

École polytechnique de Louvain

FuxCP: Constraint Programming Formalisation of Three-Voice Counterpoint According to Fux

Author: **Anton LAMOTTE**

Supervisor: **Peter VAN ROY**

Readers: **Yves DEVILLE, Karim HADDAD, Damien SPROCKEELS**

Academic year 2023–2024

Master [120] in Computer Science and Engineering

Abstract

This thesis presents a formalisation of three-part musical counterpoint according to the classical theory of Fux. It is an extension of Thibault Wafflard's previous thesis, which formalised two-part counterpoint and implemented it in FuxCP, a software tool designed specifically for composers to help them compose counterpoint without the need for technical expertise. Counterpoint consists of several musical voices that are independent and distinct from each other, yet balanced and beautiful in sound. It consists of a fixed part, the *cantus firmus*, and one or more counterpoints derived from it. Three-part composition is much more expressive than two-part counterpoint because of the interaction between the two derived parts.

FuxCP is implemented in OpenMusic, a musical interface, and uses Gecode, a well-known constraint solver, to automatically generate counterpoints. The implementation is based on Johann Joseph Fux's *Gradus ad Parnassum*, a seminal treatise on counterpoint published in 1725, by translating its rules into formal logic and implementing them as constraints. In particular, the extension to three voices places special emphasis on the lowest voice, introducing innovative concepts and variables to address this key aspect. This work contributes to the research and understanding of automated contrapuntal composition by overcoming the challenge of generalising the interaction between voices. It also addresses preferences, which are treated as optional rules, introducing nuance into the generation of musical solutions and enhancing the overall aesthetic considerations in automated counterpoint composition. Importantly, this work builds seamlessly on T. Wafflard's previous efforts, ensuring full compatibility with his thesis.

Acknowledgements

Thanks to Peter Van Roy for his unwavering availability and presence, his always wise advice, his cheerfulness and his constant encouragement throughout this thesis.

Thanks to Damien Sprockeels for his astute guidance and his sharp insight in solving problems.

Thanks to Karim Haddad for his enthusiasm in the projet and his valuable musical advice.

Thanks in advance to Yves Deville for reading and reviewing this thesis.

Thanks to Vanessa Maons for all her dedication and help, and for being probably the most supportive secretary at the University.

Thanks to Thibault Wafflard for having paved the way and provided such good work to build on.

Thanks to all the people who took part in the development of this thesis whether by reviewing it or by offering me hot chocolate breaks.

Contents

1	Introduction and context of this work	1
1.1	A brief history of counterpoint: from Bach to algorithmic generation	1
1.1.1	Software tool for writing species counterpoint	2
1.2	Fux’s theory of counterpoint for two-, three- and four-part composition	3
1.2.1	Species counterpoint	5
1.3	Tools and implementation	5
1.3.1	Constraint Programming	6
1.3.2	OpenMusic	8
1.3.3	GiL and Gecode	8
1.3.4	Software integration	8
1.4	Standing on the shoulders of giants: underlying works and editions of <i>Gradus ad Parnassum</i> used	9
1.5	The contributions of this thesis	9
2	Definition of concepts and variables	11
2.1	Voices, parts and strata	11
2.2	Exploring the interaction of the parts with the lowest stratum	14
2.3	Definitions of the variables used in the formalisation	16
2.3.1	Variables and array notation	17
2.3.2	Overview of all the variables	17
2.3.3	In depth definition of the variables	20
3	Formal rules for three-part counterpoint	25
3.1	Implicit rules	26
3.1.1	Formalisation in English	26
3.1.2	Formalisation into constraints	26
3.2	First species	27
3.2.1	Formalisation into English	27
3.2.2	Formalisation into constraints	31
3.3	Second species	33
3.3.1	Formalisation in English	33
3.3.2	Formalisation into constraints	35
3.4	Third species	36
3.4.1	Formalisation in English	36
3.4.2	Formalisation into constraints	37
3.5	Fourth species	37
3.5.1	Formalisation in English	37
3.5.2	Formalisation into constraints	39
3.6	Fifth species	41
3.7	Writing a three-part composition using various species	42
4	Solution search for three-part counterpoint	43
4.1	Dealing with the higher computational complexity	43
4.1.1	Using Branch-And-Bound as a search algorithm	43
4.1.2	Heuristics	44
4.1.3	Time to find a solution	44

4.2	Designing the costs of the solver to be as faithful as possible to the preferences of Fux	45
4.2.1	Linear combination	46
4.2.2	Minimising the maxima	47
4.2.3	Lexicographic order	48
4.2.4	Comparison between the three types of costs.	50
4.3	Combining the three types of costs	50
4.3.1	Comparing the linear combination and the lexicographic order in practice	51
4.3.2	Mixing the technique of maximum minimisation with lexicographic order	56
4.4	Conclusion on the search methods	57
5	Musicality of the solutions	58
5.1	Combining first species with another species	58
5.2	Using preferences to improve musicality	60
5.3	Combining arbitrary species	61
6	Known issues and future improvements	62
6.1	Known issues about the current state of the work	62
6.2	Future improvements	63
	Conclusion	66
	Bibliography	67
A	Software Architecture	71
B	User Guide	73
B.1	Installing FuxCP	73
B.1.1	Prerequisites	73
B.1.2	Loading FuxCP in OpenMusic	73
B.2	Using FuxCP in OpenMusic	74
B.3	Interface Parameters Description	78
C	Complete set of rules for two and three part compositions	80
D	Code	93
D.1	FuxCP.lisp	93
D.2	package.lisp	93
D.3	interface.lisp	94
D.4	fuxcp-main.lisp	108
D.5	3v-ctp.lisp	123
D.6	cf.lisp	128
D.7	1sp-ctp.lisp	130
D.8	2sp-ctp.lisp	133
D.9	3sp-ctp.lisp	138
D.10	4sp-ctp.lisp	143
D.11	5sp-ctp.lisp	147
D.12	constraints.lisp	156

