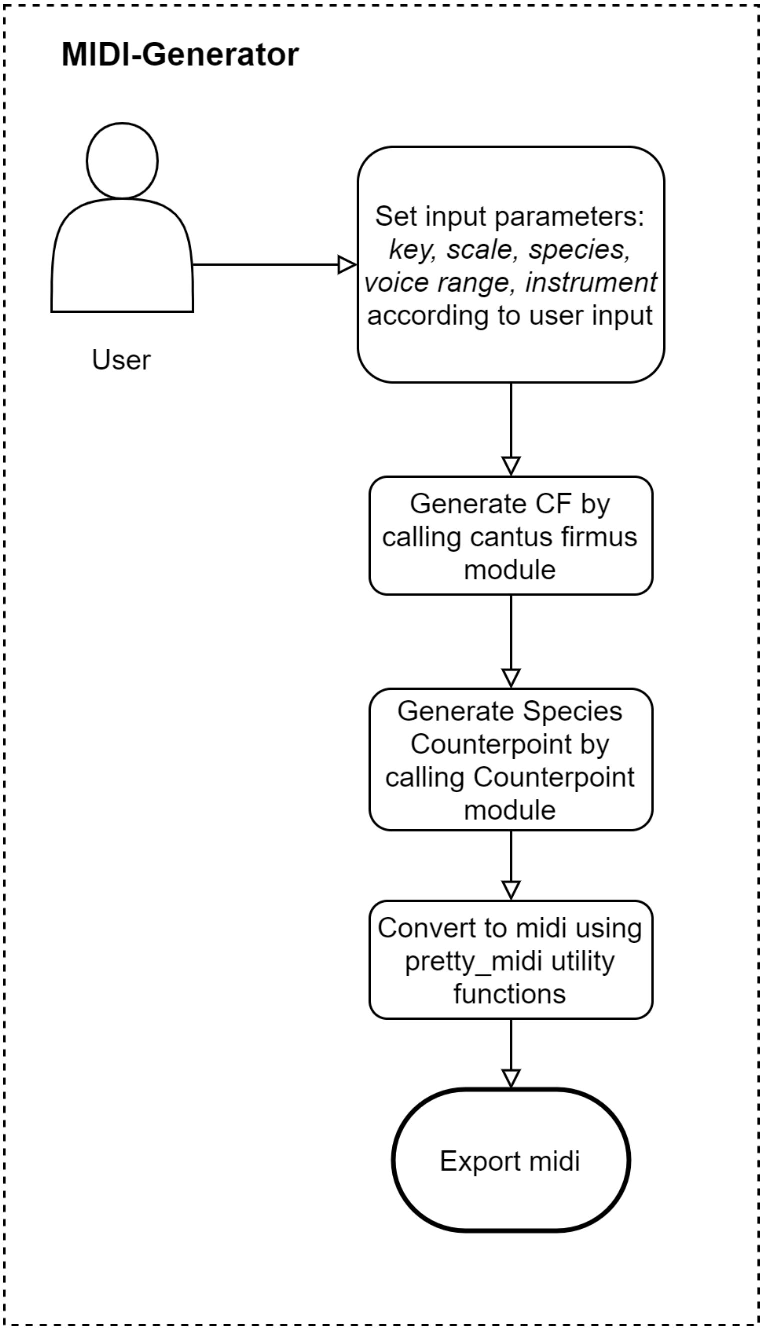

MIDI-Generator
User
Set input parameters: *key, scale, species, voice range, instrument* according to user input
Generate CF by calling cantus firmus module
Generate Species Counterpoint by calling Counterpoint module
Convert to midi using pretty_midi utility functions
Export midi

Figure 3.13: Main functional flow of the MIDI-generator.

Chapter 4

Implementation

In this chapter, the implementation of each individual system module is presented. The source-code is made available online, and can be found here: `https://github.com/JohanGHole/AutomaticCounterpoint`.

4.1 Assumptions

1. The different notes are identified by their corresponding midi-numbers. This means that each pitch value is assigned an integer value between 0 and 127. Examples include the number 60 representing middle C (C4) and 62 representing middle D (D4).

2. The rhythmic resolution is eight-notes as this is the fastest note that is allowed in species counterpoint.

3. Dynamic playing is not considered in this implementation. Therefore, the MIDI information that identifies how hard the note should be played of each note object is set to 100.

4. Possible counterpoint tones for a given note in the cantus firmus can be expressed as a list of midi-numbers. In this way, it is possible to implement one search algorithm that can find solution for all the different species.

5. We have chosen to not use the church modes, instead focus on generating species counterpoint for all the different minor and major keys. This deviates from the style of Fux, but it does make the counterpoint more tonal. Since major and natural minor corresponds to the church modes *Ionian* and *Aeolian*, respectively, this decision does not have major impact on the contrapuntal style.

6. Melodic fluency is prioritized over rhythmic fluency for fifth species. This means that rhythmic considerations, beyond leading to a sufficient melody, is not required.

4.2 Musical Representation

We begin with the implementation of the musical representation. This is due to the objects defined here acting as the information passed between the other modules. The music module is tasked with representing the musical concepts necessary to express the search domain and format of the variables used in the cost function and search algorithm. Therefore, the objects must contain sufficient information while still keeping it as simple as possible. As presented in the design chapter, the lowest level of musical granularity is the **Note Object**. But to have a common ground for all the musical objects, some constants must first be defined.

Constants

Before implementing the musical objects, we must first define some common terms. This include data-representation of the different key names, represented as a list of strings:

```
KEY_NAMES = ['C', 'Db', 'D', 'Eb','E', 'F', 'Gb',
             'G', 'Ab', 'A', 'Bb', 'B']
KEY_NAMES_SHARP = ['C', 'C#', 'D', 'D#', 'E', 'F','F#',
                   'G', 'G#', 'A', 'A#', 'B']
```

The different intervals also need a token representation as to clarify what interval corresponds to what integer value representing the number of semitones in said interval. This is more easily explained with an example. A perfect fifth consists of 7 semitones or half-steps, and therefore has an integer value of 7. The abbreviation of a perfect fifth is set to P5. Minor and major intervals are notated "m" and "M", respectively. Diminished intervals are notated with "d", and augmented intervals "aug". In addition, some intervals have other tokens as well, such as diminished fifth equaling a tritone. The representation of the intervals from unison to perfect fifth is illustrated below.

```
P1 = Unison = 0
m2  = 1
M2 = 2
m3  = 3
M3 = 4
P4 = 5
```

```
d5 = Tritone = 6
P5 = 7
```

It is worth to mention that this is only an illustration of how the intervals are named, and not the extensive list. The extensive list contains all the intervals within an octave.

Since sequences of pitches are represented as a list of corresponding midi-values, it is also possible to express the vocal ranges as a sequence of midi-numbers. As an example, let us consider the vocal range *alto*. The alto range is typically the notes between F3 and F5, which has the corresponding midi-numbers 53 and 77. The alto range representation is therefore:

```
ALTO_RANGE = [53, 54, 55, 56, 57, 58, 59, 60, 61,
              62, 63, 64, 65, 66, 67, 68, 69, 70,
              71, 72, 73, 74, 75, 76, 77]
```

All the vocal ranges are further represented in a 2D list of length 4, from lowest to highest register. The **BASS** voice is therefore the first entry, and the **SOPRANO** voice the last. The **ALTO** voice is the second highest range, and can therefore be expressed as the third entry in the 2D **RANGES** list:

```
ALTO_RANGE = RANGES[2]
```

The constants also include dictionaries for scale representation within an octave, and a representation of the different species names. The scales are either minor or major, and expressed as tuples with instructions on how to build the scale within an octave. The numbers in the tuple illustrates how many semitones there are between the corresponding notes in the scale:

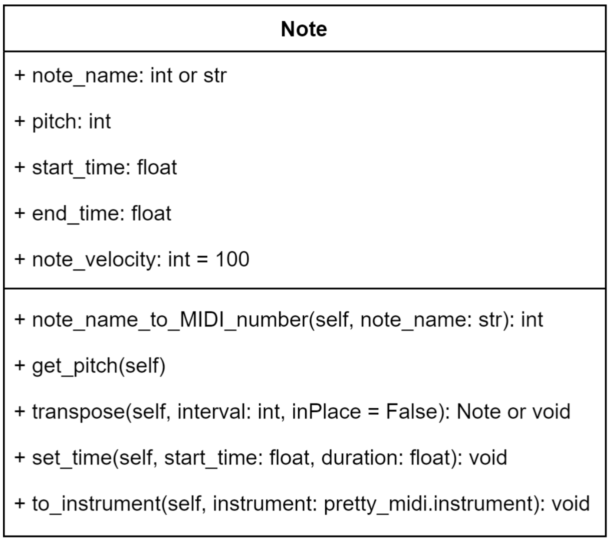
```
NAMED_SCALES = {
    "major": (2, 2, 1, 2, 2, 2, 1),
    "minor": (2, 1, 2, 2, 1, 2, 2),
}

SPECIES = {
    "first": 1,
    "second": 2,
    "third": 3,
    "fourth": 4,
    "fifth": 5,
}
```

Lastly, the list of constants include a categorization of the different intervals. This representation is used extensively in the rest of the system. Notice how the melodic intervals have negative entries. This is because they are used to check consecutive intervals within a melodic line, and can therefore be negative. The harmonic intervals are always computed by subtracting the lower voice from the upper voice, meaning that they are always positive. If not, we have voice crossings, which is not allowed. The different categorizes are illustrated below, which conclude the set of system constants.

```
MELODIC_INTERVALS =[Unison,m2,M2,m3,M3,P4,P5,m6,P8,
                   -m2,-M2,-m3,-M3,-P4,-P5,-P8]
HARMONIC_DISSONANT_INTERVALS = [m2,M2,P4,M7,m7,
                               P8+m2,P8+M2]
HARMONIC_CONSONANCES = [m3,M3,P5,m6,M6,P8,
                       P8+m3,P8+M3]
PERFECT_INTERVALS = [P5,P8]
```

Note Object



```
Note
+ note_name: int or str
+ pitch: int
+ start_time: float
+ end_time: float
+ note_velocity: int = 100
+ note_name_to_MIDI_number(self, note_name: str): int
+ get_pitch(self)
+ transpose(self, interval: int, inPlace = False): Note or void
+ set_time(self, start_time: float, duration: float): void
+ to_instrument(self, instrument: pretty_midi.instrument): void
```

Figure 4.1: Diagram of the Note class.

Notated music consists of successive notes both vertically and horizontally arranged on a staff. There are also other things needed to express music symbolically, like clefs, key signatures and time signatures. But at the lowest level we have the singular note, which is a good place to start to structure the object-oriented hierarchy of musical classes. The system's container for singular note information is contained in the **Note Class** located in the music module. This object is tasked with both containing note information but also contains a lightweight wrapper to pass the note information to a pretty_midi object. This makes it possible to use some helper functions from the pretty_midi library for midi export. To better illustrate how the note object should behave, we give an example. First, the object must be instantiated. We begin with a common note, the middle C:

```
middle_C = Note("C4")
```

We here use the convention where middle C is notated C4, with C representing the note name and 4 representing the octave. We have now instantiated the note object, and a set of attributes are now initialized. These include *pitch* (midi-number) and note *velocity*. To support simple midi conversion, the attributes *start_time* and *end_time* are also included.

A set of class methods must also be included to support the construction of the note object in addition to alterations such as transpose:

```
middle_D = middle_C.transpose(2, inPlace = False)
```

If inPlace = True, the original note is changed. If inPlace = False, a new note object is returned that represent the transposed note of the original note object. In the example above, the transpose method returns a new note two semitones above the already constructed middle C note.

The note object is implemented as dictated by the class diagram in figure 4.1. The class attributes are set according to the inputs given to the constructor, except for *pitch* which can either be given in string format or MIDI-numbers. Two valid instances of note objects that both represent middle C is therefore:

```
C_str = Note["C4"]
C_midi = Note[60]
```

The constructor is implemented as follows:

```
class Note:
    def __init__(self, note_name, start=None,
                 end=None, velocity=100):
        if isinstance(note_name,str):
            self.pitch = pm.note_name_to_number(note_name)
        elif isinstance(note_name,int): # MIDI-number
```

```
            self.pitch = note_name
        else:
            print("error: Wrong pitch"+
                 "format in Note class")
        self.start = start
        self.end = end
        self.velocity = velocity
```

Notice the use of the help function *note_name_to_number* from the pretty_midi library. The *note_name* input is converted to its corresponding midi-number, and the rest of the attributes are set according to the corresponding inputs.

The implementation of the class methods was straight-forward. All the methods are therefore listed in the code shown below. Notice especially the implementation of the *to_instrument* method. Here, the information contained within the self-implemented class is passed in a format that is interpretable by the pretty_midi library. In this way, the note can be added to pretty_midi instruments which then can be exported as midi.

```
    @Classmethods
    def get_pitch(self):
        return self.pitch

    def set_time(self, start, duration):
        self.start = start
        self.end = start + duration

    def get_duration(self):
        if self.start == None or self.end == None:
            return None
        else:
            return self.end - self.start

    def transpose(self, i, inPlace = False):
        if inPlace:
            self.pitch += i
        else:
            return Note(self.pitch+i,self.start,self.end,
                        self.velocity)

    def to_instrument(self, instrument):
        # adds the note to the given instrument
        if self.start == None or self.end == None:
            print("Error: no temporal"+
                   "information is given for the Note")
            pass
        note = pm.Note(velocity=self.velocity,
                       pitch=self.pitch,
                       start=self.start, end=self.end)
        instrument.notes.append(note)
```

Interval Class

The next abstraction level is the **Interval Class**, which encapsulates the relationship between two note objects. This will prove useful in the constraint formalism, as the interplay between different voices is given to a large degree by the intervals between them. The methods in this class must therefore include functionality to distinguish between the different types of intervals discussed in chapter 2.1. The interval class is instantiated as follows: two note objects are given as input. The constructor will then save the interval as the distance in number of semitones between the two given notes:

```
Interval(note1, note2)
```

The majority of the Fuxian rules are concerned with the interval relationship between two notes. Therefore, the interval class must also include methods for interval-analysis. As to ensure that the interval class structure is preserved as a container, these methods are limited to be boolean expressions to check if the interval is perfect, consonant or dissonant. An overview of the main functionality is shown in class diagram 4.2.

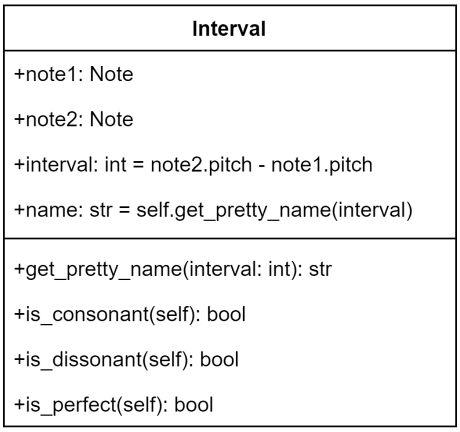

Interval
+note1: Note
+note2: Note
+interval: int = note2.pitch - note1.pitch
+name: str = self.get_pretty_name(interval)
+get_pretty_name(interval: int): str
+is_consonant(self): bool
+is_dissonant(self): bool
+is_perfect(self): bool

Figure 4.2: Diagram of the interval class.

The constructor follows the class diagram, and the implementation is shown below.

```
class Interval:
    def __init__(self, note1, note2):
        self.note1 = note1
        self.note2 = note2
        self.interval = self.note2.pitch - self.note1.pitch
        self.name = self.get_pretty_name(self.interval)
```

The *get_pretty_name()* checks the integer value of the interval and returns the corresponding string name. The implementation, slightly abbreviated, is shown below. For each interval within a single octave, a corresponding string name is given. If the interval is larger than an octave, the name is given as "compound" + how much the interval exceeds an octave.

```
    def get_pretty_name(self,interval):
        i = interval
        pretty_name = ""
        if i == 0:
            pretty_name = "unison"
        elif i == m2:
            pretty_name = "minor second"
        .
        .
        .
        elif i == Octave:
            pretty_name = "octave"
        else:
            pretty_name = "compound"+
                        self.get_pretty_name(interval-Octave)
        return pretty_name
```

Lastly, we have the analysis part of the interval class. These methods are boolean expression that checks if the interval is within a certain category.

```
    def is_dissonant(self):
        if self.interval in HARMONIC_DISSONANT_INTERVALS:
            return True
        else:
            return False
    def is_melodic_consonant(self):
        if self.interval in MELODIC_CONSONANT_INTERVALS:
            return True
        else:
            return False
    def is_consonant(self):
        if self.interval in HARMONIC_CONSONANCES:
            return True
        else:
            return False

    def is_perfect(self):
        if self.interval in PERFECT_INTERVALS:
            return True
        else:
            return False
```

Scale Class

The scale object constructs a list of all the possible notes in a given scale. The scale class is meant to help the higher order **Melody Class** to constrain the set of possible notes. A scale object is instantiated as follows:

```
Scale(key, scale)
```

Where key is the key signature and scale is the name of the scale to be generated. A valid instance of the object is therefore:

```
C_major = Scale("C", "major")
```

Which will contain note objects for all the different tones in a C major scale that is possible to sound on a 88 key piano. The main functionality is illustrated in figure 4.3. The *intervals* attribute is a tuple containing the intervals to form the given scale from root position over one octave. This tuple is equal to one of the two NAMED_SCALES ("major" or "minor") declared in the constants file.

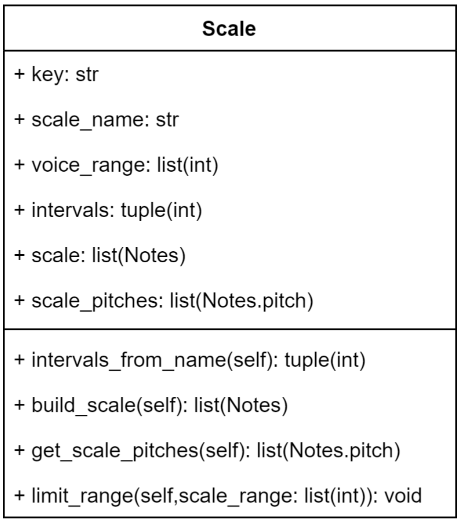

Scale
+ key: str
+ scale_name: str
+ voice_range: list(int)
+ intervals: tuple(int)
+ scale: list(Notes)
+ scale_pitches: list(Notes.pitch)
+ intervals_from_name(self): tuple(int)
+ build_scale(self): list(Notes)
+ get_scale_pitches(self): list(Notes.pitch)
+ limit_range(self,scale_range: list(int)): void

Figure 4.3: Class diagram of the scale class.

As can be seen in the methods part of the class diagram 4.3, the class also contain functionality to limit the scale to lie within a given vocal range. This is more in line with the restriction that Fux imposes on the possible melodies, since they *must* lie within their respective ranges.

The class constructor for the scale object is more complicated than the preceding classes. First, the *key* parameter must be checked to see if it is valid. In addition, given the layout of a 88 key piano, the lowest note in different scales may be in different octaves. The keys "A", "A#", "B" and "Bb" all have their lowest note in octave 0, while the remaining keys have their lowest note in octave 1. This is also handled in the constructor, as illustrated below. The rest of the functionality of the constructor is to *build* and possibly *limit* the scale to lie within a given range:

```
class Scale:
    def __init__(self, key, scale, scale_range=None):
        if key[0].upper() not in (KEY_NAMES_SHARP or KEY_NAMES):
            print("Error, key name not valid!"+
                  "Try on the format 'C' or 'Db' ")
            pass
        if key in ["A", "A#", "B", "Bb"]:
            oct = 0
        else:
            oct = 1
        # sets the root of the scale as a note object
        self.root = Note(key + str(oct))
        self.key = key
        self.scale_type = scale
        if isinstance(scale, str):
            scale = Scale.intervals_from_name(scale)
        elif isinstance(scale, Scale):
            scale = scale.intervals
        self.intervals = tuple(scale)
        self.scale = self.build_scale()
        self.scale_pitches = self.get_scale_pitches()
        if scale_range != None:
            self.limit_range(scale_range)
```

Help methods are also implemented to aid in the scale construction. These are illustrated in figure 4.3. The *intervals_from_name(scale_name))* method finds the tuple declared in the constants that corresponds with the given scale name. The *build_scale()* method constructs a sequence of note objects that lie within the scale. Lastly, the *limit_scale()* limits the range of the scale according to the vocal ranges declared in the constants-file. The implementation of these three methods is illustrated below.

```
    @classmethod
    def intervals_from_name(self, scale_name):
        global NAMED_SCALES
        scale_name = scale_name.lower()
        # support for alternative formatting..
        for text in ['scale', 'mode']:
            scale_name = scale_name.replace(text, '')
        for text in [" ", "-"]:
            scale_name = scale_name.replace(text, "_")
        return NAMED_SCALES[scale_name]

    def build_scale(self):
        start_pitch = self.root.get_pitch()
        scale_len = len(self.intervals)
        highest_pitch = 108 # MIDI-number for C8
        lowest_pitch = 21   # MIDI-number for A0
        j = 0
        scale = []
        pitch = start_pitch
        # adds all possible values above the root pitch
        while pitch <= highest_pitch:
            scale.append(Note(pitch))
            pitch = scale[j].get_pitch() +
                    self.intervals[j % scale_len]
            j += 1
        # adds all possible values under the root pitch
        j = scale_len - 1
        pitch = start_pitch - self.intervals[j % scale_len]
        while pitch >= lowest_pitch:
            scale.insert(0, Note(pitch))
            j -= 1
            pitch = pitch - self.intervals[j % scale_len]
        return scale

    def limit_range(self, scale_range):
        scale = []
        for notes in scale_range:
            if notes in self.scale_pitches:
                scale.append(Note(notes))
        self.scale = scale
```

The last code snippet below show the implementation of the remaining help function. The *to_instrument(instrument)* methods load each note object in the scale to the given pretty_midi instrument. In this way, it is possible to export whole scales to MIDI. The *set_time(duration)* method set equal durations for each note object in the scale list.

```
    def get_scale_pitches(self):
        scale_pitches = []
        for notes in self.scale:
            scale_pitches.append(notes.get_pitch())
        return scale_pitches

    def get_scale_range(self, scale_range):
        scale_pitches = []
        for notes in scale_range:
            if notes in self.scale_pitches:
                scale_pitches.append(notes)
        return scale_pitches

    def set_time(self, duration):
        t = 0
        for notes in self.scale:
            notes.set_time(t, duration)
            t += duration

    def to_instrument(self, instrument):
        for notes in self.scale:
            notes.to_instrument(instrument)
```

Melody Class

To better illustrate the needed functionality in this class, we will now sketch a simple example. The main functionality of the *entire system* is to generate a cantus firmus and a counterpoint harmonization of said cantus firmus. Let us say that the cantus firmus module wants to generate a 12 note melody in C major. What information does the CF generator need to be able to create such a melody? It is this information that must be contained in the melody class. In this example, the token "C" and "major" indicate the key and scale type of the melody, respectively. Therefore, the melody class must call the scale class to get a list of note pitches within the given scale and within a specific vocal range. This limits the options for possible melody pitches, which is in line with the Fuxian rules in which the melody must be diatonic and within a singable range.

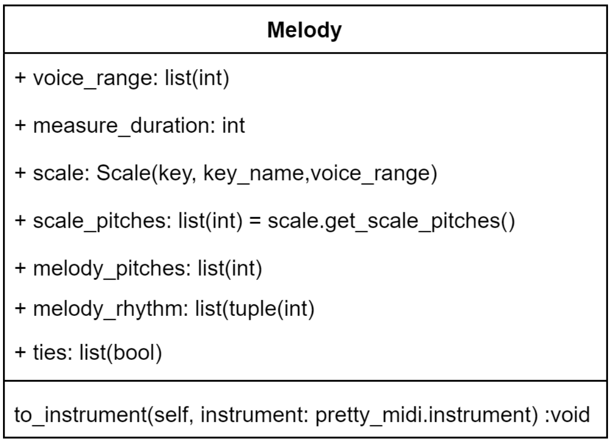
The list of possible vocal range pitches is the first main part that the melody class must include. This helps in defining the *search domain* of the search strategy. In addition, the two musical dimensions concerning rhythm and melody must be defined. The granularity of this representation can be extracted by analysis of the structure of the different species of counterpoint presented in section 2.1. The rhythm can be represented as a sequence of integer values which represent the number of eight-notes the corresponding note has as duration. Since some of the rules are concerned with checking dependencies across entire measures, the rhythm is structured measure by measure. To keep the data-structure as simple as possible for the guided search strategy, the pitches of the melody is represented as a list of their respective MIDI symbols, and *not* as a list of note objects. Lastly, some notes can move between measures. For example: tied notes in fourth species. The melody class must therefore contain some data-structure to identify if a note is tied forward or not.

To summarize; the main data-structure of the melody class is three parallel lists containing the melody pitches, note lengths in eight-notes, and if the note should be tied forward or not. The short melodic fragment shown in figure 4.5, therefore, has the following representation in the melody class:

```
melody = [60,62,64,65,69,67,65,65,64,62,60]
melody_rhythm = [(4,1,1,2),(2,2,4),(2,4,2),(8,)]
ties = [False, False, False, False, False,
        False, True, False, False, False, False]
```

Notice how the sum of all the measure-tuples are 8. This is because there can maximum be eight eight-notes in each measure. The tied note is also repeated in the list of melody pitches. This is to keep the length of the lists equal and set according to the measures.

As was the case of notes, intervals and scales, the melody class must also include functionality for loading note information to pretty_midi instruments for easy midi export. The main functionality that the melody class must satisfy is illustrated in the class diagram in figure 4.4. As can be seen, this is mostly a container of information.



```
Melody
+ voice_range: list(int)
+ measure_duration: int
+ scale: Scale(key, key_name,voice_range)
+ scale_pitches: list(int) = scale.get_scale_pitches()
+ melody_pitches: list(int)
+ melody_rhythm: list(tuple(int)
+ ties: list(bool)
to_instrument(self, instrument: pretty_midi.instrument) :void
```

Figure 4.4: Main functionality of the melody class.

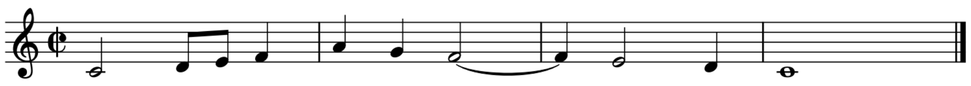


Figure 4.5: A small melodic fragment i C major alto range to illustrate the data-structure in the melody class.

It is important to specify that the melody class only contains the information *necessary* to generate a melody, but it does not contain functionality of melody generation. It can therefore *store* information such as the MIDI pitches and corresponding note durations and load this information to pretty_midi instruments, but it cannot *generate* said information. This was a deliberate design choice made to stay true to the container like structure as in the rest of the music module, delegating the task of generation to other modules of the system.

The constructor is implemented according to the class diagram in figure 4.4. Similar to the scale class, the melody class have input parameters "key", "scale" and "vocal range" to limit the list of possible notes for the melody. In addition, temporal information regarding the length of each measure is needed when loading the information to pretty_midi instruments. This is why *bar_length* (named *measure_duration* in 4.4) is included as an attribute. The parallel lists *melody_pitches*, *melody_rhythm* and *ties* are all declared as void. This information is to be filled in by the other modules. The code below also include the *set* and *get* methods for the three parallel lists that constitute the music representation.

```
class Melody:
    def __init__(self, key, scale, bar_length,
                 melody_notes=None, melody_rhythm = None,
                 ties = None, start=0, voice_range = None):
        self.key = key
        self.scale_name = scale
        self.voice_range = voice_range
        self.scale = Scale(key, scale, voice_range)
        self.scale_pitches = self.scale.get_scale_pitches()
        self.note_resolution = 8
        self.start = start
        self.bar_length = float(bar_length)

        """Music Representation"""
        self.pitches = melody_notes
        self.rhythm = melody_rhythm
        self.ties = ties
        if self.pitches != None:
            self.search_domain = [self.scale_pitches for
                                  notes in self.pitches]
        else:
            self.search_domain = [self.scale_pitches]

    def set_ties(self,ties):
        self.ties = ties.copy()

    def set_rhythm(self,rhythm):
        self.rhythm = rhythm.copy()

    def set_melody(self,melody):
        self.pitches = melody.copy()

    def get_ties(self):
        return self.ties.copy()

    def get_rhythm(self):
        return self.rhythm.copy()

    def get_melody(self):
        return self.pitches.copy()
```

The last part of the music class is the *to_instrument(instrument))* method, which now is more complex compared to the preceding classes. This is due to the consideration of having possible tied notes across measures. For each measure, the method iterates through all the note duration in the corresponding measure tuple. If the note is tied, the duration of the first note in the next measure is appended. In this way, notes can be expressed to lie outside the metric bounds of the bar lines. If the pitch value is *-1*, it means at the method should interpret the note duration as a *rest* and not a midi-number. Therefore, no note is loaded to the instrument if the pitch is *-1*, but the duration is still appended which means that the next note will be loaded after said rest. Notice also how the *bar_line* attribute is used to quantify the length of the eight-notes. If $bar\_length = 2$, then each eight-note in the measure has a length of $2/8 = 0.25$ *seconds*

```
""" MIDI SUPPORT """
def to_instrument(self, instrument, start = 0):
    i = 0
    measure_idx = 0
    t = start
    while measure_idx < len(self.rhythm):
        duration_idx = 0
        while duration_idx < len(self.rhythm[measure_idx]):
            dur = self.rhythm[measure_idx][duration_idx]
            duration = float(dur*self.bar_length /
                             float(self.note_resolution))
            if self.ties[i] == True:
                measure_idx += 1
                duration_idx = 0
                dur = self.rhythm[measure_idx][duration_idx]
                duration += float(dur*self.bar_length /
                                  float(self.note_resolution))
                i += 1
            if self.pitches[i] != -1:
                note = Note(self.pitches[i],start=t,
                            end=t+duration)
                note.to_instrument(instrument)
            t += duration
            i += 1
            note_duration += 1
        measure_idx += 1
```

4.3 Cantus Firmus

The **Cantus Firmus Class** is an extension of the melody class. The methods in the extension are for generating the data structures from the melody class necessary to express the music representation. The constructor is shown below, with the main functionality tasked with generating the rhythm, ties, and pitch values for the melody super class.

```
class Cantus_Firmus(m.Melody):
    def __init__(self,key, scale, bar_length,
                 melody_notes=None, melody_rhythm = None,
                 start=0, voice_range = RANGES[ALTO]):
        super(Cantus_Firmus, self).__init__(**args)
        self.cf_errors = []

        """ Music representation"""
        self.rhythm = self._generate_rhythm()
        self.length = len(self.rhythm)
        self.ties = [False]*len(self.rhythm)
        self.pitches = self._generate_cf()
```

The implementation of the cantus firmus class is divided into how the different dimensions of the music representation is generated, starting with the simplest one.

Rhythm and Ties Generation

Following the rules of Fux and Jeppesen, *all notes must be whole-notes* and the *length should be between 8 to 14 notes*. This restricts the rhythm and ties, which therefore easily can be stated as follows:

```
self.rhythm = self._generate_rhythm()
self.ties = [False]*len(self.rhythm)
```

The *_generate_rhythm()* method is private as indicated by the underscore at the beginning of the method name. The method generates a random number between 8 and 14 which acts as the number of measures in the cantus firmus. This number is then multiplied by the rhythmic skeleton, which in this case is tuples with the value 8, representing whole-notes:

```
def _generate_rhythm(self):
    random_length = rm.randint(8,14)
    return [(8,)]*random_length
```

Melody Generation

With the rhythm and list of tied notes in place, we can now turn the attention towards the harder task of generating the *melody pitches*. By inspecting the rules presented in section 2.1.3, two important observations can be made. First, rule 1 stating that *one must begin and end on the tonic to emphasize the key* and second, rule 6 stating that *the penultimate note should be a major second or minor second below the tonic* means that we can pre-constrain some of the pitches. The start, end and penultimate notes are therefore found using the following methods:

```
def _start_note(self):
    root = self.key
    try:
        root_idx = KEY_NAMES.index(root)
    except:
        root_idx = KEY_NAMES_SHARP.index(root)
    v_range = self.voice_range
    possible_start_notes = []
    for pitches in v_range:
        if pitches % Octave == root_idx:
            possible_start_notes.append(pitches)
    tonics = possible_start_notes
    return tonics,possible_start_notes[0]

def _penultimate_note(self):
    """ The last note can be approached from
        above or below. It is however most
        common that the last note is approached from above
    """
    leading_tone = self.start_note - 1
    super_tonic = self.start_note + 2
    weights = [0.1,0.9] # it is more common
                        # that the penultimate note is
                        # the supertonic than leading tone
    penultimate_note = rm.choices([leading_tone,
                                    super_tonic],weights)[0]
    return penultimate_note
```

Using these two methods as a basis, we can now begin to implement a randomized cantus firmus melody:

```
def _initialize_cf(self):
    """
    Randomizes the initial cf and sets
    correct start, end, and penultimate notes.
    :return: list of cf pitches.
    """
    start_note = self._start_note()[1]
    end_note = start_note
    penultimate_note = self._penultimate_note()
    length = len(self.rhythm)
    cf_shell = [rm.choice(self.scale_pitches)
                for i in range(length)]
    cf_shell[0] = start_note
    cf_shell[-1] = end_note
    cf_shell[-2] = penultimate_note
    return cf_shell
```

With this method in place, the first stage of the CF pitch sequence generator outlined in 3.5 is done. The next step is to quantify the rules in table 3.1, formalizing the *cost function* and *local search algorithm*

Cost Function and Constraint Formalism

The *cost function* is the sum of the accumulated penalty over a given sequence of cantus firmus pitches. The different constraints are expressed as boolean functions that scans through the given cantus firmus pitches and returns either *True* or *False* depending on whetever the rule is satisfied or not. For clarity, the structure of the cost function is presented first, as to give the reader an overview of the interface between the cost function and constraints:

```
def _cost_function(self,cf_shell):
    penalty = 0
    penalty = self._check_leaps(cf_shell) # rule 8,9,10
    if not self._is_valid_note_count(cf_shell): # rule 11
        self.cf_errors.append("note repetition")
        penalty += 100
    if not self._is_climax(cf_shell): # rule 3
        self.cf_errors.append("no unique cf climax")
        penalty += 100
    if not self._is_valid_range(cf_shell): # rule 5
        self.cf_errors.append("exceeds the range of a tenth")
        penalty += 100
    if self._is_repeated_motifs(cf_shell): # rule 11
        self.cf_errors.append("motivic repetitions")
        penalty += 100
    if not self._is_resolved_leading_tone(cf_shell): # rule 13
        self.cf_errors.append("leading tone not resolved")
        penalty += 100
    if self._is_dissonant_intervals(cf_shell): # implicit rule 2
        self.cf_errors.append("dissonant interval")
        penalty += 100
    return penalty
```

As can be seen in the code above, the cost function, given a cantus firmus draft as input, enforces a set of constraints and appends possible accumulated penalties to the total score. The different rules that are not satisfied are also stored in string format in a *cantus firmus error list* declared in the constructor. In this way, it is possible to easily identify possible mistakes in the generated result. For each boolean rule expression, the associated rule number corresponding to the rule presented in section 2.1.3 is also commented. For example: *_is_repeated_motifs()* corresponds to rule 11. The remaining rules have been enforced during pre-constraining in the cf initialization. The structure of the constraints are mostly similar, with one constraint being implemented differently than the other, namely the first checked *_check_leaps(cf_shell)* constraint. This is because the leaps checks are similar in design and scope, and are therefore grouped together:

```
def _check_leaps(self,cf_shell):
    penalty = 0
    num_large_leaps = 0
    for i in range(len(cf_shell)-2):
        if self._is_large_leap(cf_shell[i],cf_shell[i+1]):
            num_large_leaps += 1
            if abs(cf_shell[i]-cf_shell[i+1]) == Octave:
                # small penalty for octave leap
                self.cf_errors.append("penalty for octave leap")
                penalty += 50
            # Check consecutive leaps first
            elif self._is_large_leap(cf_shell[i+1],cf_shell[i+2]):
                self.cf_errors.append("consecutive leaps")
                penalty += 50
                if sign(cf_shell[i+1]-cf_shell[i]) /
                != sign(cf_shell[i+2]-cf_shell[i+1]):
                    self.cf_errors.append("Large leaps in"
                                          +"opposite direction")
                    penalty += 50
            elif self._is_step(cf_shell[i+1],cf_shell[i+2]) /
            and sign(cf_shell[i+1]-cf_shell[i]) /
            == sign(cf_shell[i+2]-cf_shell[i+1]):
                self.cf_errors.append("A leap is not"
                                      +"properly recovered")
                penalty += 75
    if num_large_leaps >= int(len(self.rhythm) /2) - 2:
        penalty += 100
    return penalty
```

The *_check_leaps(cf_shell)* shown above is expressed in its source code as to give the reader an idea of how the constraints were formalized. Notice especially the use of the help methods *_is_step(interval)*, *_is_small_leap(interval)* and *_is_large_leap(interval)*. These are used to identify the melodic leaps in the melody. Also, the help function *sign(x)* is used to recognize the *direction* of the melody. If *sign* returns 1, it means that the melody is ascending. If the sign method return -1, it means that the melody is descending.

The rest of the constraints are implemented in similar fashion. Showcasing the entire source code implementation would be extensive, which is why only two additional implementations are given: the *is_climax(cf_shell)* and *is_dissonant_intervals(cf_shell)*. Both are given in pseudo-code for clarification of program flow:

```
def _is_dissonant_intervals(self,cf_shell):
    dissonant_intervals = [m7, M7, Tritone,-m6,-m7,-M7]
    for each note in the cf_shell:
        if successive notes are in dissonant_intervals:
            return True
    return False

def _is_climax(self,cf_shell):
    if the number of max entries in cf_shell is 1:
        return True
    else: # has more than one climax point
        return False
```

This concludes the presentation of the cost function used in the cantus firmus generation. With all of the rules formalized, the value of the cost function reflects the "goodness" of a proposed solution, which can be used by the search algorithm to locate local optimal solutions.

Search Strategy

The search strategy is implemented according to figure 3.5. We now have methods for initializing a random melody, and the cost function provides a way to compute local and total penalties. The search algorithm is implemented as part of the *generate_cf()* method, which returns a sequence of optimal pitches. The structure of the *generate_cf()* method is given below:

```
def _generate_cf(self):
    total_penalty = math.inf
    iteration = 0
    while total_penalty > 0 and iteration < 1000:
        cf_shell = self._initialize_cf() # initialized randomly
        for i in range(1,len(cf_shell)-2):
            self.cf_errors = []
            local_max = math.inf
```

```
                cf_draft = cf_shell.copy()
                possible_notes = /...
                    self._get_melodic_consonances(cf_shell[i-1])
                best_choice = possible_notes[0]
                for note in possible_notes:
                    cf_draft[i] = note
                    local_penalty = self._cost_function(cf_draft)
                    if local_penalty <= local_max:
                        local_max = local_penalty
                        best_choice = notes
                cf_shell[i] = best_choice
            self.cf_errors = []
            total_penalty = self._cost_function(cf_shell)
            iteration += 1
        return cf_shell.copy()
```

The algorithm literates through all of the randomized notes of the cantus firmus except for the terminals and penultimate note, which is pre-set. For each note, a list of possible notes is extracted using the *_get_melodic_consonances(prev_note)* method. For each of these possibilities, the cost function is called to identify which of the possibilities leads to the minimum penalty. This best choice is then picked, and the search can move to the next note. After the whole cantus firmus has been searched, a last call to the cost function is made to find the total penalty for the now optimized cantus firmus draft. If this totals more than 0, a new pass is made with a randomization of the cantus firmus notes. Given the few constraints and short melody, the algorithm usually converges within 30 iterations.

The cantus firmus constructor now have the necessary functionality to load the data-structure with the correct music representation format. The call:

```
C_major_cf = Cantus_Firmus("C","major",bar_length=2,
                           voice_range = RANGES[ALTO])
```

now constructs a cantus firmus melody in alto range with the melody super-class containing feasible rhythm, ties and melody pitches. Since the bar length is 2, each eight note has a length of 0.25 seconds. Temporal information is therefore also stored, which means that the call:

```
C_major_cf.to_instrument(pretty_midi.instrument)
```

can be used to load the music representation to a pretty_midi instrument, which then can be written to a midi-file.

4.4 Counterpoint

The counterpoint module is to contain information regarding the structure of the different counterpoints, subject to two main tasks. The first is to limit the search domain of the search algorithm as to make the convergence to valid solutions faster. The second is to abstract away as much information as possible from the search algorithm, making the search strategy more generalized and usable by all the species despite their differences. To achieve this, the musical dimensions concerning rhythm, ties and instrumentation must be abstracted away from the search strategy.

Therefore, each individual species class is tasked with generating the rhythm, ties and search domain of their respective species. The *Counterpoint Super Class* contains functionality common for all the different species. This includes methods for extracting the possible start, end, and penultimate notes, in addition to acting as the interface with the search strategy. The initial randomization of the counterpoints is also done here, following the music representation formalized in the different species classes. The class hierarchy is illustrated in the class diagram in figure 3.7.