Chapter 1

Introduction and context of this work

This thesis is a formalisation for three-part counterpoint based on the theory of Johann Joseph Fux, as given in his classic treatise of 1725. It provides a mathematical formalisation of Fux's rules and a computer environment capable of implementing these logical rules in a concrete way to produce Fux-style counterpoint.

This thesis will therefore be divided into several parts: we will first immerse ourselves in *Gradus ad Parnassum*, Fux's central work, from which we will meticulously extract the rules laid down by its author. We will briefly discuss these rules to make them unambiguous, and then translate them into formal logic, so that each rule Fux had in mind when writing his work is mathematically recorded. On this basis, we will create a computer implementation using constraint programming. We will then look at how this implementation finds results, discussing the search algorithm and heuristics used. We then discuss the cost techniques used to obtain the best possible results. Finally, we will analyse the musical compositions produced by the tool created.

It is very important to know that this thesis is based on T. Wafflard's thesis "FuxCP: a constraint programming based tool formalizing Fux's musical theory of counterpoint" [1], written in 2023, and on article "A Constraint Formalization of Fux's Counterpoint" [2], by D. Sprockeels, P. Van Roy, T. Wafflard and K. Haddad. The present work takes up the concepts and definitions of T. Wafflard and could only be understood in its full depth by reading and fully understanding his works as well. This thesis also assumes a basic knowledge of music theory, which can be found in Chapter 1 of T. Wafflard's thesis.

1.1 A brief history of counterpoint: from Bach to algorithmic generation

Before delving into the formalities of our study, let's first examine Fux's theory of counterpoint, which forms the basis of the formalisation undertaken in this work. Counterpoint is a compositional technique in which there are several musical lines (or voices) that are independent and distinct from each other, but that are balanced and sound beautiful [3]. No voice is dominant over the others, and all are main voices, although some may take a small precedence during part of the composition [4].

Counterpoint has been central to the work of many famous composers from different artistic movements, such as Bach in the Baroque era, Mozart in the Classical era and Beethoven in the Romantic era [5]. It is still present in some modern music [6], and has aroused interest over the centuries with the development of key texts on the subject, such as Schenker's Counterpoint [7] or Jeppesen's Analysis [8]. And while Bach mastered counterpoint to an unprecedented level for his time [9], the central and foundational work in the teaching of counterpoint belongs to another great Baroque composer: the Austrian Johann Joseph Fux and his treatise *Gradus ad Parnassum*. In it, this composer gives a detailed analysis of the writing of two-, three- and four-part counterpoint, all narrated as a conversation between a master and his pupil. *Gradus*

ad Parnassum is one of the works that deal with species counterpoint¹, a way of conceiving counterpoint in five different types that could then be combined. It is on this work that this dissertation is based.

1.1.1 Software tool for writing species counterpoint

We have been discussing the longstanding tradition of counterpoint, a musical technique shaped by countless generations of composers. As technology advanced, the idea of automating counterpoint composition emerged. This section takes a look at some of the work that has been done in the field of counterpoint generation.

An early attempt, by Schottstaedt in 1984 [10], involved an expert system that is also based on Fux's rules. His approach used over 300 if-else clauses. However his method had obvious limitations compared to what modern constraints are capable of, since if-else clauses are unidirectional, whereas constraints are bidirectional, which ensures better propagation of the constraints. More importantly constraint modelled problems don't just lead to single solutions, they represent sets of potential solutions. This flexibility is a significant improvement over the directional nature of if-else clauses. Furthermore, constraint systems offer an advantage in specifying intricate search heuristics. This adaptability and efficiency highlights the stark contrast between the outdated approach of if-else clauses and the modern capabilities of bidirectional constraint systems in the realm of counterpoint composition.

In 1997, a genetic programming and symbiosis approach to automatic counterpoint generation was developed by J. Polito et al. This team from Michigan used a genetic approach to optimise counterpoints of the 5th species and make them more attractive [11]. A similar approach was used in 2004 to generate fugues (another musical technique that relies a lot on counterpoint), also using genetic algorithms [12]. The results are quite promising, and generate more than interesting results, but the end result is still far from being able to provide a complete counterpoint composition.

Many years later, in 2010, G. Aguilera et al., from the University of Malaga, developed an automated method for the generation of first-species counterpoint using probabilistic logic [13]. Their approach was specifically tailored to compositions in C major, providing a generated counterpoint in response to a given *cantus firmus*. However, this application evaluates only the harmonic attributes of the composition, ignoring the melodic aspect of the counterpoint.