4.4.1 The Counterpoint Super Class

The code below includes the attributes and methods of the **Counterpoint Class**. The possible start, end, and penultimate notes are equal for all of the five different species, which is why the functionality is located here;

Code 4.1: Counterpoint Class implementation

```
class Counterpoint:
    """ CONSTRUCTOR """
    def __init__(self,cf,ctp_position = "above"):
        if ctp_position == "above":
            self.voice_range = /...
                RANGES[RANGES.index(cf.voice_range)+1]
        else:
            self.voice_range = /...
                RANGES[RANGES.index(cf.voice_range)-1]
        self.melody = m.Melody(cf.key,cf.scale,cf.bar_length,
                               voice_range = self.voice_range)
        self.ctp_position = ctp_position
        self.scale_pitches = self.melody.scale_pitches
        self.cf = cf
        self.species = None
        self.search_domain = []
        self.ctp_errors = []
        self.MAX_SEARCH_TIME = 5 #seconds

    """ VALID START, END, AND PENULTIMATE NOTES"""
    def _start_notes(self):...

    def _end_notes(self):...

    def _penultimate_notes(self, cf_end):...

    """ INITIALIZING COUNTERPOINT WITH RANDOM VALUES"""
    def get_consonant_possibilities(self,cf_note):...

    def randomize_ctp_melody(self):...

    """ GENERATE COUNTERPOINT PITCHES BY
        CALLING THE SEARCH ALGORITHM"""
    def generate_ctp(self):...
```

The methods for extracting possible *start*, *end* and *penultimate* notes are similar to that of the cantus firmus module. According to Fux, if the counterpoint is above the cantus firmus, the possible start notes are perfect intervals including unison for first species. The end note is either unison or octave, and the penultimate is as for the cantus firmus either a major second above or minor second below the end note. If the counterpoint is below the cantus firmus, the possible notes are further restricted to emphasize the key. The methods therefore have the following implementation in pseudo-code:

```
def _start_notes(self):
    cf_tonic = self.cf.pitches[0]
    if self.ctp_position == "above":
        if SPECIES[self.species] == 1:
            return [cf_tonic, cf_tonic + P5, cf_tonic + Octave]
        else:
            return [cf_tonic+P5,cf_tonic + Octave]
    else:
        if SPECIES[self.species] == 1:
            return [cf_tonic - Octave, cf_tonic]
        else:
            return [cf_tonic - Octave]

def _end_notes(self):
    cf_tonic = self.cf.pitches[0]
    if self.ctp_position == "above":
        return [cf_tonic, cf_tonic + Octave] # unison and octave
    else:
        return [cf_tonic, cf_tonic - Octave]

def _penultimate_notes(self, cf_end):
    if last note in cantus firmus is approached from below:
        # Then the end note of the
        # counterpoint must be approached from a major
        # second above
        penultimate = end_note+M2
    elif last note in cantus firmus is approached from above:
        # Then the end note of the
        # counterpoint must be approached by a minor
        # second below
        penultimate = end_note-m2
    if self.ctp_position = "above":
        return [penultimate, penultimate + Octave]
    else:
        return [penultimate, penultimate - Octave]
```

Two methods to aid in the initialization of the melody pitches of the different species is also included in the super-class. These methods are the *get_consonant_possibilities(cf_note)* and *randomize_ctp_possibilities(cf_note)*, declared in 4.1. The *get_consonant_possibilties(cf_note)* method returns a list of the possible consonant pitches either above or below the given cantus firmus note, and is used for limiting the search domain according to rule 3 in 2.1.4 stating that *all harmonies must be consonant*:

```
def get_consonant_possibilities(self,cf_note):
    poss = []
    for interval in HARMONIC_CONSONANCES:
        if self.ctp_position == "above":
            if cf_note+interval in self.scale_pitches:
                poss.append(cf_note+interval)
        else:
            if cf_note-interval in self.scale_pitches:
                poss.append(cf_note-interval)
    return poss
```

The counterpoint pitch randomization method uses the search domain defined by the corresponding species class and returns a sequence of feasible pitches.

```python
def randomize_ctp_melody(self):
    ctp_melody = []
    i = 0
    measure = 0
    while measure < len(self.melody.rhythm):
        note_duration = 0
        while note_duration < len(self.melody.rhythm[measure]):
            if i == 0:
                ctp_melody.append(/...
                    rm.choice(self.search_domain[i]))
            elif i > 0 and self.melody.ties[i-1] == True:
                ctp_melody.append(ctp_melody[i-1])
            else:
                ctp_melody.append(/...
                    rm.choice(self.search_domain[i]))
            i += 1
            note_duration += 1
        measure += 1
    return ctp_melody
```

Notice how the underlying music representation contained in the *melody object* declared in the counterpoint constructor is used when constructing a randomized counterpoint. This is illustrated by the calls to *self.melody.rhythm* and *self.melody.ties*. For each note duration in the rhythm generated by the underlying child class, a note is picked randomly from the corresponding list of possible notes in the search domain. If the note is tied, the previous note is used as to stay true with the musical representation defined in the melody class. This is emphasized in the code below:

```python
.
.
.
elif i > 0 and self.melody.ties[i-1] == True:
    # The note is tied forward from the preceding measure.
    # The note appended to the randomized sequence of pitches
    # must therefore be equal to the previous note
    ctp_melody.append(ctp_melody[i-1])
else:
    ctp_melody.append(rm.choice(self.search_domain[i]))
.
.
.
```

4.4.2 The General Structure of the Species Class

To load the counterpoint melody with a valid music representation, child classes of the counterpoint class are implemented for each of the five different species. The method names are the same for all the classes, but the implementation is varied according to the different rhythms, possible dissonances and ties. All the species classes have the following constructor form:

```
class Species(Counterpoint):
    def __init__(self,cf,ctp_position = "above"):
        super(Species,self).__init__(cf,ctp_position)
        self.species = "first","second",
                        "third","fourth", or "fifth"
        self.ERROR_THRESHOLD = 100

        """ Music Representation """
        self.melody.set_rhythm(self.get_rhythm())
        self.num_notes = sum(len(row)
                             for row in self.get_rhythm())
        self.melody.set_ties(self.get_ties())
        self.search_domain = self._possible_notes()
        self.melody.set_melody(self.randomize_ctp_melody())
```

The ERROR THRESHOLD attribute is the maximum allowed penalty for a generated species counterpoint. For first species, given its low complexity, this is set to 50. This corresponds to one allowed bad penalty. For the remaining species, the threshold is set to be below 100, which represents a severe penalty.

Each species has three main methods, *get_rhythm()*, *get_ties()* and *_possible_notes()*. These methods generate information regarding the musical dimensions that is *not* to be explored by the search strategy. How these methods are implemented for each of the five different species is illustrated in the following subsections.

4.4.3 FirstSpecies Class

Rhythm and Ties

The rhythmic skeleton of first species is equal to the cantus firmus; one whole-note for each measure and zero ties. The *get_rhythm()* and *get_ties()* methods therefore have the following trivial form:

```
""" FIRST SPECIES RHYTHM """
def get_rhythm(self):
    #Voices all move together in the same
    #rhythm as the cantus firmus.
    return [(8,)]*self.cf.length

def get_ties(self):
    return [False]*len(self.cf.length)
```

Search Domain

The search domain is structured as a 2D list of possible counterpoint notes for each note in the cantus firmus. For first species, only harmonic consonant intervals of the cantus firmus note is allowed. The *_possible_notes()* method therefore has the following implementation:

```
""" FIRST SPECIES SEARCH DOMAIN """
def _possible_notes(self):
    poss = [None for elem in self.melody.rhythm]
    for i in range(len(self.melody.rhythm)):
        if i == 0:
            poss[i] = self._start_notes()
        elif i == len(self.melody.rhythm)-2:
            poss[i] = self._penultimate_notes(/...
                      self.cf.pitches[-1])
        elif i == len(self.melody.rhythm)-1:
            poss[i] = self._end_notes()
        else:
            poss[i] = self.get_consonant_possibilities(/...
                      self.cf.pitches[i])
    return poss
```

The method iterates through all the notes as dictated by the rhythmic skeleton, constructing a list of possible notes for each note in the cantus firmus. The terminal and penultimate notes are further constrained, and is set accordingly.

4.4.4 SecondSpecies Class

Rhythm and Ties

In second species, there are two notes for each note in the cantus firmus except for the last note, which is a whole-note. There are still no ties allowed, so the methods *get_rhythm()* and *get_ties()* have the following implementation:

```
""" SECOND SPECIES RHYTHM """
def get_rhythm(self):
    rhythm = [(4,4)]*(self.cf.length-1)
    rhythm.append((8,))
    return rhythm

def get_ties(self):
    return [False]*self.num_notes
```

Search Domain

In second species, dissonances on weak beats are allowed. The difference between strong and weak beats in measures are illustrated in figure 4.6. The *_possible_notes()* method is therefore a bit more extensive than that of first species. Therefore, an additional method is implemented to help distinguish between which notes can be dissonant and which can only be consonant. This is done in the *get_harmonic_possibilities()* method, which add dissonant possibilities if the note index is on a rhythmic weak beat:

```
    def get_harmonic_possibilities(self, idx, cf_note):
        poss = super(SecondSpecies,self)./...
                get_consonant_possibilities(cf_note)
        upbeats = self.get_upbeats()
        if idx in upbeats:
            for diss in HARMONIC_DISSONANT_INTERVALS:
                if self.ctp_position == "above":
                    if cf_note+diss in self.scale_pitches:
                        poss.append(cf_note+diss)
                else:
                    if cf_note-diss in self.scale_pitches:
                        poss.append(cf_note-diss)
        return poss
```

The *_possible_notes()* method is similar to that of first species. But now, the rhythm is iterated measure by measure, and the first note is always set to be a rest, illustrated by the "-1" token.

```
""" SECOND SPECIES SEARCH DOMAIN """
def _possible_notes(self):
    poss = [None for elem in range(self.num_notes)]
    i = 0
    for m in range(len(self.get_rhythm())):
        for n in range(len(self.get_rhythm()[m])):
            if m == 0:
                # First measure. start notes
                if n == 0:
                    poss[i] = [-1]
                else:
                    poss[i] = self._start_notes()
            elif m == len(self.get_rhythm()) - 2 and n == 1:
                # penultimate note before last measure.
                poss[i] = self._penultimate_notes(/...
                             self.cf.pitches[-1])
            elif m == len(self.get_rhythm())-1:
                # Last measure
                poss[i] = self._end_notes()
            else:
                poss[i] = self.get_harmonic_possibilities(/...
                             i, self.cf.pitches[m])
            i += 1
    return poss
```

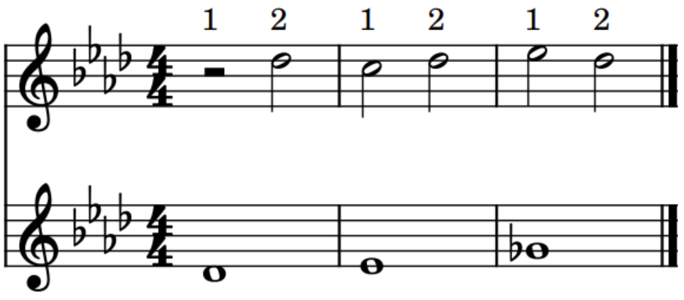


Figure 4.6: Strong beats are notated 1, weak beats are notated 2.

4.4.5 ThirdSpecies Class

Rhythm and Ties

Again, the rhythm and ties are easy to generate, since for third species there are four notes for each note in the cantus firmus, and no ties are allowed. Each note has a length of 2 eight-notes, except for the last measure which is a whole-note. This leads to the following implementation:

```
""" THIRD SPECIES RHYTHM """
def get_rhythm(self):
    rhythm = [(2, 2, 2, 2)] * (self.cf.length - 1)
    rhythm.append((8,))
    return rhythm

def get_ties(self):
    return [False] * self.num_notes
```

Search Domain

As for second species, weak beats can be dissonant. The weak beats are the second and fourth beats of each measure, and the strong beats the first and third. This is illustrated in figure 4.7. Since the structure of the search domain generation is just an extension of that of second species, the same methods used in second species can be used in third species, with the only change being the rhythmic difference.

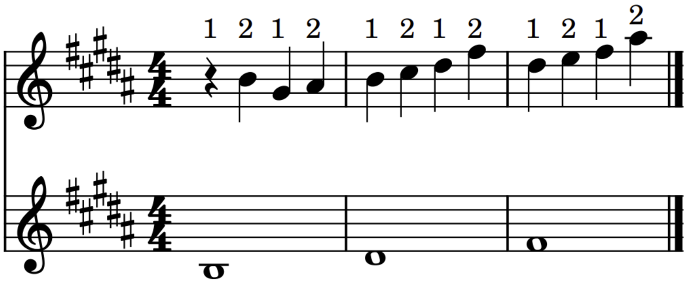


Figure 4.7: Strong beats are notated 1, weak beats are notated 2.

4.4.6 FourthSpecies Class

Rhythm and Ties

The rhythm of the fourth species is equal to that of second species. However, there is now a tied note at the end of each measure except for the last and penultimate measure. This leads to the first introduction of tie handling, which given the strict syncopated rhythm of fourth species is quite trivial:

```
""" FOURTH SPECIES RHYTHM """
def get_rhythm(self):
    rhythm = [(4, 4)] * (self.cf.length - 1)
    rhythm.append((8,))
    return rhythm

def get_ties(self):
    ties = []
    for i in range(self.num_notes-2):
        if i%2 == 0:
            ties.append(False)
        else:
            ties.append(True)
    # The last two notes are not tied:
    ties.append(False)
    ties.append(False)
    return ties
```

Search Domain

The notes in fourth species can only be consonant intervals. The *_possible_notes()* method for fourth species is therefore equal to that of second species, except for dissonances not being allowed. Given the tied notes, each note duration is in effect a whole note going between consecutive measures. Since the beginning of the counterpoint is a rest, fourth species can be viewed as a first species counterpoint shifted by a half-note.

4.4.7 FifthSpecies Class

Rhythm and Ties

Fifth species introduces a more florid rhythm and list of tied notes. This is due to fifth species being an extension of all the preceding species. The rhythm in fifth species can therefore have rhythmic elements from second, third and fourth species, making the counterpoint border tonal music.

Section 2.1.8 presents several rules regarding rhythm, which in this system is not to be explicitly handled. This was also discussed in the design and in the assumptions, and is due to the focus on the melodic dimension rather than the rhythmic dimensions. However, the proposed rhythmic generation seen below still provide interesting and varied rhythms, with the possibility of having eight notes on weak beats.

```
""" FIFTH SPECIES RHYTHM """
def get_rhythm(self):
    rhythm = []
    measure_rhythms = [(2,2,2,2),(4,2,2),(2,2,4),(4,4),
                       (2,1,1,2,2),(2,1,1,4),(4,2,1,1),
                       (2,2,2,1,1),(2,1,1,2,2)]
    rhythmic_weights = [100,50,50,25,10,5,5,5,5]
    for measures in range(cf.length-1):
        if measures == 0:
            rhythm.append((4,4))
        else:
            rhythm.append(/...
            rm.choices(measure_rhythms,rhythmic_weights)[0])
    rhythm.append((8,))
    return rhythm
```

The first measure is fixed to be two half-notes. The first note is a rest, the second a start note. In this way, the tonality of the counterpoint is highlighted with the slower rhythm in the first measure. Faster rhythms are instead introduced in subsequent measures. Different rhythmic tuples have been listed. These acts as instructions for the possible measure-wise rhythm, and is listed by relevance. The first entry is the same rhythm as that of third species. The second and third entry is a combination of second and third species, while the fourth entry is the rhythm of second species. The last options all include eight notes on weak beats. Since they are to appear more rarely, they are given a lower weight. For each measure except the terminals, a randomized measure rhythm is chosen based on the list of possibilities and associated weights. In this way, the rhythm becomes more florid.

There are also some considerations that has to be made when handling possible tied notes. A variation of rule 3 presented in 2.1.8 is therefore implemented, stating that *all half-notes followed by a quarter-note across measures is to be tied*. In this way, the counterpoint gets a syncopated effect while not halting the melodic flow to a large extent. This also helps in the independence between the cantus firmus and counterpoint. As in fourth species, the penultimate note and end note can not be tied.

```
def get_ties(self):
    rhythm = self.rhythm
    ties = []
    for m in range(len(rhythm)-1):
        for n in range(len(rhythm[m])):
            if m == 0 and n == 1:
                ties.append(True)
            elif m > 0 and n == len(rhythm[m])-1:
                if rhythm[m+1][0] == rhythm[m][n]/2:
                    ties.append(True)
                else:
                    ties.append(False)
            else:
                ties.append(False)
    ties.append(False)
    ties.append(False)
    return ties
```

Search Domain

The construction of the search domain is similar to that of third species. However, eight notes is now a possibility. Since they always appear on weak beats, they must always include dissonant possibilities. Other possible dissonances are identified by the sum of the note durations in the measure up until the given index. If the sum is 2 or 6, the corresponding beat of the measure is either 2 and 4, which are weak beats. Dissonance should therefore be included in these cases. The methods *get_harmonic_possibilities()* and *_possible_notes()* are implemented as follows:

```python
""" DEFINING SEARCH DOMAIN FIFTH SPECIES """
def get_harmonic_possibilities(self, m,n, cf_note):
    add_diss = False
    if self.rhythm[m][n] == 1:
        add_diss = True
    if sum(self.rhythm[m][:n]) in [2,6]:
        add_diss = True
    poss = super(FifthSpecies, self)./...
           get_consonant_possibilities(cf_note)
    if add_diss:
        for diss in HARMONIC_DISSONANT_INTERVALS:
            if self.ctp_position == "above":
                if cf_note + diss in self.scale_pitches:
                    poss.append(cf_note + diss)
            else:
                if cf_note - diss in self.scale_pitches:
                    poss.append(cf_note - diss)
    return poss

def _possible_notes(self):
    poss = [None for elem in range(self.num_notes)]
    i = 0
    for m in range(len(self.rhythm)):
        for n in range(len(self.rhythm[m])):
            if m == 0:
                # First measure. start notes
                if n == 0:
                    poss[i] = [-1]
                else:
                    poss[i] = self._start_notes()
            elif m == len(self.rhythm) - 2 and /...
                        n == len(self.rhythm[m])-1:
                # penultimate note before last measure.
                poss[i] = self._penultimate_notes(/...
                          self.cf.pitches[-1])
            elif m == len(self.rhythm) - 1:
                # Last measure
                poss[i] = self._end_notes()
            else:
                poss[i] = self.get_harmonic_possibilities(/...
                          m,n, self.cf.pitches[m])
            i += 1
    return poss
```

4.4.8 generate_ctp

The last method in 4.1 yet to be presented is the *generate_ctp()* method, which calls the search strategy to find a local optimal melody sequence for the given counterpoint. This acts as the interface between the counterpoint object and the search strategy:

```
def generate_ctp(self):
    if self.species == None:
        print("No species to generate!")
    self.ctp_errors = []
    self.error, best_ctp, self.ctp_errors = /...
                Search_Algorithm.guided_local_search(self)
    self.melody.set_melody(best_ctp)
```

As can be seen above, the search algorithm returns the integer error, the best pitch sequence it found, and a list of possible unsatisfied constraints in string format. The search algorithm takes a *Counterpoint* object as argument.

4.5 The Guided Local Search Strategy

The search strategy was implemented according to the flow diagram shown in 3.12. Given its importance in the system structure and compact form, the implementation is here presented in a high level of detail. The search strategy module consists of only two functions: the *guided_search()* and *local_search()*. The local search is to adjust the initial sequence of pitches that was initialized randomly in the corresponding species class. For each slot in this pitch sequence, there is a corresponding list of possible pitches that can be chosen for that slot. These pitches were defined by the search domain of the given counterpoint object. The *search order*, that is, the order in which the slots of the pitch sequence is visited, is given by a list of weighted indices. This to avoid a implicit "look-ahead" preference in the search order by always scanning from the lowest index and up. For each possible note in the search domain for a given pitch index, the pitch which leads to the lowest penalty is chosen as the option. In this way, the local search always picks the option which results in the *globally least accumulated penalty* from the set of possible note possibilities. The local search function returns the best counterpoint pitch sequence it found, along with the associated penalty score and a dictionary of the pitch indices ordered by which index has the most local penalties. The weighted indices are given by the constraints module after the cost function has been issued. An example of a weighted index dictionary is given below:

```
weighted_idx = {2: 5, 4: 5, 3: 4,
                0: 0, 1: 0, 5: 0,
                         6: 0, 7: 0, 8: 0, 9: 0}
```

In the above example, the pitches at index 2, 5, 4 and 3 are the only ones leading to penalties, with index 2 and 5 being the worst with a value of 5. These weighted indices are used to identify which pitches that should be changed first. The local search function has the following form in high level pseudo-code:

```
def local_search(ctp,search_order):
    search_domain = ctp.search_domain
    search_ctp = ctp.melody.get_melody()
    best_global_ctp = search_ctp.copy()
    best_global_error = math.inf
    best_global_weighted_indices = []
    for i in search_order:
        # for each slot in the pitch sequence..
        best_note = search_domain[i][0]
        local_error = math.inf
        local_weighted_indices = []
        for j in range(len(search_domain[i])):
            # for each note possibility for the pitch slot..
            # check the note possibility by assigning it to the
            # appropriate pitch slot in the pitch sequence
            search_ctp[i] = search_domain[i][j]
            ctp.melody.set_melody(search_ctp.copy())
            validity = Constraints(ctp)

            # check the associated accumulated penalty
            # and get indices ordered by their local error
            error = validity.cost_function()
            weighted_indices = validity.get_weighted_indices()
            if error <= local_error:
                # Update the best note possibility
                best_note = search_domain[i][j]
                local_error = error
                local_weighted_indices = weighted_indices
        # after all the possibilities are checked,
        # set the best one in the associated pitch sequence slot
        search_ctp[i] = best_note
        if local_error < best_global_error:
            # if the proposed counterpoint has lower score
            # than the current best, update current best
            best_global_ctp = search_ctp.copy()
            best_global_error = local_error
            best_global_weighted_indices = local_weighted_indices

    return best_global_error, best_global_ctp,
           best_global_weighted_indices
```

To help the local search strategy to exit sub-optimal local minimums, another function is wrapped around the local search function. This is the *guided_search(ctp)* procedure, which is the function that is directly called from the counterpoint module. The function is tasked with guiding the local search with the aid of the returned weighted indices. If the local search strategy has reached a sub-optimal local minimum, identified by the penalty being the same for each call, one or more pitches in the pitch sequence is randomized starting with the pitch index that has the highest associated penalty in the weighted index dictionary. In this way, the search strategy always prioritize the most penalized pitches. If the initial randomization did not guide the search out of the local minimum, the number of pitches to be randomized is increased by one. This continues until the penalty is again minimized, which resets the number of pitches to be randomized back to one.

The guided search iterates until a valid solution is found with accumulated penalty below the threshold, or the time limit of 5 seconds is reached. When the search terminates, a last call is made to the constraints module to get the penalty and list of errors in string format of the final sequence of counterpoint pitches. The implementation represented in high-level pseudo-code is shown below:

```
def guided_search(ctp):
    start_time = t.time()
    penalty = math.inf
    elapsed_time = t.time()-start_time
    best_ctp = ctp.melody.get_melody()
    lowest_penalty = math.inf
    # the initial search order is a
    # scan from left to right
    search_order = [i for i in range(len(best_ctp))]
    prev_penalty = penalty

    # how many of the pitch slots to be randomized
    # according to the weights
    randomize_idx = 1
    while penalty >= ctp.ERROR_THRESHOLD and /...
                     elapsed_time < ctp.MAX_SEARCH_TIME:
        penalty, ctp_notes,weighted_idx = /...
                local_search(ctp,search_order)
        if penalty == prev_penalty: # no improvement
            weighted_idx = list(weighted_idx.keys())
            for i in range(randomize_idx):
                # picks a new, possible local sub-optimal
                # pitch from the search domain
                ctp_notes[weighted_idx[i]] = /...
                    rm.choice(ctp.search_domain[weighted_idx[i]])

            # the search order is shuffled
```

```
                # to avoid implicit search order preferences
                # and as an additional step to break out of
                # unwanted local minima
                rm.shuffle(search_order)
                ctp.melody.set_melody(ctp_notes)
                if randomize_idx != len(best_ctp)-1:
                    randomize_idx += 1
            if penalty < lowest_penalty:
                randomize_idx = 1
                best_ctp = ctp_notes
                ctp.melody.set_melody(best_ctp)
                lowest_penalty = penalty
                weighted_idx = weighted_idx
            elapsed_time = t.time()-start_time
            prev_penalty = penalty
        constraint = Constraints(ctp)
        lowest_penalty = constraint.get_penalty()
        lowest_error_list = constraint.get_errors()
        return lowest_penalty, best_ctp,lowest_error_list
```

4.6 Constraints

The implementation of the cost function and constraints formalism for the counterpoint generation follows the design presented in section 3.6 quite thoroughly. That means that for each of the constraints in their respective rule-category, a method is implemented to check the validity of said constraint issued on a given counterpoint. The cost function is structured by calling each of the four main constraint categories:

```
def cost_function(self):
    penalty = 0
    self.ctp_errors = []

    # pitch sequence provided by the counterpoint object:
    ctp_draft = self.ctp

    # extended version of the cantus firmus to
    # ensure 1:1 correspondence between cf pitch sequence
    # and counterpoint:

    cf_notes = self.extended_cantus_firmus

    # accumulated penalty:
    penalty += self._melodic_rules(ctp_draft)
    penalty += self._voice_independence_rules(ctp_draft,
                                              cf_notes)
    penalty += self._dissonance_handling(cf_notes, ctp_draft)
    penalty += self._harmonic_rules(ctp_draft, cf_notes)
    return penalty
```

The different constraint categories are based on their respective list of outlined rules that they must satisfy. Each constraint is given a penalty related to the severity of breaking said rule. The integer values associated with each penalty is illustrated in table 4.1.

Penalty	Integer Value
severe	100
bad	50
minor	25
preference	5

Table 4.1: The different penalties and associated integer values.

In the following subsections, the layout of the different categorizes are presented with the inclusion of how the different species are handled.

4.6.1 Melodic Rules

The melodic rules category is the most extensive. However, most of the rules are applicable to all species of counterpoint with only minor alterations. To keep the structure as streamlined as possible, each constraint is implemented as a boolean expression with a descriptive name. An integer value is associated with each constraint expression according to the severity of breaking said constraint. The *_melodic_rules()* method is therefore structured as a sequence of IF statements, checking each of the implemented boolean constraint expressions. If the rule is broken, a penalty is added to the accumulated penalty for the rule-category, a text representation of what rule is broken is added to the list of errors, and lastly, the weight of the corresponding error pitch index is increased. With the implementation of all the rules listed in 3.3 in section 3.6, the *_melodic_rules()* method has the following form:

```
def _melodic_rules(self, ctp_draft):
    penalty = 0
    # Index based rules
    # valid melodic rules for each species
    for i in range(len(ctp_draft)):
        if self._is_melodic_leap_too_large(ctp_draft, i):
            self.ctp_errors.append("Too large leap!")
            penalty += 100
            self.weighted_indices[i] += 4
        if self._is_melodic_leap_octave(ctp_draft, i):
            self.ctp_errors.append("Octave leap!")
            penalty += 25
            self.weighted_indices[i] += 1
        if not self._is_leap_compensated(ctp_draft, i):
            self.ctp_errors.append("Leap not compensated!")
            penalty += 50
            self.weighted_indices[i] += 2
        if not self._is_octave_compensated(ctp_draft, i):
            self.ctp_errors.append("Octave not compensated!")
            penalty += 25
            self.weighted_indices[i] += 1
        if self._is_successive_same_direction_leaps(ctp_draft, i):
            self.ctp_errors.append("Successive Leaps"+
                                   "in same direction!")
            penalty += 25
            self.weighted_indices[i] += 1
            if not self._is_successive_leaps_valid(ctp_draft, i):
                self.ctp_errors.append("Successive leaps"
                                       +"strictly not valid!")
                penalty += 100
                self.weighted_indices[i] += 4
        if self._is_chromatic_step(ctp_draft, i):
            self.ctp_errors.append("Chromatic movement!")
            penalty += 100
            self.weighted_indices[i] += 4
        if self._is_repeating_pitches(ctp_draft,i):
            self.ctp_errors.append("Repeats pitches!")
            penalty += 100
            self.weighted_indices[i] += 1
        # Global rules
        if not self._is_within_range_of_a_tenth(ctp_draft):
            self.ctp_errors.append("Exceeds the"+
                                   "range of a tenth!")
            penalty += 50
        if not self._is_unique_climax(ctp_draft):
            self.ctp_errors.append("No unique climax or at same"+
                                   "position as other voices!")
            penalty += 100
        if not self._is_leading_tone_properly_resolved(ctp_draft):
            self.ctp_errors.append("leading tone not"
                                   +"properly resolved!")
            penalty += 100
    if SPECIES[self.species] >= 2:
        for i in range(len(ctp_draft)):
            if self._is_motivic_repetitions(ctp_draft,i):
                self.ctp_errors.append("Motivic repetitions!")
                penalty += 100
    return penalty
```

Showcasing the implementation of each individual boolean expression would be too extensive, which is why only two examples are given to illustrate the implementation of one global and one index based rule. The first example is the important rule regarding if leaps are too big. The implementation illustrates a minor difference in handling leaps in the case of fifth and third species. The constraint is index based, and given a sequence of counterpoint pitches and index it checks ahead to see if the melodic leap is allowed:

```
def _is_melodic_leap_too_large(self, ctp_draft, idx):
    if idx in self.end_idx or ctp_draft[idx] == -1:
        # rest, start and end index should be ignored
        return False
    interval = ctp_draft[idx + 1] - ctp_draft[idx]
    if abs(interval) > P5: # the interval is a large leap
        if self.species == "fifth":
            if self.note_durations[idx] < 4 /...
                               and self.note_durations[idx+1] < 4:
                # Having large leaps between short notes can
                # make the counterpoint
                # feel jagged, and is therefore not allowed
                return True
        if sign(interval) == 1.0 and interval == m6 /...
                               and self.species != "third":
            # ascending minor sixth interval
            # allowed in all species
            # except third species
            return False
        if abs(interval) == Octave:
            return False
        return True
    else:
        return False
```

The second example has a simpler implementation, and checks if the pitches in the counterpoint is within the range of a tenth. Notice how the first entry in the pitch sequence is omitted when identifying the lowest pitch value. This is due to the possibility of having a rest, which should be ignored:

```
def _is_within_range_of_a_tenth(self, ctp_draft):
    if max(ctp_draft) - min(ctp_draft[1:]) > Octave + M3:
        return False
    else:
        return True
```

4.6.2 Voice-Independence Rules

The voice-independence rules is structured in similar fashion to that of the melodic rules. The implemented constraints are equal to the ones formalized in table 3.4 in section 3.6. The number of constraints are less extensive than the previous category, but now the harmony between the cantus firmus and counterpoint melody is checked. The *_voice_independence_rules()* therefore have an additional parameter, namely the extended cantus firmus notes:

```
def _voice_independence_rules(self, ctp_draft, cf_notes):
    if self.ctp_position == "above":
        upper_voice = ctp_draft
        lower_voice = cf_notes
    else:
        upper_voice = cf_notes
        lower_voice = ctp_draft
    penalty = 0
    # Index based rules
    # valid rules for each species
    for i in range(len(ctp_draft)):
        if not self._is_perfect_interval_properly_approached(/...
                                  upper_voice, lower_voice, i):
            self.ctp_errors.append("Perfect interval not"+
                                   "properly approached!")
            penalty += 100
            self.weighted_indices[i] += 4
        if not self._is_valid_consecutive_perfect_intervals(/...
                                 upper_voice, lower_voice, i):
            self.ctp_errors.append("Consecutive perfect intervals,
                                   +"but they are not valid!")
            penalty += 100
            self.weighted_indices[i] += 4
        if self._is_parallel_fourths(/...
             upper_voice, lower_voice, i):
            self.ctp_errors.append("Parallel fourths!")
            penalty += 50
            self.weighted_indices[i] += 2
        if self._is_voice_overlapping(/...
              upper_voice, lower_voice, i):
            self.ctp_errors.append("Voice Overlapping!")
            penalty += 100
            self.weighted_indices[i] += 4
        if self._is_voice_crossing(upper_voice, lower_voice, i):
            self.ctp_errors.append("Voice crossing!")
            penalty += 50
            self.weighted_indices[i] += 2
        if self._is_contrary_motion(upper_voice, lower_voice, i):
            # This not not a severe violation,
            # but more of a preference to avoid similar motion
            penalty += 5
    # Global rules
    if not self._is_valid_number_of_consecutive_intervals(/...
                                        upper_voice, lower_voice):
        self.ctp_errors.append("Too many consecutive intervals!")
        penalty += 100
    if self._is_unisons_between_terminals(ctp_draft):
        self.ctp_errors.append("Unison between terminals!")
        penalty += 50

    # Additional rule for species 2,3,4,5
    if SPECIES[self.species] >= 2:
        if self._is_parallel_perfects_on_downbeats(/...
                ctp_draft, upper_voice, lower_voice):
            self.ctp_errors.append("Parallel perfect intervals"
                                   +"on downbeats!")
            penalty += 100
    return penalty
```

Notice how there is only one additional rule for species 2, 3, 4 and 5. This is the constraint regarding parallel perfect intervals on downbeats, which is not allowed in the species with fast rhythms.

Again, two constraint implementations are shown. The first one is index based, and arguably one of the most important rules of Fux; how approaches to perfect intervals should be handled:

```
def _is_perfect_interval_properly_approached(self,
                    upper_voice, lower_voice, idx):
    # the start and end notes are allowed to be perfect
    if idx in self.start_idx or idx in self.end_idx:
        return True
    # always checked between the strongest measure beat
    # if the index is not on a strong beat,
    # it is therefore accepted
    if idx not in self.measure_idx:
        return True
    if upper_voice[idx] - lower_voice[idx] in PERFECT_INTERVALS:
        # ^The current harmonic interval is perfect
        if self.motion(idx, upper_voice, lower_voice) /...
                        not in ["oblique", "contrary"]:
            # if the harmonic interval is not approached
            # by oblique or contrary motion, it is not valid
            return False
        if self._is_large_leap(upper_voice, idx - 1) or /...
                self._is_large_leap(lower_voice, idx - 1):
            if upper_voice[idx] - lower_voice[idx] == Octave:
                # Octave must be approached by oblique motion
                if self.motion(idx, upper_voice, lower_voice)/...
                                                    == "oblique":
                    return True
            else:
                return False
    return True
```

The second example is a more trivial one, which checks whether the two melodies cross:

```
def _is_voice_crossing(self, upper_voice, lower_voice, idx):
    # possible rests should be ignored
    if upper_voice[idx] == -1 or lower_voice[idx] == -1:
        return False
    if upper_voice[idx] - lower_voice[idx] < 0:
        # if the interval difference between
        # the upper and lower voice is negative,
        # the lower voice is above the upper voice
        # which is not allowed
        return True
    return False
```

4.6.3 Harmonic Rules

Given how most of the harmonic rules are handled implicitly in the formation of the search domain, the list of explicit harmonic rules is limited. In fact, the remaining harmonic rule is only one; the scanning for outlined dissonances in species with a rapid rhythm. The *_harmonic_rules()* method therefore has the following form:

```
def _harmonic_rules(self, ctp_draft, cf_notes):
    penalty = 0
    if SPECIES[self.species] in [3,5]:
        if not self._no_outlined_tritone(ctp_draft):
            self.ctp_errors.append("Outlined dissonant interval!")
            penalty += 50
    return penalty
```

With the *_no_outlined_tritone(ctp_draft)* having the following implementation:

```
def _no_outlined_tritone(self, ctp_draft):
    outline_idx = [0]
    outline_intervals = []
    not_allowed_intervals = [Tritone]
    # between endpoints + step in opposite direction
    dir = [sign(ctp_draft[i + 1] - ctp_draft[i])
              for i in range(1,len(ctp_draft) - 1)]
    for i in range(1,len(dir) - 1):
        if dir[i] != dir[i + 1]:
            outline_idx.append(i + 1)
    outline_idx.append(len(ctp_draft) - 1)
    # Iterate over the outline indices and check if
    # a tritone is found
    for i in range(len(outline_idx) - 1):
        outline_intervals.append(/...
        abs(ctp_draft[outline_idx[i]] /...
        - ctp_draft[outline_idx[i + 1]]))

    for interval in not_allowed_intervals:
        if interval in outline_intervals:
            return False

    return True
```

4.6.4 Dissonance Handling

The last rule-category is that of dissonance handling, which is the set of constraints which is most dissimilar between the different species. In species 2, 3 and 5, all notes on weak beats may contain dissonances if they are properly left and approached. In addition, fifth species has the possibility of having eight-notes, which must be handled separately. Both fourth and fifth species might have dissonances when notes are tied over from the previous measure. If that is the case, the dissonance must be resolved by a downward step. In addition, third and fifth species might have a figure known as the cambiata, which despite breaking rules regarding dissonance, is allowed by Fux. The cambiata figure must therefore also be handled separately.

All these constraints equal the dissonance rules in table 3.6, and result in the following dissonance-handler:

```
def _dissonance_handling(self, cf_notes, ctp_draft):
    penalty = 0
    if SPECIES[self.species] == 1:
        # In first species there is no dissonance,
        # so the allowed harmonic intervals are consonances
        return penalty
    if self.ctp_position == "above":
        upper = ctp_draft
        lower = cf_notes
    else:
        upper = cf_notes
        lower = ctp_draft
    if SPECIES[self.species] in [2,3,5]:
        for i in range(1, len(ctp_draft)-1):
            if SPECIES[self.species] in [3,5] and /...
                self._is_cambiata(i,cf_notes,ctp_draft):
                # allowed
                penalty += 0
            elif self._is_dissonant_interval(upper,lower,i):
                if not self._is_dissonance_properly_handled(/...
                                                i,ctp_draft):
                    self.ctp_errors.append("Dissonance not"+
                             "properly left or approached!")
                    penalty += 100
    if SPECIES[self.species] == 5:
        for i in range(1,len(ctp_draft)-1):
            if not self._is_eight_note_handled(i,ctp_draft):
                self.ctp_errors.append("eight notes"+
                              "not properly handled!")
                penalty += 100
    if SPECIES[self.species] in [4,5]:
        penalty += self._tied_note_properly_resolved(/...
                                      cf_notes,ctp_draft)

    return penalty
```

To illustrate how the dissonance rules might be slightly different between the different species, the implementation of *_is_dissonance_properly_handled(idx, ctp_draft)* is shown. This function checks whether the dissonance has been properly approached and left by step. For first species, the dissonance must be approached and left by step in the same direction. This is, however, relaxed for third and fifth species:

```
def _is_dissonance_properly_left_and_approached(/...
                                    self,idx,ctp_draft):
    current_note = ctp_draft[idx]
    prev_note = ctp_draft[idx-1]
    next_note = ctp_draft[idx+1]
    if abs(next_note-current_note) <= M2 and /...
                abs(current_note-prev_note) <= M2:
        if SPECIES[self.species] in [3,5]:
            return True
        if sign(next_note-current_note) == /...
                 sign(next_note-current_note):
            return True
        else:
            return False
    else:
        return False
```

4.7 MIDI-Generator

The last module is the user-interface, which consists of a midi-generator class and the main loop. The midi-generator is tasked with calling the cantus firmus and counterpoint modules, generating a counterpoint with parameters set by the user. The midi-generator then loads the musical representation to pretty_midi instruments, which is then exported to midi.

The midi-generator constructor is tasked with calling related modules to generate the cantus firmus and species counterpoint:

```
class Midi_Generator:
    instruments = ["Church Organ","Church Organ"]
    def __init__(self,key,scale_name,species,
                 bar_length = 2,ctp_position = "above",
                 cf_range = RANGES[TENOR]):
        self.cf_range_name = RANGES.index(cf_range)
        self.species = species
        self.cf = Cantus_Firmus(key,scale_name,bar_length,
                                voice_range = cf_range)
        self.loaded_instruments = []
        if species == "first":
            self.ctp = FirstSpecies(self.cf, ctp_position)
        elif species == "second":
            self.ctp = SecondSpecies(self.cf,ctp_position)
        elif species == "third":
            self.ctp = ThirdSpecies(self.cf, ctp_position)
        elif species == "fourth":
            self.ctp = FourthSpecies(self.cf, ctp_position)
        elif species == "fifth":
            self.ctp = FifthSpecies(self.cf, ctp_position)
        else:
            print("error: "+species+" is not a valid species")
        self.ctp.generate_ctp()
```

The first method is the *set_instrument(instrument)* method, which set the instrument(s) according to the input. The class supports different instruments for the cantus firmus and counterpoint, which is handled in *set_instrument()*:

```
def set_instrument(self,instrument):
    if isinstance(instrument,list):
        self.instruments = instrument
    else:
        self.instruments = [instrument]*2
```

The second method loads the information contained in the generated cantus firmus and counterpoint to pretty_midi instruments. This is done by first instantiating two pretty_midi instruments, which are then loaded using the lower level *melody.to_instrument()* method:

```
def to_instrument(self):
    inst_number1 = pretty_midi./...
            instrument_name_to_program(self.instruments[0])
    inst_number2 = pretty_midi./...
            instrument_name_to_program(self.instruments[1])
    cf_inst = pretty_midi.Instrument(inst_number1,name="cf")
    ctp_inst =pretty_midi.Instrument(inst_number2,name="ctp")
    self.ctp.melody.to_instrument(ctp_inst)
    self.loaded_instruments.append(ctp_inst)
    self.cf.to_instrument(cf_inst)
    self.loaded_instruments.append(cf_inst)
```

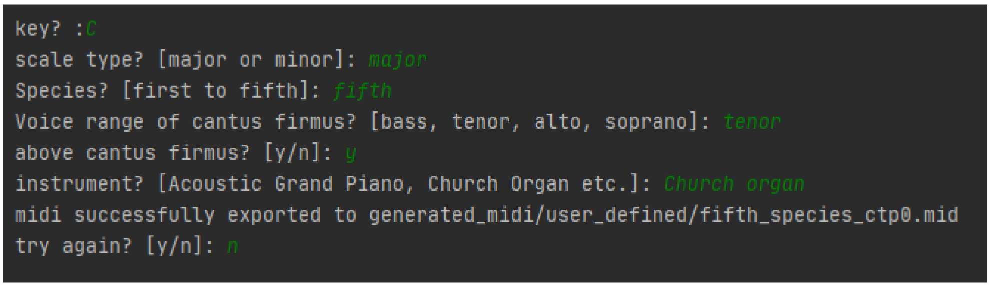
The last method exports the loaded instrument to midi. This is done using the function *pretty_MIDI.write(path)*:

```
def export_to_midi(self,tempo = 120,
                   name = "generated_midi/user_defined/ctp.mid"):
    pm = pretty_midi.PrettyMIDI(initial_tempo= tempo)
    for inst in self.loaded_instruments:
        if inst != None:
            pm.instruments.append(inst)
    pm.write(name)
```

The *main loop* generates species counterpoint according to parameters set by the user. The structure of the main loop has the following implementation, showcased in pseudo-code:

```
    def main():
    cont = True
    i = 0
    print("Automatic Species Generation")
    while cont:
        Input(key, scale_name, species,
              ctp_position, cf_range, instrument)
        name = "ctp"+str(0)
        mid_gen = Midi_Generator(key,scale_name,species,
                                 ctp_position = ctp_position,
                                    cf_range = cf_range)
        mid_gen.set_instrument(instrument)
        mid_gen.to_instrument()
        mid_gen.export_to_midi(name = path+name+".mid")
        print("midi successfully exported to "+path+name+".mid")
        cont_str = input("try again? [y/n]: ")
        if cont_str[0].upper() == "Y":
            cont = True
        else:
            cont = False
```

As an example of valid input, the inputs shown in figure 4.8 generated the results shown in figure 4.9. The audio rendition of the results can be heard here [1].



```
key? :C
scale type? [major or minor]: major
Species? [first to fifth]: fifth
Voice range of cantus firmus? [bass, tenor, alto, soprano]: tenor
above cantus firmus? [y/n]: y
instrument? [Acoustic Grand Piano, Church Organ etc.]: Church organ
midi successfully exported to generated_midi/user_defined/fifth_species_ctp0.mid
try again? [y/n]: n
```

Figure 4.8: Example of valid user inputs in the main loop of the system.

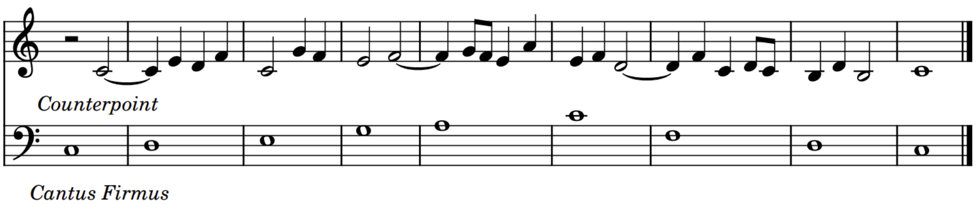


Figure 4.9: Sheet music rendition of the generated result using inputs from 4.8.