Two years later, D. Herremans and K. Sørensen developed a way to generate high-quality first-species counterpoint using a variable neighbourhood search algorithm [14]. Their research was limited to first-species counterpoint, but they addressed issues such as preferences (finding the best counterpoint) and user-friendly interface. Once again, their results are more than impressive, but their research is limited to the first two-voice species.

Finally, a research was carried out in 2015 on Fux's counterpoint [15], with the aim of generating the first species counterpoint using dominance relations, has yielded fairly good results. The search demonstrates the use of this paradigm and its applicability, and is a good starting point for composing counterpoints of other species based on the same concept.

¹There are many other types of counterpoint, such as free counterpoint, dissonant counterpoint, linear counterpoint, ...

If we now focus on applications that have gone as far as the user interface and are now ready to use, we should mention two namesakes, both called ‘Counterpointer’, which have the merit of offering a functional tool for composing counterpoint.

The first Counterpointer [16] is a tool which anyone can use to check the validity (or not) of their counterpoint. Its last release was in 2019 as a desktop application, and it works like this: an apprentice composer tries to write a counterpoint, and then submits it to the tool. The tool then decides whether the counterpoint is valid according to the traditional rules of counterpoint². It also provides feedback to help the student composer improve their future counterpoint writing. The tool is not able to write counterpoints automatically, nor is it explicit about how it works, as it is completely closed source and has no accessible report. It is therefore impossible to know the paradigm it uses or the exact rules it follows.

Another attempt at automatic counterpoint writing is the Counterpointer project in 2021, created by a team of students at Brown University as part of a software engineering course [17]. The project is less accomplished than the aforementioned application, but it has the merit of being able to generate two-voice counterpoints of the first, second and third species. It is an entirely free and open source project. While the results are encouraging, the project has been discontinued as it was a course project and their method of finding a counterpoint seems much less efficient than the efficiency that a constraint solver can achieve.

This brief overview leads us to conclude that there is no satisfactory tool for composing counterpoint in a user-friendly way, with good quality, quickly and with several voices. It is to fill this gap that this research has been carried out. This was the aim of T. Wafflard’s thesis and it is therefore natural that this thesis should follow in his footsteps.

1.2 Fux’s theory of counterpoint for two-, three- and four-part composition

As with so many other authors who have attempted to automate the writing of counterpoint, it is only natural that this work should be based on Fux’s theory. He was one of the first to theorise counterpoint in such a comprehensive way, and although his theory has been extended many times since, it remains a very good foundation.

For Fux, as for many other authors, species counterpoint is governed by many different rules, and it is these rules that interest us in the present work. The rules are based on old concepts that can be traced back to older styles and have been studied and discussed by generations of authors [18]. Those concepts include, for example, the notion of opposite motion, or the notion of consonance (which in turn can be either perfect or imperfect). These concepts and their application to counterpoint are particularly interesting because they allow us to consider the composition of counterpoint both in a ‘vertical’ way, in which we consider the harmony of the notes played together, and in a ‘horizontal’ way, in which we consider the melodic development of each of the parts individually, which provides the independence of the counterpoints from each other and their melodic beauty.

All the rules defined by Fux can be divided into three categories: melodic rules, harmonic rules and motion rules. We will examine them here to get an initial sense of what they mean, in order to be able to formalise them afterwards.

²Not only Fux’s rules, but also those of other authors.

Melodic Rules

Fux explains that there are rules that apply within parts (the horizontal rules). These rules are concerned with the intervals between one note and the next ones: we find, for example, that a melody is more "beautiful"³ when the intervals between its successive notes are small, when there is no chromatic succession between the notes, when the notes are varied, and so on. These 'horizontal' rules are called 'melodic' rules because they are concerned only with the melody of a given voice, and therefore apply within that voice.

Harmonic Rules

If there is a horizontal perspective to counterpoint, there is also, of course, a vertical perspective. This perspective is expressed in a harmonic relationship between the different voices. At each point in the composition, a series of rules apply that concern harmony alone, and puts some constraints on the harmonic intervals that the voices can have between each other. Here are some harmonic rules from Fux, given as an example: the harmonic interval between any voice and the lowest one must be either a third, a fifth, a sixth or an octave; thirds and sixths are preferred to fifths, which in turn are preferred to octaves; and the voices can't use the same note at the same time. These rules apply between the voices.

Motion Rules

Finally, there is a third type of rule: the motion rules. These rules are a hybrid of the two discussed above in that they consider not only vertical interaction, i.e. harmony, but also horizontal interaction, i.e. melody. They can therefore be seen as 'diagonal' rules that relate the unique melody of each counterpoint to its respective harmonies. There exists three types of motions: the direct motion, when two voices move together, the oblique motion, when one voice stays idle and another one moves, and the contrary motion, when both voices move in opposite directions. When coupled to harmonic concepts, we get rules such as: contrary motions are preferred to direct motions; there should be no successive fifths or successive octaves between the voices; and a direct motion should not lead to a perfect consonance. As you can see, these rules take into account not just two voices at a given point, but over several measures. They ensure a harmonious interaction between the voices.