[1]Audio of figure 4.9: https://soundcloud.com/johan-gangsas-hole/valid-main-result

Chapter 5

Result

In this chapter some examples of generated results for each of the five different species is given. In addition, an analysis is made by generating batches of 100 counterpoints for each species and inspecting the data. links to audio files of all of the presented results in this chapter are also made available in footnotes[1]. The reader is strongly encouraged to listen to the results, as this gives auditory reference while analyzing the sheet music of the different examples in this chapter.

5.1 Generative Analysis

A generator analysis was made to achieve an evaluation paradigm applicable for all of the different species. This analysis consisted of, for each species, generating 100 randomly instantiated counterpoints. The parameters that are set randomly are as follows:

Key - what key the counterpoint is in. All of the different 12 keys are possible.

Scale - which scale that should be used. The scale can either be major or harmonic minor.

Counterpoint position - if the counterpoint should be above or below the cantus firmus.

[1]Fifth Species example: `https://soundcloud.com/johan-gangsas-hole/fifth-species`

Vocal range - which of the four main vocal ranges the cantus firmus should be in. The different vocal ranges are bass, tenor, alto and soprano.

Some meta-information is extracted for each batch of 100 counterpoints. The information are as follows:

Error - the accumulated error of the generated counterpoint.

List of errors - a list of the different constraints that is not satisfied for the generated counterpoint.

Time - how long, in seconds, it took to generate said counterpoint.

Penalties - the accumulated error for each step in the search algorithm.

5.1.1 First Species Generation

First Species Metric	**Score**
% of generated counterpoint below penalty threshold:	96 %
Average penalty:	30.65
Worst case penalty:	205
Average runtime of search algorithm:	0.27 sec

Table 5.1: Extracted evaluation metrics from the batch of 100 generated first species counterpoint.

Table 5.1 shows the extracted metrics for the 100 counterpoints generated in first species. The error threshold for first species was set to 50. This means that a minor penalty in addition to possible preference penalties regarding contrary motion is allowed. As can be seen in the table, the results for first species generation are satisfactory. Of the 100 counterpoints in the batch, there are only four which are over the error threshold. The worst case penalty is 205, and a sheet music representation of said counterpoint is shown in 5.1. In this example, there are in all three constraints that are broken. The first one is a melodic constraint, and is violated in measures 4-6. Two voice-independence rules are broken in measures 12-13. The corresponding audio representation is made using a church organ as instrument to emphasize the effect of the broken rules[2]. After an initial pleasing harmony in the first three measures, a clear clash is heard during the similar motion between the cantus firmus and counterpoint in measures 4-6. This is further highlighted

[2]Worst case first species:`https://soundcloud.com/johan-gangsas-hole/first-species-worst-case`

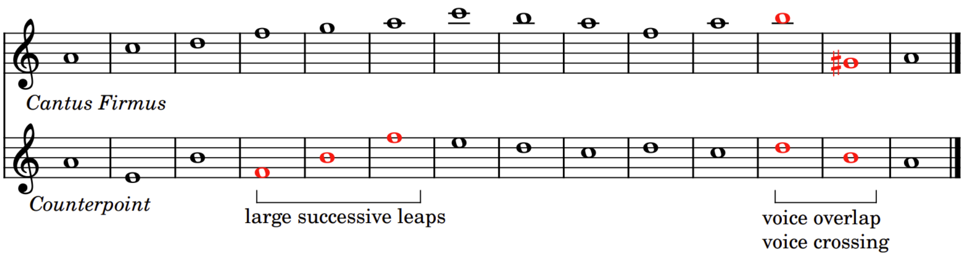


Figure 5.1: Worst case first species. There are in all three constraints that are not satisfied. Two large successive leaps in measure 4-6, and voice overlap and voice crossing in measure 12-13.

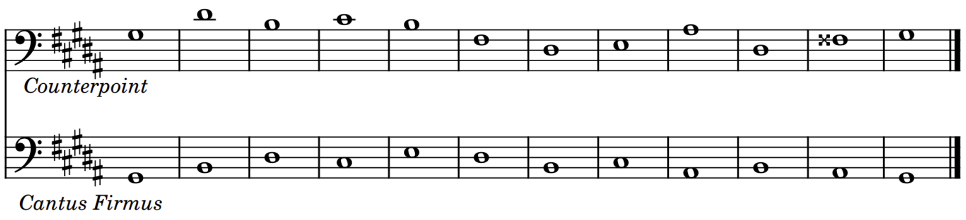


Figure 5.2: First species with zero penalty. Notice how the melodic flow of the cantus firmus helps in guiding the counterpoint in the right direction.

by the two successive large leaps in the counterpoint which is not allowed. The voice overlap and voice crossing in the penultimate measures creates quite a audible dissonance, leading to the resolution in the last measure feeling unsatisfactory.

To give a counter example, figure 5.2 shows a first species example with zero accumulated penalty. The associated audio file is again rendered using church organ as instrument [3]. At first glance it is very similar to 5.1. The difference is the avoidance of excessive leaps and emphasis on similar motion. In this way, the two melodic lines are perceived as being both independent but when combined creating a pleasant sounding harmony.

5.1.2 Second Species Generation

A similar analysis to that of first species is now made for second species. Table 5.2 show the metrics for the batch of second species counterpoint. The error threshold is now increased to 100 to reflect the added complexity and as a factor to lower

[3]Zero penalty first species: `https://soundcloud.com/johan-gangsas-hole/first-species-zero-penalty`

Second Species Metric	**Score**
% of generated counterpoint below penalty threshold:	84 %
Average penalty:	31.5
Worst case penalty:	300
Average runtime of search algorithm:	1.37 sec

Table 5.2: Extracted evaluation metrics from the batch of 100 generated second species counterpoint.

the run-time. As illustrated in table 5.2, the percentage of generated counterpoints below the threshold of 100 is *lower* than for first species. The reason is mostly due to the stricter constraints. The dissonance handling introduced in second species adds a entirely new class of constraints and as a effect a new dimension to the constraint formalism. As illustrated in the worst case example shown in figure 5.3, the stricter penalty for repeating pitches (increased from *bad* in first species to *severe* in second species) has a profound effect on the accumulated penalty, since note repetitions rarely come alone. The corresponding audio clip is, as for first species, a church organ [4]. Notice how the repeated pitches makes the counterpoint more "muddy", while still keeping the harmonization going. This illustrates how a broken rule can have more effect on one musical dimension over another. The melodic fluency of the counterpoint is halted, while the harmonic fluency is barely affected.

The counterexample is a generation with zero penalty, and is shown in figure 5.4. The cantus firmus mostly move step-wise, with the counterpoint mimicking its contour in the first four measures, after which the counterpoint becomes more prone to leaps. As can be heard in the audio [5], the contour imitation by the counterpoint makes it predictable. Luckily, this is compensated by the leap in measure 8.

5.1.3 Third Species Generation

Third species has a much faster rhythm, with four notes in the counterpoint melody for each note in the cantus firmus. This in turn means on average twice the number of notes as in the preceding second species. As can be seen in table 5.3, the algorithm struggles more with finding local optimum. The average run-time is higher than first- and second-species, and the average penalty is above 50. This

[4]Worst case second species: `https://soundcloud.com/johan-gangsas-hole/worst-case-second-species`

[5]Zero penalty second species: `https://soundcloud.com/johan-gangsas-hole/second-species-zero-penalty`

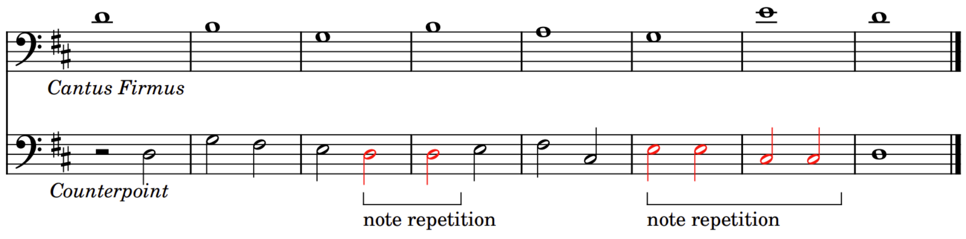


Figure 5.3: Worst case second species. Notice how the harmony keeps on going in oblique motion during note repetitions. In this way, note repetitions are more noticeable melodically than harmonically.

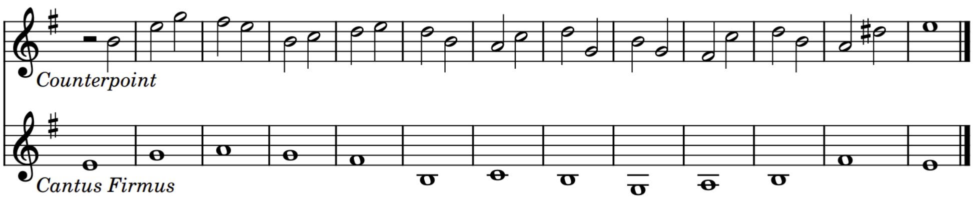


Figure 5.4: Second species with zero penalty This generated second species example has zero accumulated penalty, which results in an interesting counterpoint. Notice especially the dissonance in measure 4 which is properly approached and left by step.

means that on average, the generated results fail to satisfy some of the constraints. The batch of 100 generated third species counterpoint give an indication of what errors that seem to be most common. The most common is *exceeding the range of a tenth* which totals 26 % of all the penalties in the batch. While being classified as a *bad* penalty, exceeding the range of a tenth is subjectively better than excessive note repetitions and wrong dissonance handling.

As for the preceding species, two examples are given. One with zero accumulated penalty, the other the worst of the batch. Given the faster rhythm, the examples are now rendered using piano as instruments [6] [7]. As can be seen and heard in the worst case example in figure 5.5, the counterpoint is prone to excessive note repetitions. In addition, there is both a large leap and unresolved dissonance in measure 1. Despite this, the errors are audibly more difficult to detect given the faster rhythm. The error that is most noticeable is therefore the note repetition in the penultimate

[6]Third species zero penalty: https://soundcloud.com/johan-gangsas-hole/third-species-zero-penalty

[7]Third species worst case: https://soundcloud.com/johan-gangsas-hole/worst-case-third-species

measure.

As seen in figure 5.6, the cantus firmus for the zero penalty example is now in the upper voice. The counterpoint is characterized by bar-wise motivic repetitions as can be seen in measures 5, 6 and 8. While not being an intended feature, it helps at making the melody feel less random.

Third Species Metric	Score
% of generated counterpoint below penalty threshold:	74%
Average penalty:	88.5
Worst case penalty:	550
Average runtime of search algorithm:	1.91 sec

Table 5.3: Extracted evaluation metrics from the batch of 100 generated third species counterpoint.

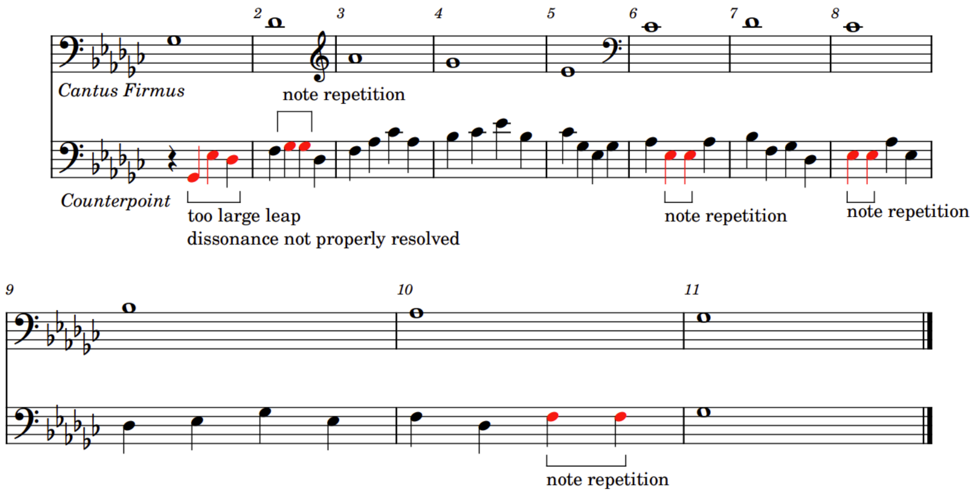


Figure 5.5: Worst case third species. There is a high error rate in the first two measures. This is however masked by the rapid movement in the counterpoint. The most noticeable error is therefore the note repetitions in bar 6, 8 and 10.

5.1.4 Fourth Species Generation

Given the slower rhythm and therefore fewer notes compared to third species, the guided search strategy has a better time finding solutions below the threshold. The threshold is, as in second and third species, set to 100. As seen in table 5.4, the percentage of generated counterpoint below the threshold is at 90%. The average

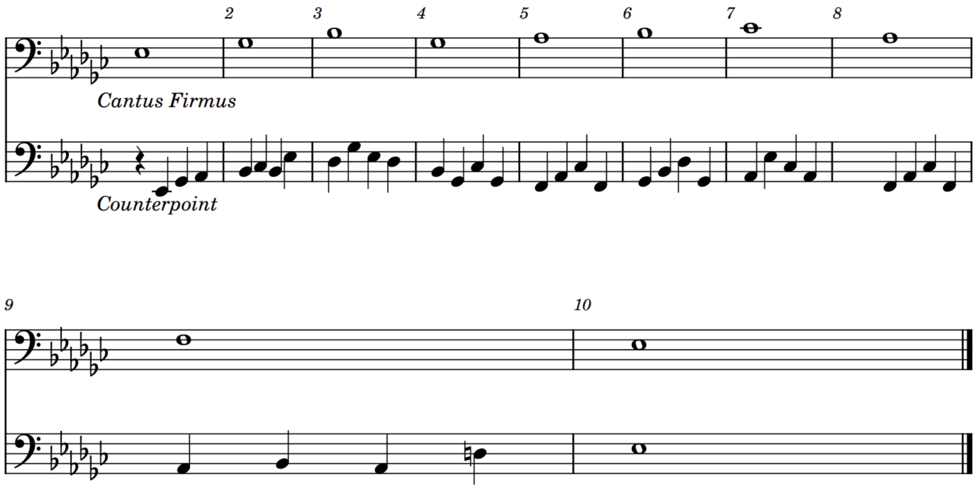


Figure 5.6: Third species with zero penalty. Notice the resolved dissonance in measure 2 and the motifs appearing in measures 5, 6 and 8.

penalty is also quite low, averaging on a score of 25 which corresponds to a minor penalty. The average run-time is comparable to that of second species, which is expected since the rhythm is similar.

The instrumentation is again church organs, as this better fit with the slower rhythm [8] [9]. The worst case example shown in figure 5.7 has a accumulated penalty of 300. This is due to the successive note repetitions on measures 8 and 9, which together total a penalty of 200. In addition, there is a harsh voice crossing in measure 6, which is quite noticeable in the audio rendering. Despite this, the worst case still has some interesting harmonizations, especially before the voice crossing in measure 6.

The zero penalty example shown in figure 5.8 can be viewed as a shifted second species counterpoint with tied notes. The melody is quite uneventful, but the stepwise resolution from measure 11 to the last measure is quite nice and pleasing.

[8]Fourth species zero penalty: `https://soundcloud.com/johan-gangsas-hole/fourth-species-zero-penalty`

[9]Fourth species worst case: `https://soundcloud.com/johan-gangsas-hole/worst-case-fourth-species`

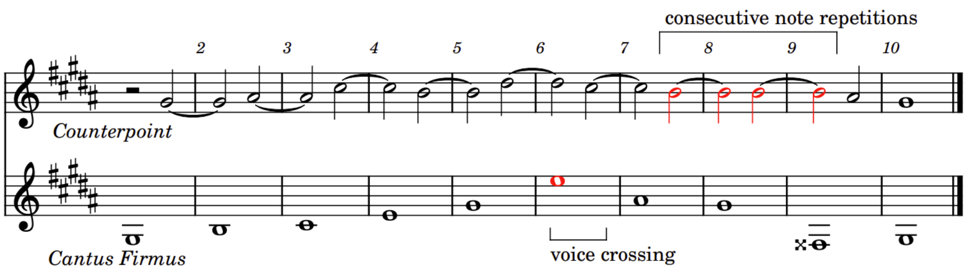


Figure 5.7: Worst case fourth species. The voice crossing in measure 6 is quite noticeable.

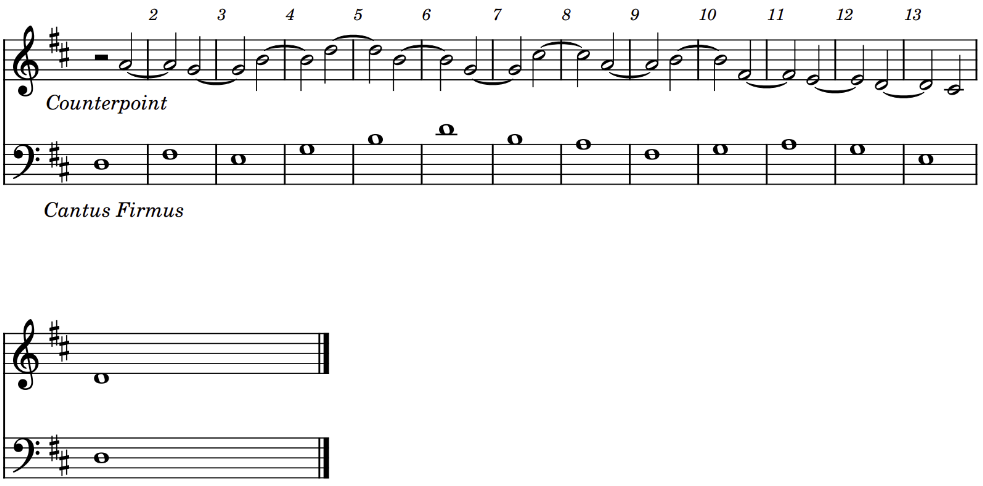


Figure 5.8: Fourth species with zero penalty.

Fourth Species Metric	Score
% of generated counterpoint below penalty threshold:	90%
Average penalty:	24.25
Worst case penalty:	300
Average runtime of search algorithm:	1.41 sec

Table 5.4: Extracted evaluation metrics from the batch of 100 generated fourth species counterpoint.

5.1.5 Fifth Species Generation

Fifth species is the highest level of counterpoint and is therefore, together with third species, the species with the highest number of notes. In contrast with the other types of species, fifth species introduces the possibility of having eight notes. This rapid movement can create some florid melodies, as showcased in the worst case example shown in figure 5.9. The search algorithm does however have some problems in finding correct configurations when using eight notes, and since the rhythm of the fifth species is fixed in the counterpoint object initialization and is immutable, the average error score is quite high. This is shown in table 5.5, together with the highest average run-time so far with close to three seconds for each counterpoint. The worst case penalty is still reasonably low and in the same range as in the other species. The percentage of generated counterpoint below the penalty threshold which is 100 inclusive is also within a reasonable range with its 79 %.

Given the faster rhythm, the counterpoints are again rendered as piano [10] [11]. While listening to the generated results, the auditory differences between the best-case and worst-case is now less noticeable than in the preceding species. This reflects how the faster rhythm helps in masking the errors. In fact, having some dissonances and minor contrapuntal errors can make the melodies more interesting and less predictable. Figure 5.9 shows the worst case example of fifth species. There are in all three eight-notes that fail to be resolved or approached by step. While sounding dissonant, the quick rhythm and subsequent resolution to consonant intervals makes it less noticeable. In addition to the three eight-notes, there is also a unison between the start and end measures, located in measure 7. The counterpoint also exceeds the range of a tenth, which when rendered as piano is not as noticeable as if it were to be rendered as vocals.

[10]Fifth species zero penalty: https://soundcloud.com/johan-gangsas-hole/fifth-species-zero-penalty

[11]Fifth species worse case: https://soundcloud.com/johan-gangsas-hole/worst-case-fifth-species

Fifth Species Metric	Score
% of generated counterpoint below penalty threshold:	79%
Average penalty:	85
Worst case penalty:	400
Average runtime of search algorithm:	2.93 sec

Table 5.5: Extracted evaluation metrics from the batch of 100 generated fifth species counterpoint.

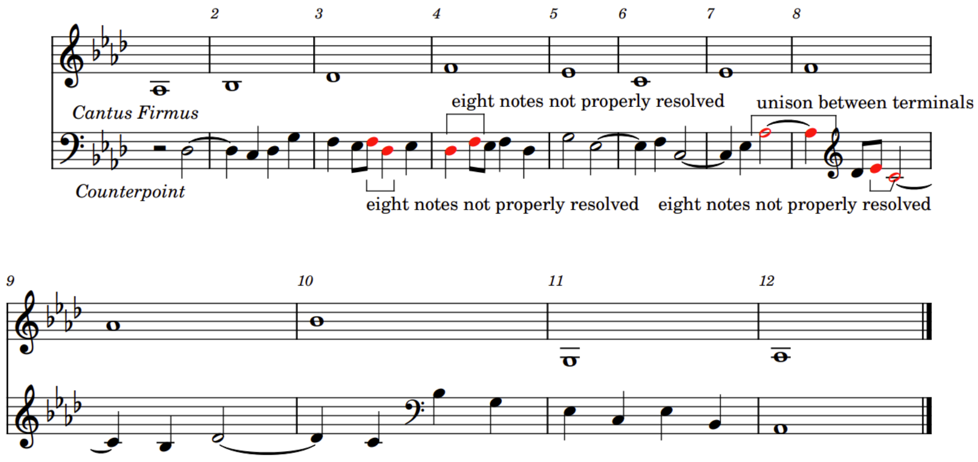

Cantus Firmus
Counterpoint
eight notes not properly resolved
unison between terminals
eight notes not properly resolved
eight notes not properly resolved

Figure 5.9: Worst case fifth species. The dissonant eight notes creates tension in the melody, but due to the fast rhythm this is quickly resolved to consonances.