Preferences

This last point is one of the most important in Fux's theory: the preferences. Preferences are hints that Fux gives in *Gradus ad Parnassum* in order to write even better counterpoints. As their name suggests, preferences are optional and not compulsory to follow, as other strict rules would be. However, preferences are crucial to Fux's work because they allow us to distinguish between two valid solutions (those that obey all the strict rules) and decide which is the best, thus allowing the composer to control how the strict theory is applied.

Fux is never clear about whether a rule⁴ is a preference or a strict rule — and that's normal, what he conveys is mostly intuition, and human beings are quite capable of

³Throughout this work we will speak of the "beauty of music". This beauty is highly subjective, and therefore reference will be made to the Fuxian concept of music to define whether a melody is beautiful or not. In other words, music is considered beautiful if it conforms to Fux's rules, and vice versa.

⁴We use the generic term "rule" to refer to both mandatory rules and preferences.

understanding whether a rule is a preference or an obligation; Fux probably didn't expect someone to try to formalise his work three centuries later.

These preferences should be respected whenever possible. However, if a preference cannot be respected, the solution is still valid. Here is a good example: Fux indicates that we prefer to have as many different notes as possible in the composition. This is not a strict rule, but a preference. The more variety there is in the composition, the more beautiful and the more preferable it will be.

1.2.1 Species counterpoint

When discussing species counterpoint, we refer to five categories of counterpoint, each of which represents a distinct concept with its own characteristics. A detailed explanation of these species is given below. First, let's concentrate on how writing counterpoint works. In counterpoint composition, the starting point is a fixed melodic line known as the *cantus firmus*, which is a fundamental melody composed entirely of whole notes. This melodic line serves as the basis for composing the entire piece of counterpoint. It's important to emphasise that once the composition is complete, the *cantus firmus* is neither more nor less important than the other voices. It has the same degree of melodic independence as the other voices and acts as a starting point rather than a more important voice in the compositional process.

Let's take a look at the five species:

1. **First species:** Note against note – the first species counterpoint is composed entirely of whole notes, and the composition is a sequence of harmonies sounding on the first beat between the counterpoint and the other voices.
2. **Second species:** Half notes against whole notes – the second species counterpoint is composed entirely of half notes, which introduce dissonant harmonies.
3. **Third species:** Quarters against whole notes – the third species counterpoint is made up entirely of quarter notes, which allow more different movements and more freedom in the composition.
4. **Fourth species:** The ligature – the fourth species counterpoint is delayed by two beats, creating syncopation. The notes are all half notes, tied two by two, which creates the effect of having only delayed whole notes.
5. **Fifth species:** Florid counterpoint – the fifth species counterpoint is a mixture of all the other species and is the richest form of counterpoint. It allows great freedom of composition while respecting the rules of the other types.

These different species can be combined to form a composition with a *cantus firmus*, a counterpoint of one species and another counterpoint of another species. Nevertheless, Fux seems to prefer writing compositions that are made of a *cantus firmus*, a counterpoint of the first species and another counterpoint of another species, probably for pedagogical reasons.

1.3 Tools and implementation

Now that we have a better understanding of the species counterpoint, let's focus on how Fux's rules are implemented in practice. This subsection discusses the implementation of FuxCP (the tool to automatically generate counterpoints). To do so, we briefly explain how constraint programming works and the tools used by FuxCP (OpenMusic, Gecode and GiL).

1.3.1 Constraint Programming

Constraint Programming (CP) is a programming paradigm used to solve large combinatorial problems, such as planning and scheduling problems. It works by defining constraints between variables that limit the values these variables can potentially take [19]. In doing so, the domains of these variables are reduced. R. Barták explains in a very clear way what a constraint really is, as he describes in his "Guide to Constraint Programming" [20]:

A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain. A constraint thus restricts the possible values that variables can take, it represents some partial information about the variables of interest. For instance, "the circle is inside the square" relates two objects without precisely specifying their positions, i.e., their coordinates. Now, one may move the square or the circle and he or she is still able to maintain the relation between these two objects. Also, one may want to add other object, say triangle, and introduce another constraint, say "square is to the left of the triangle". From the user (human) point of view, everything remains absolutely transparent.

The question now is: what is the connection between this problem-solving method and counterpoint composition? Interestingly, music happens to be an eminently suitable application for Constraint Programming. All aspects of a composition can be represented by variables, rules can be established between these variables, and the solver is responsible for finding a valid solution. In fact, in music, it is never just one factor that determines whether the music is beautiful or not, but an interaction of many factors. As humans, it is sometimes difficult to find a good solution (i.e. to compose music that sounds good) because the range of possibilities and the interactions between factors are so numerous. However, exploring a search space in which numerous constraints are defined is something that a constraint solver does very well. The most arduous task then becomes identifying the rules that make music beautiful, and this is the task that many musicologists and composers, like Fux, have set themselves. Once these rules have been defined, it is "simply" a matter of formalising them and passing them to the constraint solver so that it can compose a melody that respects these rules.

What's more, by defining a rigorous way of distinguishing a good composition from a bad one, the solver can even find increasingly beautiful solutions.

To make sure that Constraint Programming is a well understood concept, we here review its main concepts:

Constraint propagation Each time a constraint is defined, the domain of the affected variables is reduced according to the possibilities left by the constraint. This is called constraint propagation. Let's imagine a variable A whose domain is $\{0, 1, 2, 3\}$ and a variable B whose domain is $\{0, 1, 2\}$. When the constraint $A < B$ is applied, the domain of A is reduced to $\{0, 1\}$, because the new constraint makes it impossible for A to have a value of 2 or 3 without violating the constraint.

A constraint solver is an implementation that systematically searches the search space for a solution. A given problem may have many solutions, only one, or none at all.

A solution is found when all variables are fixed, i.e. their domain is reduced to a single value. We know that a problem has no solution when the domain of any vari-

able becomes empty (because this means that no value can be found for that variable without violating a constraint).

Branching Obviously, declaring constraints is not enough to magically find a solution. In the example we proposed earlier (with A and B), the single constraint placed on the search space doesn't allow us to determine the values of A or B , and their domains remain composed of more than one value. To actually find a solution, the constraint performs a branching. That is, it studies two antagonistic possibilities and splits the search space into two subproblems accordingly. More specifically, it chooses a variable and studies the case where that variable is equal to a certain value, and the case where it is not equal to that value. For example, the solver might study the case where $B = 0$ (the first branch) and the case where $B \neq 0$ (the second branch). If the solver finds an inconsistency in either case, it knows that the entire branch can be discarded (as it does not lead to a valid solution). Immediately after branching, the solver again performs constraint propagation, since constraint propagation occurs whenever any domain is modified, and consists of adjusting all domains to which the modified domain is linked by a constraint. In our case, after setting B to 0, the solver propagates all constraints linked to B , i.e. the only constraint in our problem (i.e. $A < B$). This affects the domain of A , reducing it to an empty domain (because no value in the domain of A is less than 0). The solver, noticing that one domain is empty, concludes that this branch contains no solution and therefore knows that the only possible branch is the other one, i.e. the one in which we assumed $B \neq 0$. We can therefore safely remove 0 from the domain of B , since this value is not contained in any solution. Repeating the branching process each time it is necessary produces three solutions: $A = 0$ and $B = 1$, $A = 0$ and $B = 2$, and finally $A = 1$ and $B = 2$.

Heuristics As with all problems involving searching a space in quest of a solution, it is very useful to have heuristics allowing for an efficient search. A heuristic is a rule or strategy used to make informed decisions about variable assignments and value choices during the solution search. These rules are designed to exploit the characteristics and structures of the problem to improve the chances of finding solutions more quickly. A common heuristic is variable ordering, where the algorithm selects variables to assign values to based on factors such as the size of the domain or the number of associated constraints. Another important heuristic is value ordering, which determines the order in which values are tested for a given variable assignment. By incorporating heuristics, constraint solvers can prioritise the most promising branches of the search tree, effectively reducing the search space and speeding up the identification of feasible solutions. While heuristics speed up the solving process, it's important to strike a balance between exploration and exploitation, as overly aggressive heuristics risk missing potentially valuable solution paths.

To return to our previous example, a value-ordering heuristic might be "branch first on low values of A and high values of B , since we know that we are looking for a solution where A is less than B , and we can reason that there are more chances of satisfying this constraint when B is large and A is small.

Optimal solutions Constraint programming can also be used to find optimal solutions, i.e. solutions that have minimum cost. The cost is defined as a function that assigns a value to each solution.

In our simple example, we could define the cost as the sum of A and B and want to minimise the cost. This means that we are looking for a valid solution where A and B are as small as possible. In this case, only the first of the three solutions mentioned

above is chosen, i.e. $A = 0$ and $B = 1$, as it is the best possible solution (with a cost of 1).

Branch and Bound Branch and Bound is a systematic algorithm used in Constraint Programming to efficiently explore the solution space and find an optimal solution with respect to a given cost or objective function. This technique extends the basic Constraint Programming approach by introducing a mechanism to prune unpromising branches of the search tree. By making this, it reduces the computational effort required to find the optimal solution. For each branch, the algorithm evaluates its feasibility and its potential to lead to a better solution. If a branch is deemed infeasible or cannot possibly improve on the current best-known solution, it is pruned from further consideration, without having to evaluate the full depth of it. This process continues until all branches have been explored, or until the algorithm converges on the optimal solution. Branch and Bound is particularly valuable for large combinatorial problems because it efficiently narrows the search space, allowing the solver to focus on promising regions and accelerating the discovery of optimal solutions in constrained programming scenarios [21].

1.3.2 OpenMusic

OpenMusic is a powerful and innovative visual programming environment, written in CommonLisp, designed specifically for composers, researchers and musicians involved in computer-aided composition [22]. Developed by the Institute for Research and Coordination in Acoustics/Music (IRCAM) in Paris, OpenMusic provides a graphical interface that allows users to create and manipulate musical structures using a variety of predefined modules. This visual programming environment facilitates the representation of complex musical ideas, algorithms and data flows through a user-friendly interface, making it accessible to both novice and experienced composers.

FuxCP is a library for OpenMusic, which means that OpenMusic is the interface for using FuxCP. Any user wishing to use FuxCP writes their *cantus firmus* (FuxCP's input) into OpenMusic and then launches the solution search from within OpenMusic. Specifically, FuxCP retrieves the *cantus firmus* from OpenMusic and then defines the constraint problem. When a solution (the counterpoints) is found, it is passed to OpenMusic and the user gets it as an OpenMusic object.