The zero penalty example in 5.10 has no eight-notes, but the melody is still florid. The tied half-notes into quarter-notes in measures 5 and 6 especially have a pleasant effect on the melodic flow, making it feel like the counterpoint waits for the cantus firmus to catch up. The steady increasing contour of the cantus firmus also have a nice effect on the overall perceived "goodness" of the counterpoint. It helps in keeping the counterpoint more goal oriented, resulting in the climax in measure in 9.

Additional examples for all of the five different species can be found in appendix A.1. All of the generated counterpoints have links to an audio rendering, using a choir for the slower rhythms in species 1, 2 and 4. Species 3 and 5 are rendered using piano. Each species have a total of five examples generated randomly.

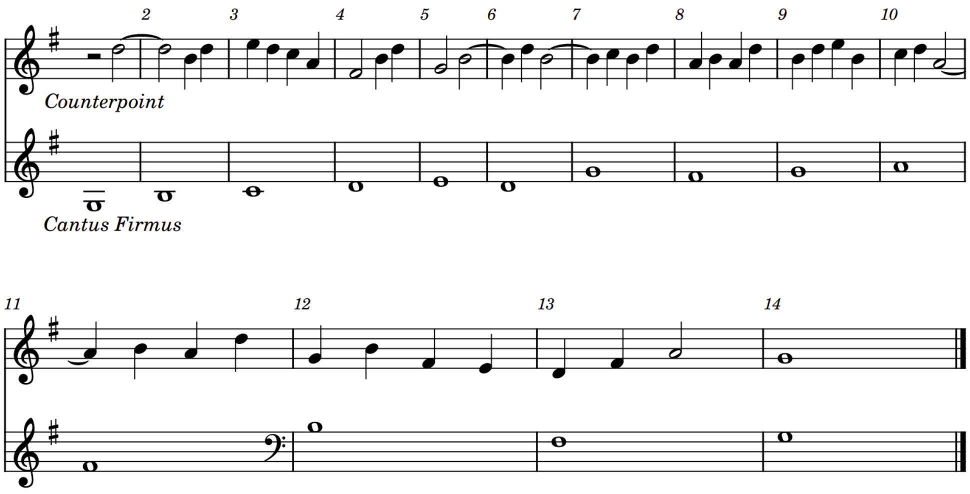


Figure 5.10: Fifth species with zero penalty. The counterpoint is characterized by a quick and syncopated rhythm. This creates tension and interplay between the counterpoint and cantus firmus which leads to a rich contrapuntal texture.

Chapter 6

Discussion

6.1 Constraint Optimization as System Structure

The iterative process of finding a good structure in the early development phase was characterized by much trial and error. During this phase, tests were made to generate the counterpoint by *building* the melodies iteratively note by note. In this stage, the cantus firmus was predefined as the start of "twinkle twinkle little star" in C-major. The results were promising for first species counterpoint, with an adequate melodic flow and movement. An example of such an early generation is shown in figure 6.1. However, when we tried to extend the system to include the generation of second species counterpoint, building the counterpoint note for note became increasingly difficult. Instead of supplementing the already implemented rules, the rules for second species had to be restated to fit with the changed rhythm (2:1 instead of 1:1). The rules also started to deviate from the fixed rules of Fux, and the system became more and more prone to ad-hoc solutions and "rules with exceptions" to be able to generate results. Some satisfactory results were eventually generated, but it was apparent that this methodology would not lead to a scalable system with a clearly defined rule-set. This also clarified that separating the rhythm and melodic pitches would lead to a more modular and structurally expandable system, with no need to restate rules for each species rhythm.

The design was therefore taken in a different direction. After restating some of the rules as boolean expressions enforced upon a randomized counterpoint, the generator managed to generate first species counterpoint after randomly trying different note combination until one satisfying all of the constraints were found. This brute force approach of the second system iteration was neither optimal nor elegant, but it paved the way for a different and more scalable system structure. While the more

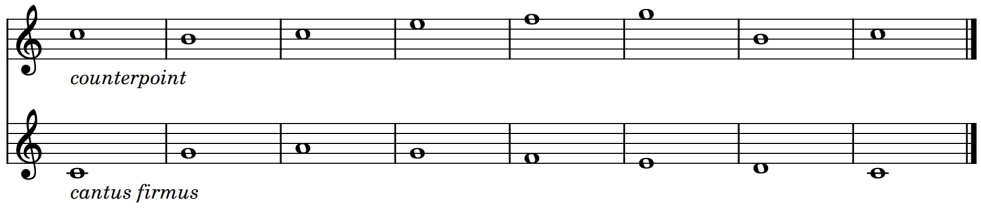


Figure 6.1: Example of first species generation using the procedural approach.

procedural first approach told the program *how* to generate the results, the latter approach told the program *how not to* generate the results. A more constraint-based system, where the constraints were issued on a given randomized draft of a counterpoint, led to a more formalized rule structure. Rules could now easily be appended to the program without restructuring the system, which was the case for the first procedural approach. Adopting the *constraint optimization* paradigm gave quick progress in implementing the different modules necessary and proved to be a valid structural design to realize a system for automatic generation for all of the five species.

By choosing to adopt the constraint optimization paradigm, many of the components needed could be outlined. This modular design also gave a natural flow to both the design and implementation. The different modules were tasked with handling different music dimensions. The counterpoint module was assigned to rhythm and ties generation, and defined the melodic search space. The cantus firmus module was an independent component for cf generation. The search strategy was tasked with finding an appropriate melody, expressed as a pitch sequence with its search domain pre-constrained by the counterpoint module. The music representation contained the musical objects necessary to give the other modules a common data structure. The constraints module quantified the "goodness" of melodies, and lastly, the midi-generator generated the complete result given parameters by the user. Having these modules outlined in the design before the second, constraint optimization based, system iteration begun meant a lot to the progress. Knowing at a high level what the module should do was crucial for upgrading and testing the various components quickly without having to restructure and change features in other modules. This became especially true in the search algorithm module, where different search strategies were tried out.

6.2 The Effect of the Abstract Design of the Music Representation

The choice of developing the music representation as the first module was done as to define the data structures necessary for the other modules. Now in retrospect, this approach could probably have benefited from being done differently. While it gave a quick initial progression to the system implementation, the granularity of the musical objects was hard to specify. Both the search strategy and the different classes related to the different species had yet to be determined. Therefore, identifying the musical information necessary to express the wanted search domain adequately was a non-trivial task. As a result, some of the functionality in the music representation proved to be redundant. This was especially the case with the interval class. While providing useful analytical functionality, it was easier to analyze intervals by computing the different pitch values between notes directly instead of instantiating interval classes. In the *melody class*, it also proved satisfactory to express the pitch sequence as a list of integer values instead of note objects, which simplified the data used in the search strategy. The rest of the music representation did however prove to be satisfactory and usable by the other modules.

Given the importance of a well-structured music representation, the initial design using the more procedural approach consisted of using a preexisting library. Tests were made using the library *music21*, a Python toolkit for computer-aided musicology. While it provided functionality for both sheet music and midi exportation, the higher-order musical objects for melody construction proved to be too complex to be used as a data structure to be constrained. While being very expressive and user-friendly, *music21* made it hard to formalize the search space in a format that made it feasible for a search strategy to find reasonable solutions. Therefore, the use of preexisting libraries in the music representation was abandoned, except for the utility functions provided by the *pretty_midi* library for easy midi-conversion.

As the melody class represents the highest abstraction in the music module, it defines the format of the music representation to be passed to the search strategy. After testing various representations, including a sequence of note objects, a simple list of respective MIDI-numbers for a given melody proved to be sufficient. It is therefore just a sub-structure, although important, of the melody structure that is actually used as the music representation in the search strategy. This was done for several reasons. By abstracting away the list of corresponding note durations, focusing rather on the MIDI pitch representation, the search strategy could be more generalized. Instead of having to handle both rhythmic and melodic dependencies, the solver is only concerned with the sequence of MIDI pitches of the generated melody. The rhythmic difference between the different species is handled separately for each species in the counterpoint module. The main justification for the removal of the rhythmic information in the search strategy was that the same search algorithm could be used for all of the different species.

6.3 The Isolated Cantus Firmus Module

The cantus firmus module turned out to be the most isolated part of the system. The original idea was to also include the cantus firmus in the search strategy, defining it as a lightweight form of species counterpoint. However, while the rules are similar to that of the melodic rules for first species, the cantus firmus must be built note by note by continuously comparing the current pitch to the next pitch. While requiring similar search strategies, this distinction meant that the system structure would benefit from having the cantus firmus generated separately. In this way, implementing functionality in the search algorithm not usable for the remaining species was avoided. Another justification was the simplicity in generating cantus firmus compared to that of species counterpoint. While a CF can be built successively pitch by pitch, species counterpoint is more comparable and dependent on a cantus firmus. Therefore, the cantus firmus generator could have a more lightweight search algorithm, with a fast convergence to a total penalty of 0.

The focus on species counterpoint over cantus firmus generation might have been taken a bit lightly. The CF melodies could probably have benefited from being constrained by more preference-defined rules. Such rules might have included giving the melody a more evident contour to make it easier to find feasible counterpoint melodies to the cantus firmus. Nevertheless, the subtle randomness in some cantus firmus melodies help to make the counterpoints more exciting and less predictable. So while it for some cases may give the counterpoint generator a hard time converging to reasonable solutions, the end result is still musically interesting.

6.4 Constraint Formalism

after adopting the constraint optimization paradigm, the first sketch of the **counterpoint structure** included classes for all the species similar to the one for cantus firmus generation. This meant that all the necessary functionality for generating the given species was contained within its object. In this way, each species included its own set of constraints and cost function. However, this rule-categorization by species rather than by the commonality between them led to more redefinition of rules than first expected. The assumed modular independence between the species became more and more blurred as the rules became shared between them. This, in effect, led to ill scaled program maintainability. Therefore, a choice was made to separate the constraints from their respective species entirely, creating a constraint module of its own. After the initial modular testing with species and their respective constraints contained within the same class, it also became apparent that a further rule categorization was in place. Since the list of constraints was quite extensive, it was expected that implementing each rule one by one with no further segmentation would lead to maintainability issues. It was during this time that the analysis of the Fux rules, as presented in section 3.6, was made.

By identifying what rules were common across all the species and categorizing them by what musical dimension they checked, the constraints' structure became clearer. The four proposed categories; melodic, voice-independence, harmonic and dissonance-handling, proved to be a valid categorization for the Fuxian rules. Spending time identifying what rules to include and what rules to omit during the design phase also made the implementation easier. Having the constraints separated into different categories made the system more organized and made the inclusion of remaining species simpler. A formalized plan led to quick progress in outlining the different rules, resulting in quick testing of the constraint module. Having each rule quantified as a weighted boolean expression came very naturally, and the constraints provided by Fux proved to be a ready-made case for a knowledge-based system. Each rule was implemented and tested using a counterpoint pitch sequence with known bugs. In this way, identifying and fixing errors in the implementation was easy since the expected functionality was quickly determined to be either satisfied or unsatisfied. Therefore, although there is a high number of constraints, the implementation was mostly straightforward.

A feature that proved invaluable during testing was the list of errors in string format. When testing the interface between the constraints module and search module, knowing what rules reappeared for multiple counterpoints helped pinpoint possible weaknesses in the structure of the constraints. An example is illustrated from the implementation of third species. The search strategy seemed to return multiple counterpoints with the "motivic repetition" error broken successively throughout the pitch sequence. While the initial idea was to check the implementation of the boolean expression associated with the "motivic repetition" rule, an observation was made when identifying how there seemed to be no rule breakings regarding dissonance handling in none of the generated counterpoints. Therefore, the bug was located in the dissonance handler rather than the implementation of the broken constraint. This also illustrates the profound effect that one rule constraint can have over another. When the constraint module failed to identify possible dissonances, it forced the counterpoint to repeat motives as the only way to avoid the more strict "repeat pitches" rule.

6.4.1 Cost Function

How one rule seem to affect another is also an effect of the penalty categorization that was formalized in the system, consisting of *preference*, *minor*, *bad* and *severe*. The granularity of the different penalties was sufficient, but now in retrospect, some of the rules might have benefited from being less strict. As an example, the counterpoint rules in Schottstaedt's [32] system are penalized using a wider range on the weights. For Schottstaedt, the rule regarding parallel motion to octaves have a penalty of 200, while note repetitions only have a penalty of 4. Therefore, adopting a penalty hierarchy more in line with "acceptable" and strictly "not-acceptable" rules could have helped the search strategy to find valid solutions faster. Nevertheless, the penalties in the implemented system reflect the severity as defined by Fux, so changing the weights of the rules would lead the system away from the Fuxian style and more towards tonal and more free counterpoint. A higher run-time was therefore preferred over making the counterpoint less style-adherent.

6.5 The Structure of the Counterpoint Module

The first approach to the structure of the counterpoint module was approached quite differently than the proposed design and implementation. Instead of limiting the functionality to only pre-constraining the search space, the original design for each species contained its own initialization, constraint formalism and search strategy. The design was therefore quite similar to that of the cantus firmus. The constraints were implemented for the species in which they belonged. This approach had certain advantages, with one being the clear partitioning of the different species and associated functionality. One clear disadvantage, however, was the need to restate and copy already implemented constraints into the new species. These restatements with only minor alteration lead to an increasingly hard maintainability. This approach also meant that the search strategy had to be slightly altered depending on what species of counterpoint it had to search for. As a result, stress-testing and identifying bugs became harder as the modules became more co-dependent. Constraints used in multiple species also led to some hard-to-track inter-dependencies between the modules, making progress and adding new functionality slow. A decision was therefore made after first- and second-species testing to separate out the functionality common for all of the different species. The segmentation also led to a more formalized and better structured search- and constraint-module. The counterpoint module was therefore only left with creating the list of possible notes to be passed to the search algorithm, which proved to sufficient for the proposed system.

Designing each species class to have the same set of methods also helped during implementation and to keep the different species module-based. After defining the methods necessary for first species, it was possible to adopt a similar structure for the remaining species, generating the species rhythm, ties and list of possible notes for each slot in the pitch sequence.

6.6 The Choice of Search Algorithm

The search strategy module was the system component that was subject to the highest amount of iterations. A total of three search strategies were explored before landing on the design presented in 3.7.

Brute Force

As was briefly mentioned in the design section, we first tried a novice brute force approach for first species generation in the early stages of the second system iteration. The search strategy was tractable since the number of pitch sequences to be explored by the search algorithm was limited. It was, however, known during this time that this approach would not be suitable for the remaining species. Still, the novice approach helped identify issues regarding the interface between the cost function and search strategy. Therefore, the brute force approach was used during testing of the constraint module to see if it was even possible to find a pitch sequence with zero accumulated penalty. After all of the known bugs in the constraint module were fixed, the brute force approach had served its purpose. It was now possible to improve the search strategy knowing that possible non-convergence to valid pitch sequence was the fault of the search module, and *not* the constraint module. In this way, the search module could be implemented separately, knowing that the other modules were implemented correctly.

Picking The best Local Option

One important observation was the exponential increase in possible paths when handling more complex rhythms. Even a worst-case first species counterpoint of length 14 can potentially have $6 * 10^9$ different pitch sequences. The number of possible paths for the remaining species, therefore, becomes intractable.

It was therefore clear that the search algorithm would have to adopt a search strategy that increased linearly and not exponentially by the number of states. By observing that each note in the pitch sequence can be one of several possibilities, a test was made to pick the best local option for the different pitch slots in the pitch sequence. This prototype strategy continuously scanned the pitch sequence, starting with the first note and ending on the last. In this way, the cost function quickly converged to a local minimum. However, the local optimum was often above the minimum allowed threshold value. This led to the algorithm often reaching a dead end, with no way to exit the sub-optimal local minimum. The local search had to be improved somehow, making it possible to exit the dead ends and traverse different paths. To achieve this, two approaches were tested.

Backtracking

The first algorithm was backtracking. After the local search had scanned the pitch sequence until convergence to a local optimum, the algorithm traversed new paths starting with changing the end note. If the total penalty was still over the allowed threshold, the algorithm moved one step back, and explored new paths from that position and picking the best path. While this approach converged to valid solutions for both first- and second species, it was extremely slow due to the number of paths it had to scan before breaking out of the sub-optimal local minimum. A new approach was therefore explored to try to break out of the local minimum at a faster rate.

Variable Window Search

The next approach was a variable window search, similar to the initial best local option algorithm. The difference was the handling when the algorithm reached a dead end. In the variable window search, the *window* of best-option-combinations is increased. This means that while the best local option algorithm only picked the best note in the given pitch layer, the variable window search increased the width of the search to include additional layers. Instead of finding the best singular option, it found the best combination of two and two notes. If the solution still did not converge to a valid minimum below the threshold, the search width was further increased by one. This meant that three and three notes were searched and picked by the lowest accumulated penalty.

This approach, compared to backtracking and best local search, did converge to valid solutions for all of the five different species. But despite converging, it was very slow, and it had a hard time to identify bad paths and / or bad combination of notes. It also did not scale well with the increased note numbers for third and fifth species, making the search slower and slower for each complexity level. This approach made it clear that an extensive search approach was not the way to go. While having moderate convergence rates for first, second and fourth species (below 2 seconds), it often ground to a halt with the added complexity of the quicker third and fifth species rhythms (15-30 seconds).

Based on the information obtained by exploring the different options, a choice was made to further improve upon the first outlined algorithm, namely the *best* *local search*. The improvement was done by guiding the local search out of sub-optimal local minima by changing the notes which accumulated the most amount of local penalty. This led to the outline of the algorithm presented in 3.7 and 4.5.

6.7 Auditory Quality - Some Musical Remarks

Since the generated results from the implemented system are purely symbolical, represented as events in midi-format, all of the provided audio-renderings are done using the external program *musescore 3*. This also emphasizes an interesting point regarding the perceived "goodness" of a musical score as an effect of the sound quality. If we instead were to use dated midi-programs such as vanBasco's MIDI Player from 2006, the generated results would have sounded more trivial. To illustrate, listen to the difference in musical quality of the two audio renderings of the same fifth species counterpoint shown in figure A.22. The first one is generated using VanBasco[1], the second using musescore 3[2]. Therefore, external musical enhancements such as dynamic playing, reverb and more exclusive soundfonts can breathe life into another vice bland musical score. This also underlines the subjective nature of music generation evaluation, further justifying the choice of expressing the generated results mainly as sheet music and using audio renderings purely for musical reference and not an evaluation metric.

6.8 Future Work

6.8.1 Improving the System

In the proposed system, not all the rules of fifth species counterpoint has been explicitly stated. A natural next step would therefore be to include these rules, staying true to how Fux presented them. This would mean to formalize a way to further quantify these rules, since they were subject to a lot of exceptions. An improvement of the rhythmic generator for fifth species would also be in place, as to better reflect the rhythmic rules such as that higher notes should have longer note durations and slower rhythms should be followed by faster rhythms and vice versa. This would also mean a restructuring of the search strategy, as to guarantee convergence to feasible solutions with the added complexity of expanding the rule-set.

It would also be interesting to compare how well the developed search strategy performs compared to other similar solutions. Schottstaedt [32] and Herremans et al. [10, 11] both approach the problem in a similar manner, using a constraint

[1]Fifth Species C minor using VanBasco: https://soundcloud.com/johan-gangsas-hole/fifth-species-c-minor-vanbasco

[2]Fifth Species C minor using musescore 3: https://soundcloud.com/johan-gangsas-hole/fifth-species-c-minor

formalism, cost function and search strategy. Similar to all three solutions (including the system presented in this thesis), is the use of a metaheuristic search strategy to find the best *possible* solution within both rule-constraints and time-constraints. Herremans et al. uses a variable neighborhood search, and Schottstaedt uses a recursive best-first guess approach. Doing a more thorough analysis comparing the proposed system with the structure of similar solutions could aid in identifying possible strengths and weaknesses of the developed system in this thesis.

6.8.2 Expanding the System

Given the modularity of the designed system, several promising directions can be explored as a natural next step. Fux and Jeppesen, as an example, present additional rules for adding up to four simultaneous sounding voices. The proposed system in this thesis could, therefore, be expanded to include functionality to support this. This would result in richer harmonies, with the possibility of developing a more optimized search strategy.

Given the structure of the proposed system, a minor test was made to explore the possibility of adding a third voice. Note that this was done purely for personal curiosity, and was not intended as a feature in the final system. The test generation is shown in figure 6.2, and consists of one cantus firmus and two counterpoints in fifth species. The result sound surprisingly pleasing despite not having included additional rules. These initial results further motivates the inclusion of more voices.

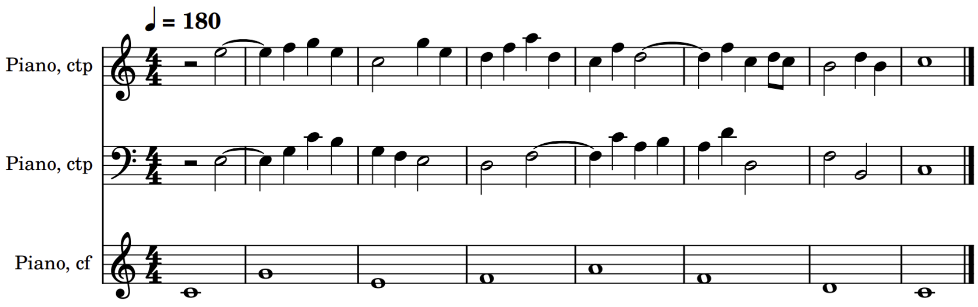


Figure 6.2: Test with three voices in fifth species, audio rendering: https://soundcloud.com/johan-gangsas-hole/three-voices-test2

Another direction is to continue to develop the rules of counterpoint, taking the generation to a step beyond fifth species. This would result in a more free counterpoint, in style with the great composer like Bach during the baroque and early classical era in music. In free (or tonal) counterpoint, the strictness regarding motivic repetition is relaxed, and the rule regarding that the cantus firmus must be in whole notes is disregarded. Free counterpoint, therefore, moves away from the pedagogical style of species counterpoint, introducing compositional techniques used by the great composers. Exploring this direction would consequently result in a more style-imitative generation rather than the proposed auto generative nature of this system.