1.3.3 GiL and Gecode

GiL is an interface between OpenMusic and Gecode, which in turn is an open source toolkit for developing constraint-based systems [23], which provides a high-level C++ library for efficiently modelling and solving constraint problems. GiL allows to use the Gecode tool within a Lisp environment, thanks to the Common Foreign Function Interface (CFFI) [24].

1.3.4 Software integration

To put it all together: FuxCP gets its input (the *cantus firmus*) from the user interface, which is OpenMusic. It then uses GiL to define a constraint programming problem in Gecode.

As for the way the problem is defined, here is a little clarification: first, when the *cantus firmus* is received, a whole series of constants and variables are defined: for example, the length of the *cantus firmus*, the arrays representing the pitches of the counterpoints, ... All these variables are then constrained according to the constraints

defined in the formalisation of Fux’s rules (discussed in Chapter 3). The constraints are set sequentially: first the constraints on the *cantus firmus*, then the constraints on the first counterpoint, and finally the constraints on the second counterpoint. A diagram of the code architecture and the integration of FuxCP with other tools can be found in Figure A.2.

1.4 Standing on the shoulders of giants: underlying works and editions of *Gradus ad Parnassum* used

As has been said, this work is the continuation of T. Wafflard’s work. However, it also relies heavily on the work of:

- Lapière [25], who presented an interface for using Gecode functions in Lisp called "GiL". This interface was then tested with some rhythm-oriented constraints.
- Sprockels [26], who explored the use of constraint programming in OpenMusic using GiL. The tool that was produced in this thesis is capable of producing songs with basic harmonic and melodic constraints.
- Chardon, Diels, and Gobbi [27], who created a tool capable of combining the strengths of the first two implementations while continuing to develop support for GiL.

As with T. Wafflard, the musical reference work chosen is Fux’s *Gradus ad Parnassum*, because it is a pillar of counterpoint theory and because it is fairly easy to extract rules from it, although Fux is sometimes very vague about his intentions. As with any book published several centuries ago (1725 in the case of *Gradus ad Parnassum*), there are many versions and translations, which is something good, since Fux can sometimes be really unclear about what he means. Having many versions (some annotated, some not) from many people who also had to interpret Fux in order to translate him is a great treasure, as it helps to clarify Fux’s meanings. This work is therefore based on several different editions and translations of the book, although it is mainly based on Alfred Mann’s English translation [28]. French (both Chevalier’s [29] and Denis’s [30]), German [31] and Latin [32] translations are used when it is necessary to remove an ambiguity or clarify an unclear rule. These translations have been chosen because French is the lingua franca of the team; German is the language of Fux and the environment in which he evolved; and Latin is the original version, so we can hope that it is the most faithful to what he wanted to convey.

1.5 The contributions of this thesis

This work generalises T. Wafflard’s formalisation of two-part counterpoint composition to three-part composition. The following is a more detailed summary of the contributions of this thesis.

- **Concepts and variables to three-part counterpoint:** Three-part composition is much more than a (two+one) part composition. This requires a whole series of concepts to be defined or adapted. While some concepts are adapted from T. Wafflard’s work, others are completely new, such as the strata. These novel concepts are essential for the formalisation of Fux’s three-part composition rules, and are discussed in Chapter 2.

- **Mathematical formalisation of three-part counterpoint:** Fux's explanations are translated into unambiguous English and then translated again into logical notation. This formalisation builds on the previous formalisation for two voices, and sometimes (rarely) has to modify it. This formalisation can be found in Chapter 3.
- **Implementation of a working constraint solver for a three-voice composition:** The formalised logical rules are then implemented as constraints. These constraints are used by a solver to find solutions that include two counterpoints. The whole code of this implementation can be found in Appendix D, and its architecture in the Appendix A.
- **Researching the best way to express Fux's preferences:** Three-part composition introduces so many possibilities for result composition that it is important to discuss the way we think about preferences. The preferences are understood by the solver as costs (where a preferred solution in Fux's sense has a lower cost to the solver). Therefore, some techniques for managing these preferences are discussed to find out the best way to implement them. This is very important as it allows the solver to produce solutions with high musicality. These techniques are discussed in Chapter 4.
- **Musical analysis of the solutions generated by the solver:** Finding the best solution also means being able to assess the quality of current solutions. For further details, please refer to Chapter 5.
- **User interface for three-point counterpoint that allows a composer to specify how preferences are used in the solver.:** All the new capabilities of the solver and the costing techniques must also be accessible to the user: it is possible for a user to freely combine any number of species to form a three-part composition, and to set a cost importance order to indicate their preferences to the solver (in addition to the already existing ability to set personalised costs). A guide to the installation and use of the interface (and of the whole tool) can be found in Appendix B.

Chapter 2

Definition of concepts and variables

The purpose of this chapter is to provide all the definitions necessary to formalise Fux's theory. It defines various concepts and variables that will be used in the next chapter, which is the actual formalisation.

The first section of this chapter deals with a very important concept in the formalisation of three-part counterpoint, namely parts and strata. It also discusses between which voices the constraints should apply. The second section contains all the definitions of the variables used in the formalisation of Fux's rules.

2.1 Voices, parts and strata

The most important definitions we introduce in this section are the concepts of *parts* and *strata*. The need for these definitions arises from the increasing complexity of the rules of counterpoint when it is generalised to three voices. When getting from two-part to three-part composition, the rules are no longer concerned solely with the counterpoints and the *cantus firmus*, but also with new concepts, such as that referred to by Fux as 'the lowest voice'. The term 'voice' is too generic: Fux uses it to describe notions as different as 'counterpoint', '*cantus firmus*', voice range and the so called 'lowest voice'. We therefore need to create a more specific vocabulary to talk about these concepts. With this in mind, let's explain what 'parts' and 'strata' are, and how they relate to the concept of 'voice'.

Voices

Voices are a *general* concept, whereas parts and strata are more precise and *specific* concepts. The concept of 'voice' includes both 'parts' and 'strata'. In other words, each of these two concepts is a type of voice. In this thesis, when a voice is mentioned, it can refer to either a part or a stratum, since both parts and strata are voices. To use an object-oriented metaphor, we could say that the 'parts' and 'strata' classes inherit from the 'voice' class.

Since there are as many parts and strata as there are voices, in a composition with n voices there will also be n parts and n strata.

Parts

A part corresponds to what a particular person sings or what a particular instrument plays. It corresponds to a musical staff (each staff corresponds to a part). The term 'part' is the same as that used by Fux in *Gradus ad Parnassum*. The parts in a three-part composition are: the *cantus firmus*, the first counterpoint and the second counterpoint. Fux distinguishes them by calling them by their range, i.e. "bass", "tenor", "alto" or "soprano" (obviously you cannot have all four in a three-part composition).

Strata

A stratum delineates discrete layers or levels of pitches at any given moment in the composition. They denote a vertical alignment of simultaneous notes and organizes them into distinct strata. By definition, the lowest stratum encompasses the lowest sounding notes, the highest stratum comprises the highest sounding notes, and the intermediary stratum represents pitch levels in between. This concept is very helpful in identifying and categorising the vertical placement of pitches, creating distinct categories of sound within the overall texture of the counterpoint composition. It provides a way of analysing and understanding the distribution of pitches across different parts, allowing more complex rules to be established. For example, it would now be possible to establish a rule between the notes of the *cantus firmus* and the highest sounding notes (no matter which part they come from). The full potential of strata lies in harmonic rules, but as we shall see, some melodic rules are also related to it. When Fux speaks about the lowest stratum, he often uses the word 'bass'. It was deliberately chosen to use a different term than him because 'bass' is also the name of a voice range (like soprano and alto, for example), and there is already enough complexity in all the terminology to add even further ambiguity.

Fux approaches the formal definition of lowest stratum without ever stating it clearly, mentioning for example that the lowest voice can change (sometimes the bass is the lowest voice, sometimes the tenor, ...), and that at any given moment the lowest voice should be considered.

Since a picture is worth a thousand words, Figure 2.1 illustrates the difference between parts (the blueish lines) and strata (the red and orange lines). The lowest stratum is shown in its own colour (red) because it is the most meaningful stratum, and it is particularly important in the formalisation.

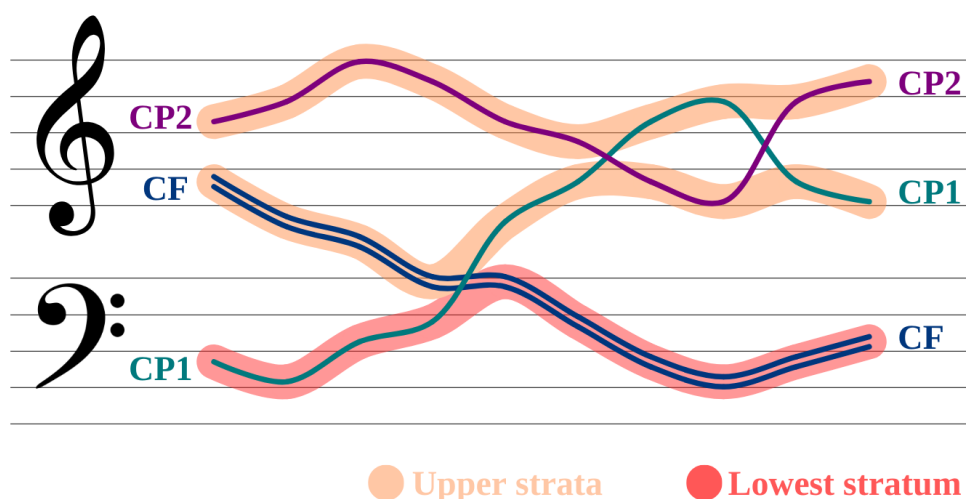


Figure 2.1: Parts and strata in a three voice composition

Important note concerning the strata Strata are an abstract concept, useful only in the mathematical formalisation of Fux's rules. They are necessary because we need a

structure that is able to comprehend the lowest sounding note for each measure. The strata concept is obviously not needed to write counterpoint as a human being, and the aim behind its definition is not to create a new concept for music theory, but to enable us to use a tool in our constraint programming way of conceiving counterpoint composition.