Disregarding the cantus firmus would also result in a more florid rhythmic development, opening up for the possibility of having both *style-imitation* and *form-imitation*. With *form* we mean the higher-level structure of the composition. One form often found in Bach music is *the fugue*. In this contrapuntal composition technique, a musical idea expressed as a melodic phrase is developed and imitated in other voices while maintaining melodic independence. In this way, a fugue can be viewed as a series of counterpoints exploring and developing the same musical idea. Given how modern algorithmic composition systems seem to be concerned with trying to represent larger musical structures (as presented in 2.2.1), forming a fugue generator would be a fruitful endeavor as it has relevance to state of the art in procedural music generation.

Chapter 7

Conclusion

In this thesis, a system capable of generating each of the five levels of species counterpoint was developed. By representing the counterpoint melodies as a sequence of integer values, it was possible to formalize a general search strategy and cost function usable for all of the different species of counterpoint. This was achieved by abstracting away the other musical dimensions such as rhythm, instrumentation, and tempo from the search algorithm. Therefore, by having different modules focus on different musical features, the complexity is evenly divided across the system's main components. For each of the five species, the local search strategy finds satisfactory results that sound pleasing, with the accumulated penalty averaging below a severe penalty.

Quantifying the Fuxian rules as weighted boolean expression proved to be a logical and maintainable constraint formalism. In this way, concurrent implementation and testing was achieved, and the list of rules could be quantified in quick succession. Having objects express different musical structures also proved to be a valid music representation, as it gave the modules a common musical format.

An interesting continuation of the presented work would be to add more voices to the contrapuntal structure, resulting in richer harmonies. Fux presents in his work additional rules that support up to four voices, and the system can be extended to include these constraints. Some results also border on baroque tonality, especially the florid melodies in fifth species. An interesting next step would therefore be to try to formalize a richer tonal counterpoint in Bach-style. An analysis of how the developed local search strategy performs compared to other approaches would also be an important topic for further research.

Appendix A

Generated Counterpoints

A.1 First Species

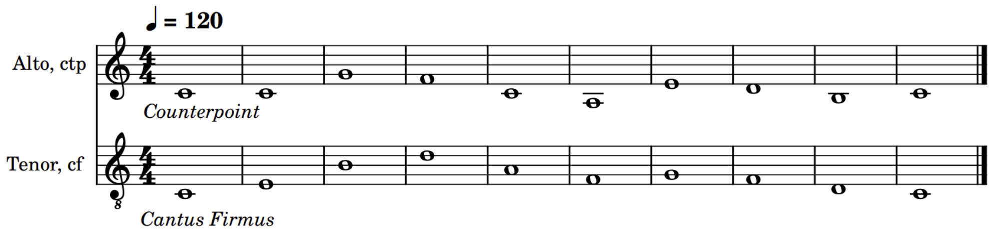


Figure A.1: First Species in C major audio: https://soundcloud.com/johan-gangsas-hole/c-major-tenor-above

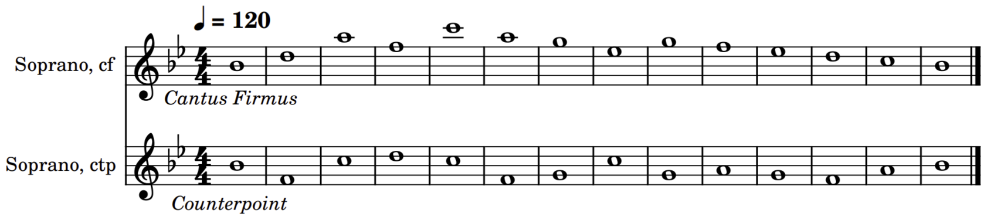


Figure A.2: First Species in Bb minor audio: https://soundcloud.com/johan-gangsas-hole/bb-major-soprano-below

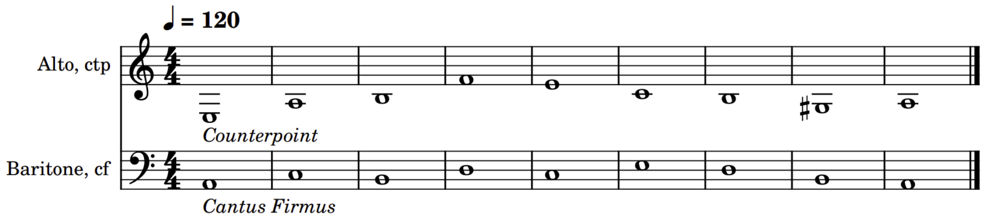


Figure A.3: First Species in A minor audio: https://soundcloud.com/johan-gangsas-hole/a-minor-bass-above

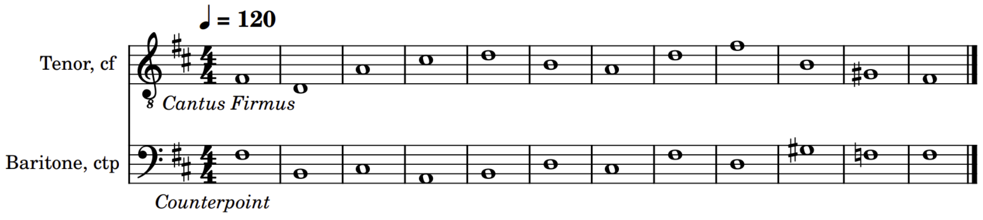


Figure A.4: First Species in F# minor audio: https://soundcloud.com/johan-gangsas-hole/f-minor-tenor-below

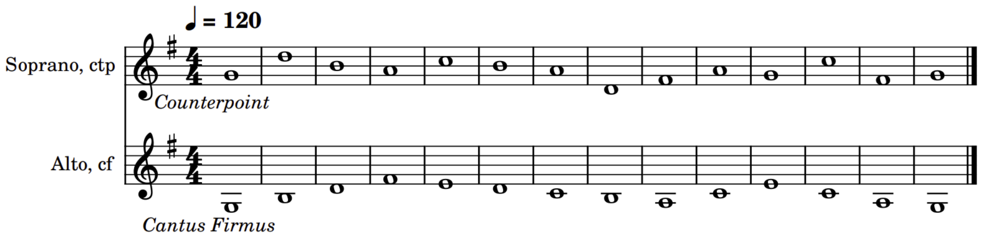


Figure A.5: First Species in G major audio: https://soundcloud.com/johan-gangsas-hole/g-major-alto-above

A.2 Second Species

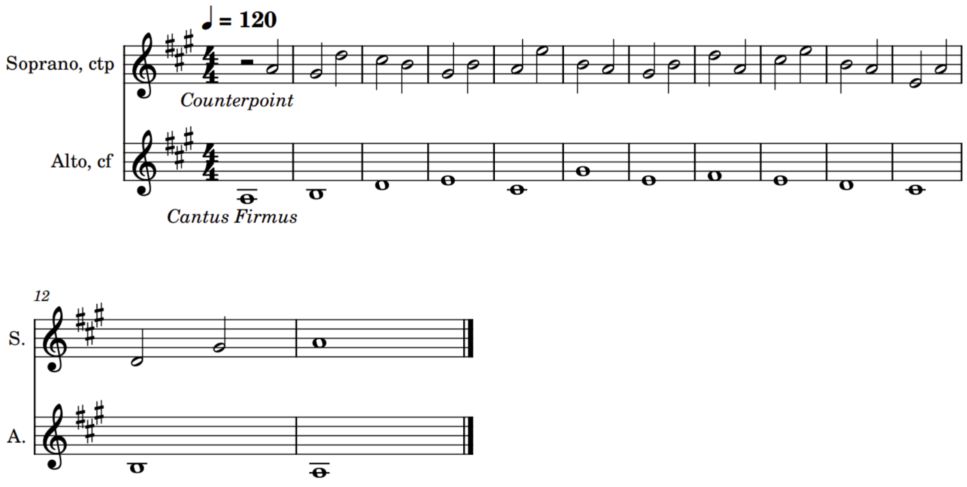


Figure A.6: Second Species in A major audio: https://soundcloud.com/johan-gangsas-hole/second-species-a-major

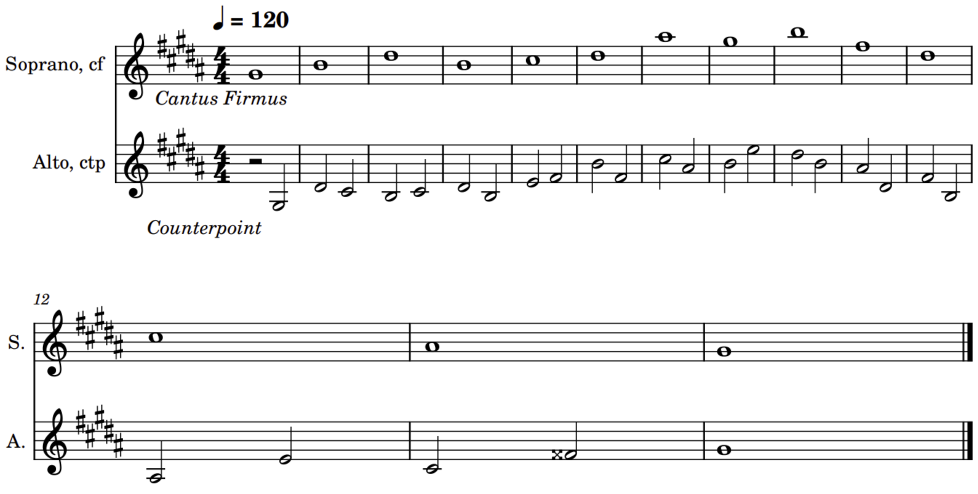


Figure A.7: Second Species in Ab minor audio: https://soundcloud.com/johan-gangsas-hole/second-species-ab-minor

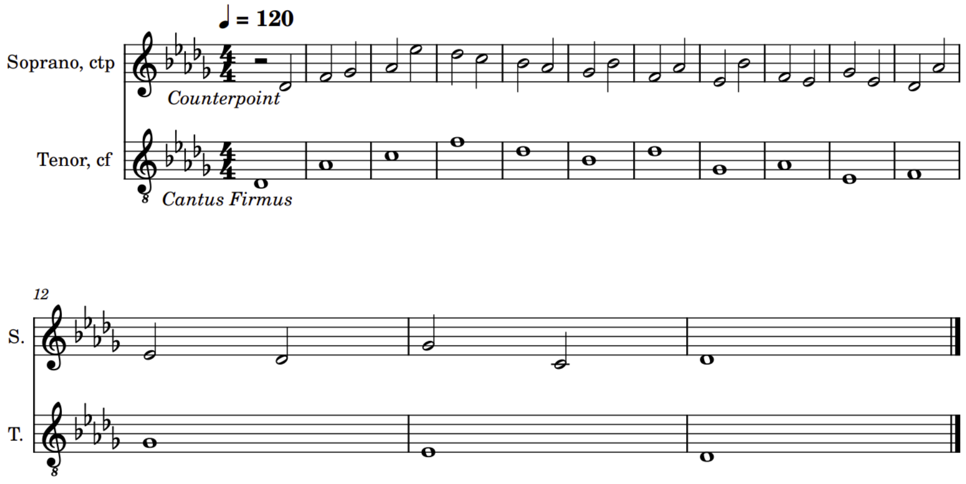


Figure A.8: Second Species in C# major audio: https://soundcloud.com/johan-gangsas-hole/second-species-c-major

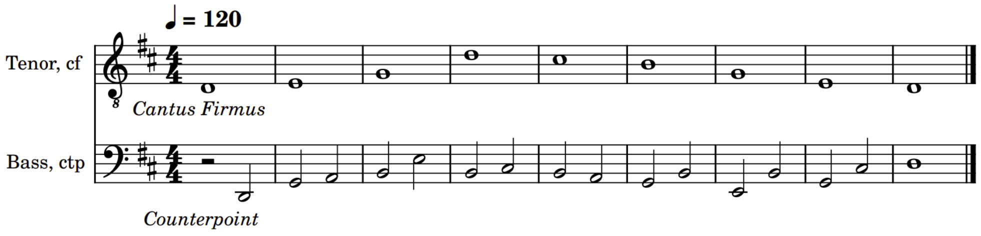


Figure A.9: Second Species in D major audio: https://soundcloud.com/johan-gangsas-hole/second-species-d-major

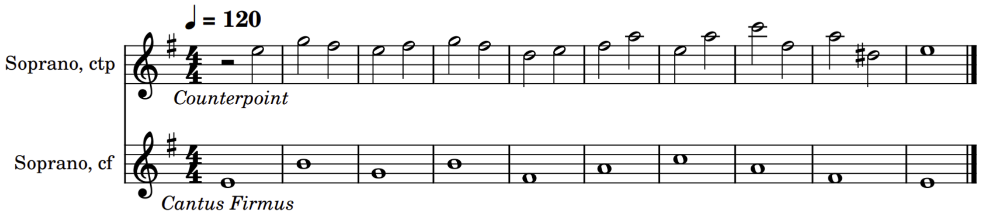


Figure A.10: Second Species in E minor audio: https://soundcloud.com/johan-gangsas-hole/second-species-e-minor

A.3 Third Species

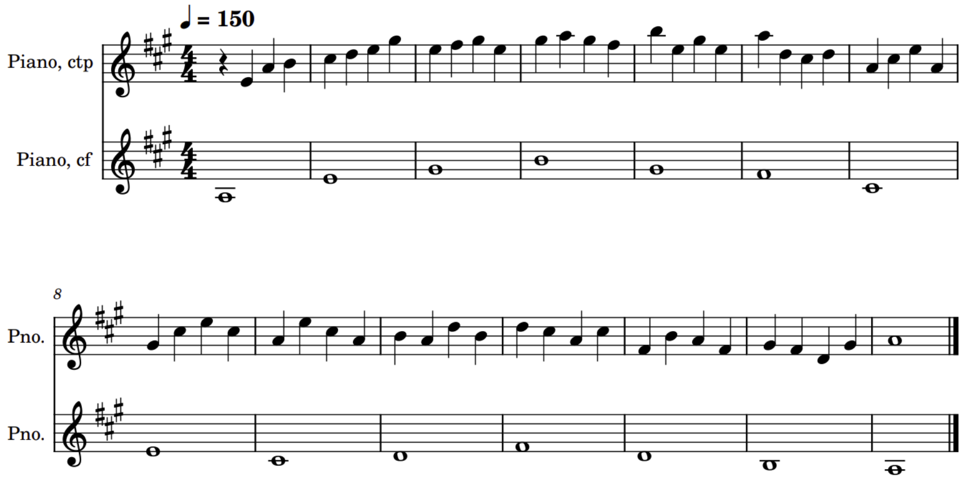


Figure A.11: Third Species in A major audio: https://soundcloud.com/johan-gangsas-hole/third-species-a-major

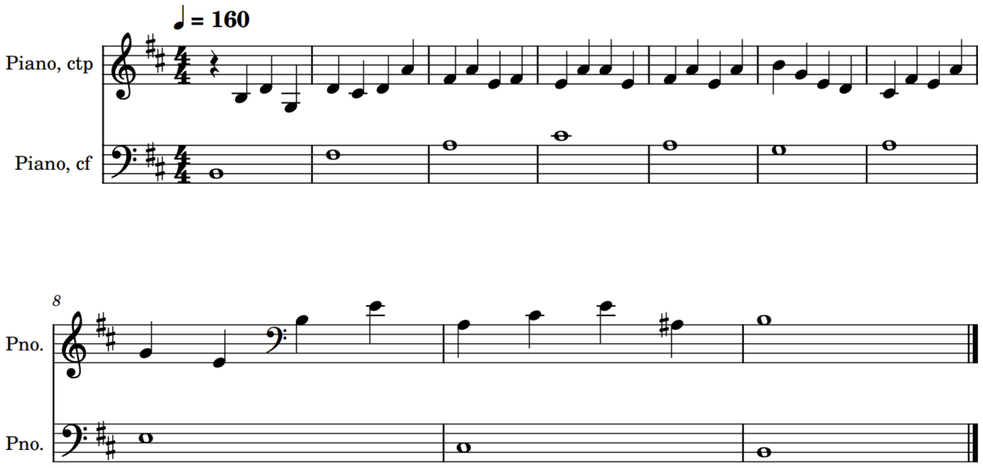


Figure A.12: Third Species in B minor audio: https://soundcloud.com/johan-gangsas-hole/third-species-b-minor

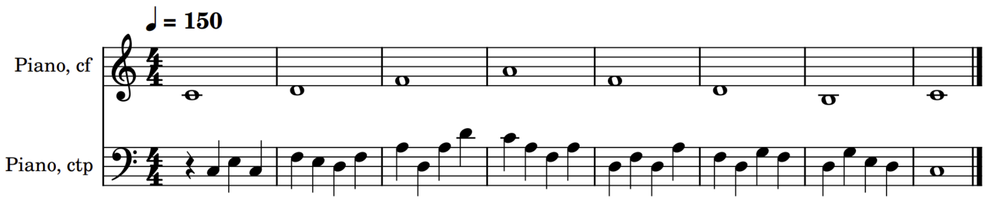


Figure A.13: Third Species in C major audio: https://soundcloud.com/johan-gangsas-hole/third-species-c-major

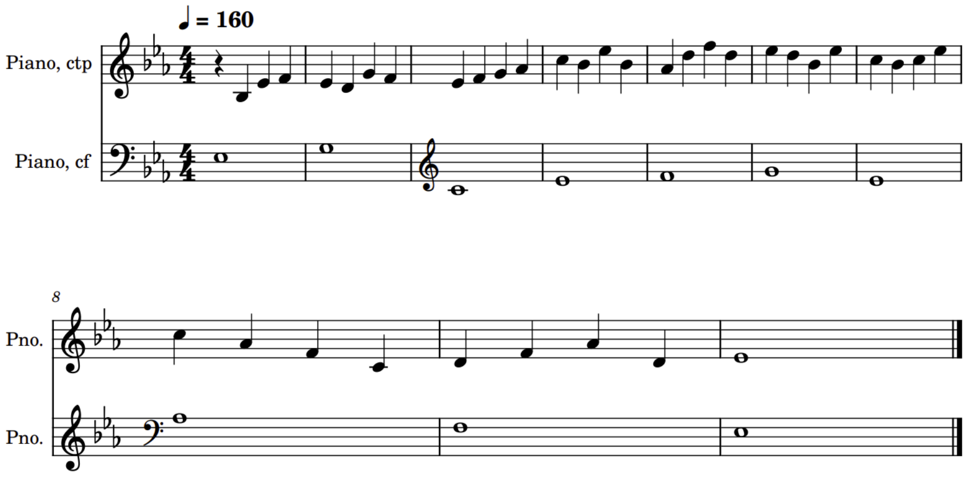


Figure A.14: Third Species in D# major audio: https://soundcloud.com/johan-gangsas-hole/third-species-d-major

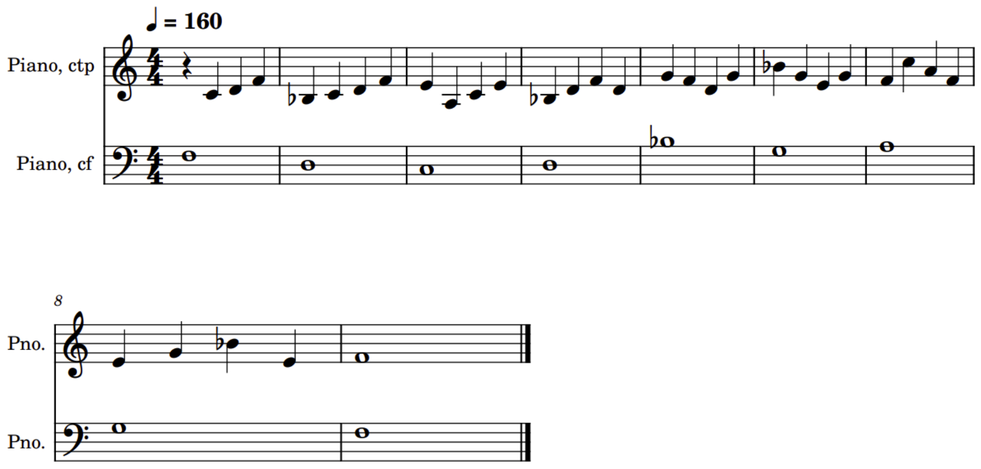


Figure A.15: Third Species in F major audio: https://soundcloud.com/johan-gangsas-hole/third-species-f-major

A.4 Fourth Species

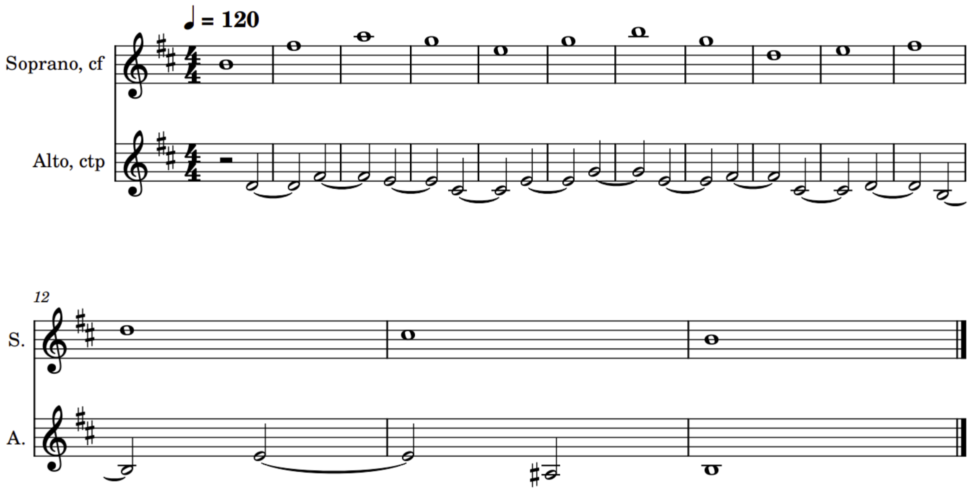


Figure A.16: Fourth Species in B minor audio: https://soundcloud.com/johan-gangsas-hole/fourth-species-b-minor

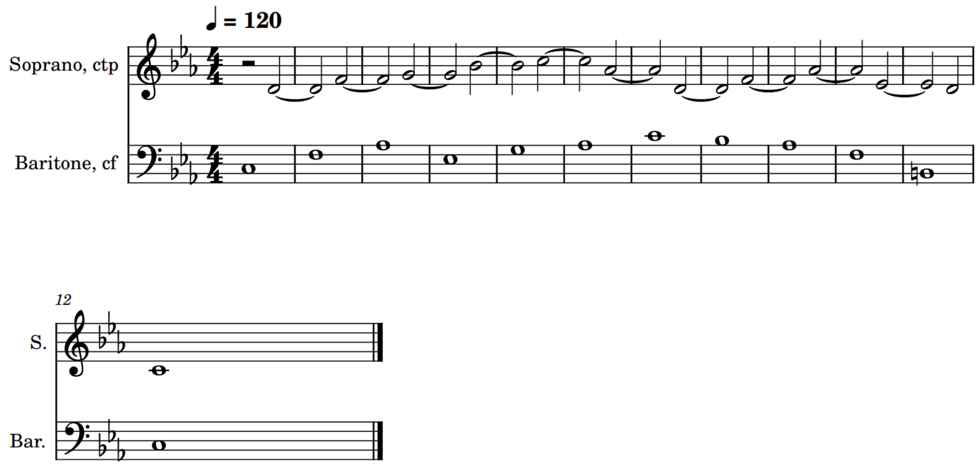


Figure A.17: Fourth Species in C minor audio: https://soundcloud.com/johan-gangsas-hole/fourth-species-c-minor

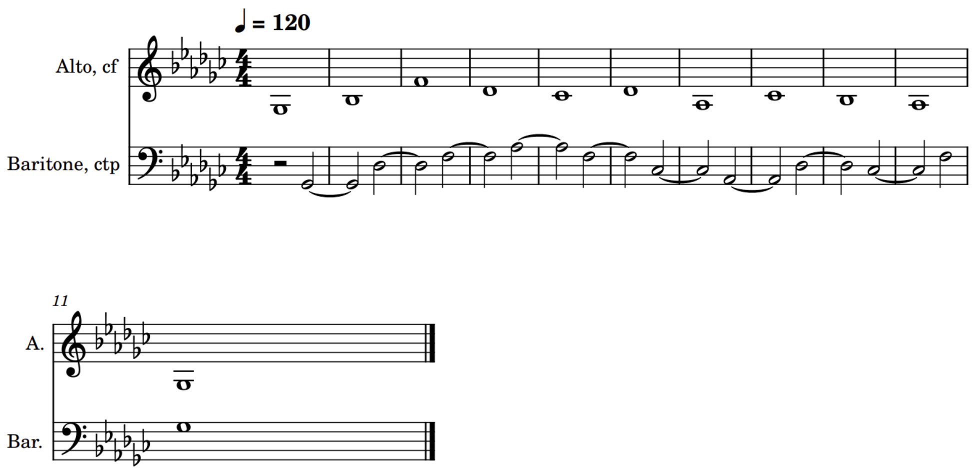


Figure A.18: Fourth Species in F# major audio: https://soundcloud.com/johan-gangsas-hole/fourth-species-f-major

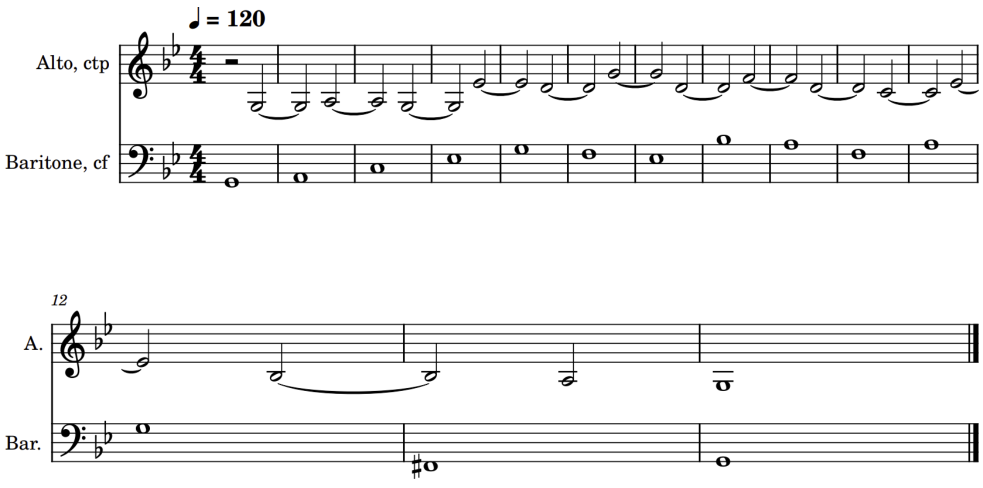


Figure A.19: Fourth Species in G minor audio: https://soundcloud.com/johan-gangsas-hole/fourth-species-g-minor

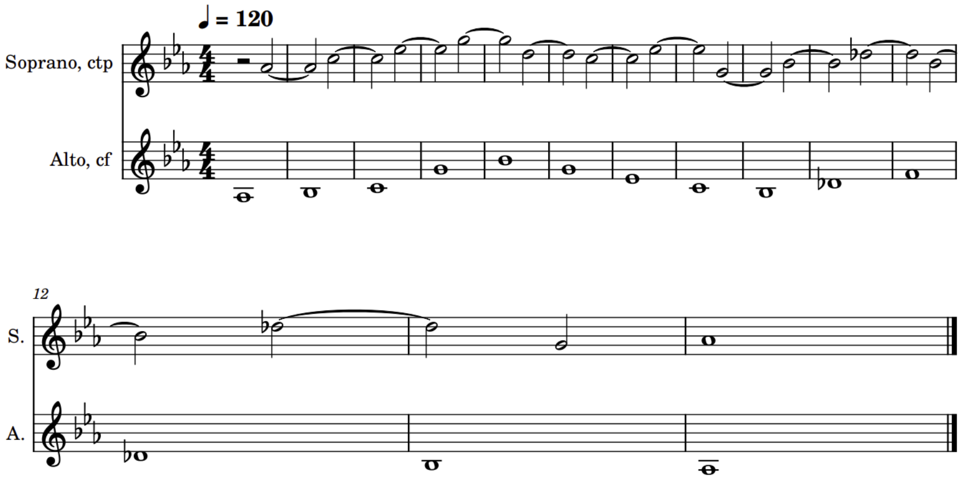


Figure A.20: Fourth Species in G# major audio: https://soundcloud.com/johan-gangsas-hole/fourth-species-g-major

A.5 Fifth Species

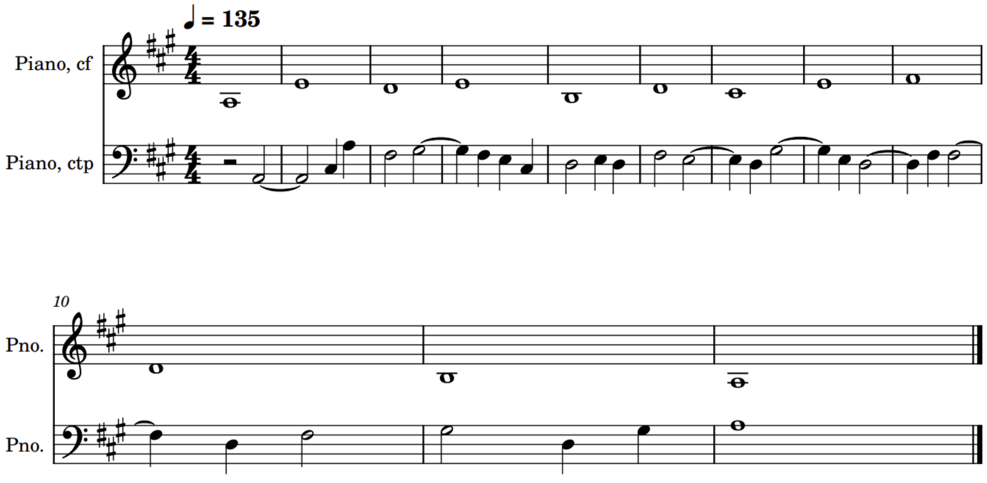


Figure A.21: Fifth Species in A major audio: https://soundcloud.com/johan-gangsas-hole/fifth-species-a-major

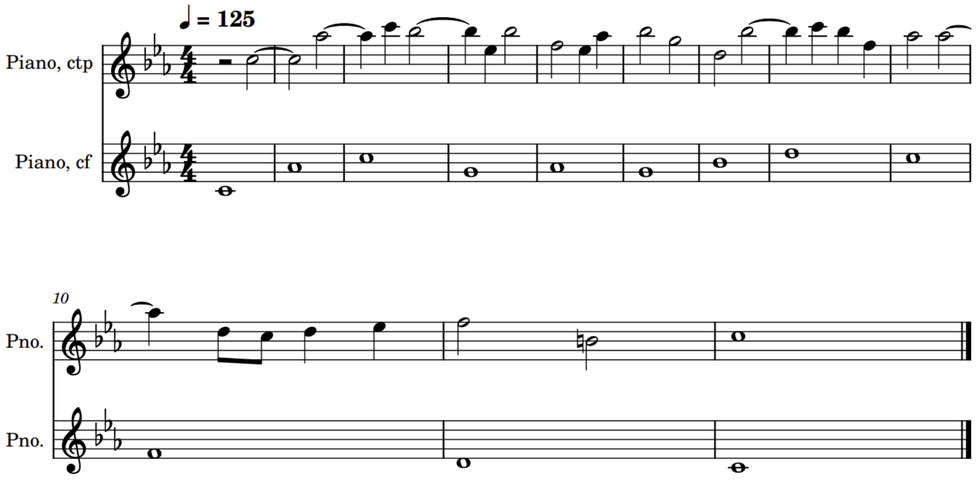


Figure A.22: Second Species in C minor audio: https://soundcloud.com/johan-gangsas-hole/fifth-species-c-minor

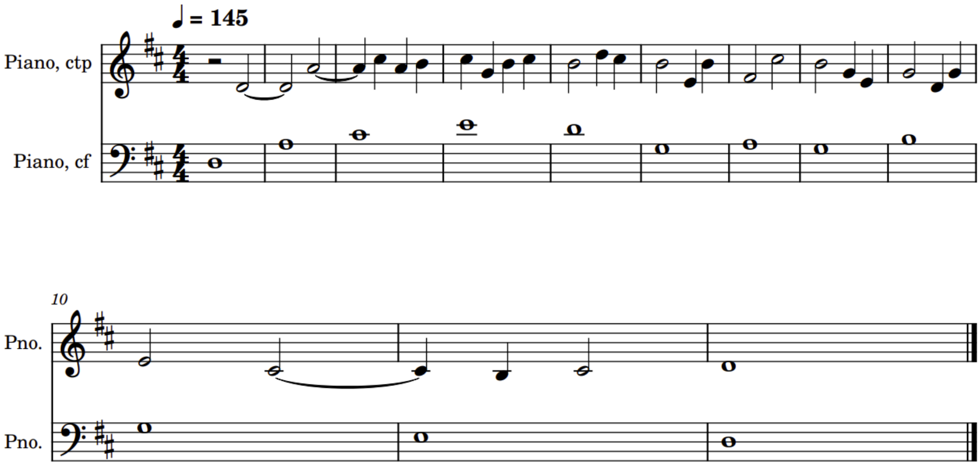


Figure A.23: Fifth Species in D major audio: https://soundcloud.com/johan-gangsas-hole/fifth-species-d-major

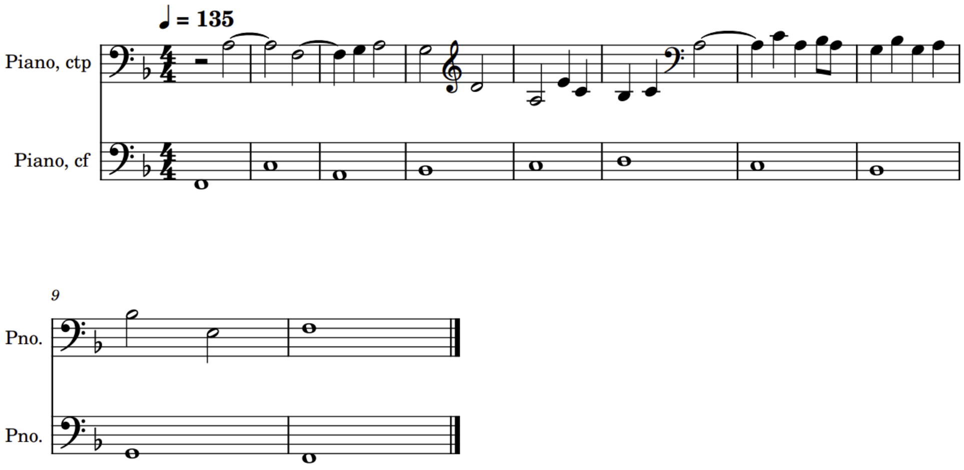


Figure A.24: Fifth Species in F major audio: https://soundcloud.com/johan-gangsas-hole/fifth-species-f-major

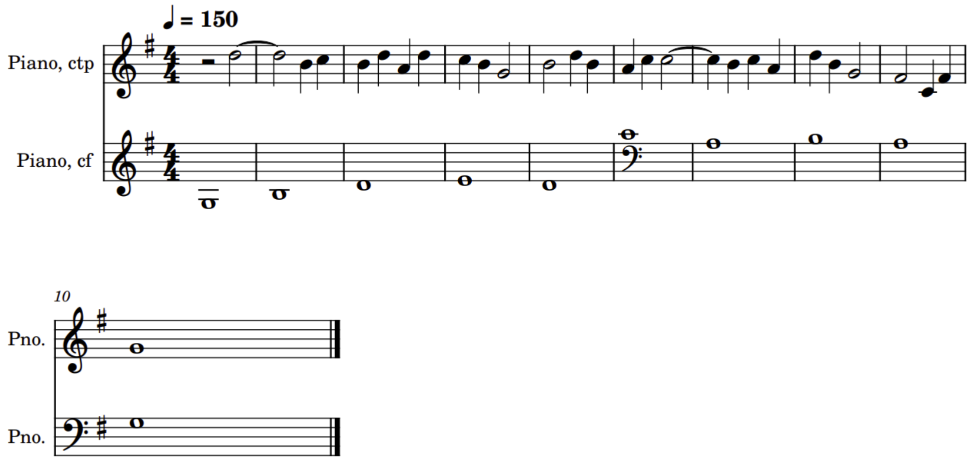


Figure A.25: Fifth Species in G major audio: https://soundcloud.com/johan-gangsas-hole/fifth-species-g-major

Bibliography

[1] Andres Acevedo. "Fugue Composition with Counterpoint Melody Generation Using Genetic Algorithms". In: vol. 3310. Feb. 2005, pp. 96–106. DOI: 10.1007/978-3-540-31807-1_7.

[2] Adam Alpern. *Techniques for algorithmic composition of music*. Hampshire College, 1995.

[3] Torsten Anders and Eduardo Miranda. "Constraint Programming Systems for Modeling Music Theories and Composition". In: *ACM Computing Surveys* 43 (Oct. 2011), 30:1–30:38. DOI: 10.1145/1978802.1978809.

[4] Alan Belkin. *Musical Composition*. Yale University Press, 2018. DOI: doi:10.12987/9780300235661. URL: https://doi.org/10.12987/9780300235661.

[5] Prafulla Dhariwal et al. *Jukebox: A Generative Model for Music*. 2020. arXiv: 2005.00341 [eess.AS].

[6] Morwaread Farbood and Bernd Schoner. "Analysis and Synthesis of Palestrina-Style Counterpoint Using Markov Chains". In: (Jan. 2001).

[7] J.J. Fux et al. *The Study of Counterpoint from Johann Joseph Fux's Gradus Ad Parnassum*. Norton library. W. W. Norton, 1965. ISBN: 9780393002775. URL: https://books.google.no/books?id=qQOnQZNvVSAC.

[8] Curtis Hawthorne et al. "Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset". In: 2019. URL: https://arxiv.org/pdf/1810.12247.

[9] Curtis Hawthorne et al. “Onsets and Frames: Dual-Objective Piano Transcription”. In: *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018, Paris, France, 2018*. 2018. URL: https://arxiv.org/abs/1710.11153.

[10] D. Herremans and K. Sörensen. “Composing fifth species counterpoint music with a variable neighborhood search algorithm”. In: *Expert Systems with Applications* 40.16 (2013), pp. 6427–6437. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2013.05.071. URL: https://www.sciencedirect.com/science/article/pii/S0957417413003692.

[11] Dorien Herremans and Kenneth Sörensen. “FuX, an Android app that generates counterpoint”. In: Apr. 2013, pp. 48 –55. DOI: 10.1109/CICAC.2013.6595220.

[12] Hermann Hild, Johannes Feulner and Wolfram Menzel. “HARMONET: A Neural Net for Harmonizing Chorales in the Style of J.S.Bach”. In: *Proceedings of the 4th International Conference on Neural Information Processing Systems*. NIPS’91. Denver, Colorado: Morgan Kaufmann Publishers Inc., 1991, 267–274. ISBN: 1558602224.

[13] Lejaren Arthur Hiller and Leonard M. Isaacson. *Experimental Music; Composition with an Electronic Computer*. USA: Greenwood Publishing Group Inc., 1979. ISBN: 0313221588.

[14] Dominik Hörnel. “MELONET I: Neural Nets for Inventing Baroque-Style Chorale Variations”. In: *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*. NIPS ’97. Denver, Colorado, USA: MIT Press, 1998, 887–893. ISBN: 0262100762.

[15] Cheng-Zhi Anna Huang et al. “Music Transformer: Generating Music with Long-Term Structure”. In: 2019. URL: https://arxiv.org/abs/1809.04281.

[16] David Miles Huber and Craig Anderton. *The MIDI Manual*. 2nd. USA: Butterworth-Heinemann, 1998. ISBN: 0240803302.

[17] Natasha Jaques et al. “Generating Music by Fine-Tuning Recurrent Neural Networks with Reinforcement Learning”. In: *Deep Reinforcement Learning Workshop, NIPS*. 2016.

[18] K. Jeppesen, G. Haydon and A. Mann. *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*. (The Prentice-Hall music series). Dover Publications, 1992. ISBN: 9780486270364. URL: https://books.google.no/books?id=OcSVGkug58gC.

[19] Alan P. Kefauver. *Fundamentals of Digital Audio*. USA: A-R Editions, Inc., 1998. ISBN: 0895794055.

[20] Maciej Komosinski and Piotr Szachewicz. “Automatic species counterpoint composition by means of the dominance relation”. In: *Journal of Mathematics and Music* 9.1 (2015), pp. 75–94. DOI: 10.1080/17459737.2014.935816. eprint: https://doi.org/10.1080/17459737.2014.935816. URL: https://doi.org/10.1080/17459737.2014.935816.

[21] Steven G. (Steven Geoffrey) Laitz. *The complete musician : an integrated approach to tonal theory, analysis, and listening*. eng. 3rd ed. New York: Oxford University Press, 2012. ISBN: 9780199742783.

[22] M. Laurson. *PATCHWORK: A Visual Programming Language and some Musical Applications. Ph.D. thesis.* Helsinki: Sibelius Academy, 1996.

[23] J. Lighthill. “Artificial Intelligence: A General Survey”. In: *Artificial Intelligence: a paper symposium* (1972). URL: http://www.chilton-computing.org.uk/inf/literature/reports/lighthill_report/p001.htm.

[24] Warren S. McCulloch and Walter Pitts. “A Logical Calculus of the Ideas Immanent in Nervous Activity”. In: *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, 15–27. ISBN: 0262010976.

[25] Michael Mozer. “Neural Network Music Composition by Prediction: Exploring the Benefits of Psychoacoustic Constraints and Multi-scale Processing”. In: *Connection Science - CONNECTION* 6 (Jan. 1994), pp. 247–280. DOI: 10.1080/09540099408915726.

[26] Gerhard Nierhaus. *Algorithmic Composition - Paradigms of Automated Music Generation*. Springer-Verlag Wien, 2009.

[27] Aaron van den Oord, Oriol Vinyals and koray kavukcuoglu. “Neural Discrete Representation Learning”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 6306–6315. URL: https://proceedings.neurips.cc/paper/2017/file/7a98af17e63a0ac09ce2e96d03992fbc-Paper.pdf.

[28] S. Oore et al. “This time with feeling: learning expressive musical performance”. In: *Neural Comput Applic* 32 (2020). DOI: https://doi.org/10.1007/s00521-018-3758-9.

[29] Colin Raffel and Daniel P.W. Ellis. “Intuitive Analysis, Creation and Manipulation of MIDI Data with pretty_midi”. In: 2014.

[30] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain.” In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 0033-295X. DOI: 10.1037/h0042519. URL: http://dx.doi.org/10.1037/h0042519.

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

[32] B. Schottstaedt. *Automatic Species Counterpoint*. Automatic Species Counterpoint nr. 19. CCRMA, Department of Music, Stanford University, 1984. URL: https://books.google.no/books?id=TJEXAQAAIAAJ.

[33] P. Todd and G. Loy. “A Connectionist Approach To Algorithmic Composition”. In: *Computer Music Journal* 13 (1989), pp. 173–194.

[34] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.