The term stratum was chosen in this context for its visual impact. In geology, a stratum "is a rock layer with a lithology (texture, colour, grain size, composition, fossils, etc.) different from the adjacent ones" [33], see Figure 2.2.



Figure 2.2: Geological strata, for the illustration

The mathematical representations for the notes of the strata are provided here:

- The lowest stratum (written $N(a)$, see Section 2.3 for the notations):

$$\forall i \in [0, 3] \quad \forall j \in [0, m-1] : N(a)[i, j] = \min(N(cf)[i, j], N(cp_1)[i, j], N(cp_2)[i, j]) \quad (2.1)$$

- The first upper stratum, or medium stratum (written $N(b)$, see Section 2.3 for the notations):

$$\forall i \in [0, 3] \quad \forall j \in [0, m-1] : N(b)[i, j] = \text{med}^1(N(cf)[i, j], N(cp_1)[i, j], N(cp_2)[i, j]) \quad (2.2)$$

- The second upper stratum, or uppermost stratum (written $N(c)$, see Section 2.3 for the notations):

$$\forall i \in [0, 3] \quad \forall j \in [0, m-1] : N(c)[i, j] = \max(N(cf)[i, j], N(cp_1)[i, j], N(cp_2)[i, j]) \quad (2.3)$$

One part per stratum and one stratum per part

For each measure, there is a bijection between the parts and the strata. This means that, for any given measure, each stratum uniquely corresponds to a single part, and vice versa. Put differently, if two parts within a measure share the same pitch, they do not constitute the same stratum. Instead, one part corresponds to one stratum, and the other one to a separate stratum.

To illustrate this, consider a scenario in a two-voice composition (see Figure 2.3), where part 'cf' and part 'cp1' in measure X both have a pitch value of 67 (representing a G). Despite having identical pitches at the same moment, one part is categorised as the lowest stratum, while the other is designated as the uppermost stratum. This distinction becomes crucial for subsequent analysis, especially when calculating aspects like motions. To decide which part gets to be the lowest stratum in such situations, an arbitrary hierarchical rule is implemented. If the ambivalence is between the *cantus firmus* and another part, the *cantus firmus* is always prioritised and assigned the role of the lowest stratum, over any other part. In the case of an ambivalence between the first counterpoint and the second counterpoint, the first counterpoint is given the status of the lowest stratum.

¹Where $\text{med}(X)$ means the median value of X .

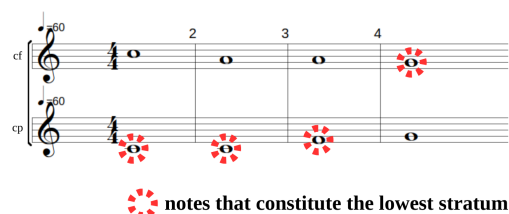


Figure 2.3: Establishing which part corresponds to the lowest stratum

Note concerning the intersection of the voices

Some authors consider that the voices should not cross for too long, whereas Fux seems to have no opinion on the subject. Let's be clearer on that point: the stratum concept makes it possible to create compositions in which the bass doesn't always play the lowest notes, and this is precisely its purpose. In order for a voice other than the bass (the tenor, for example) to play the lowest note, the melodies of the tenor and the bass must cross. Fux is strangely silent on this subject: he says that crossings are perfectly permissible, but he doesn't elaborate. Other authors are clearer, and the rules of species counterpoint generally state that crossing is allowed, but that the voices must not remain inverted for too long [34, p.28]. This rule was not taken into account in the formalisation because it is not present in *Gradus ad Parnassum*, but it is a good notion to keep in mind, as it could be a way of improving the FuxCP tool, with the aim of making it compatible with other styles of counterpoint.

2.2 Exploring the interaction of the parts with the lowest stratum

In three-part writing, most constraints apply **between the different parts and the lowest stratum**. This makes the lowest stratum the most important voice of all.

If we go back to the formalisation of two-voice counterpoint, we see that each rule applies between the single counterpoint and the *cantus firmus*. For example, when it is stated that each interval must be consonant, this refers to the harmonic interval between the counterpoint and the *cantus firmus*. When considering three-part composition, Fux explains that the rules do not necessarily have to be followed between each counterpoint and the *cantus firmus*, but rather between "each of the voices and the lowest voice" (i.e. the lowest stratum). In other words, Fux says that the main rules apply between the parts and the lowest stratum (whether or not the latter is the *cantus firmus*). This is a big difference with respect to the two-voice formalisation, as it changes the perception we have of the counterpoint writing. In summary, the constraints apply as follows:

- The main constraints apply:
 - Between the *cantus firmus* and the lowest stratum,
 - Between the first counterpoint and the lowest stratum,
 - Between the second counterpoint and the lowest stratum.
- Some constraints apply:

- Between the *cantus firmus* and the first counterpoint,
- Between the *cantus firmus* and the second counterpoint,
- Between the first counterpoint and the second counterpoint,
- Between the three parts altogether (harmonic rules only).

Generalisation of two-part composition to three-part composition

By changing our perspective to consider the interaction between the parts and the lowest stratum, we discover that the two-part composition is a special case: it allows to set the rules between the counterpoints and the *cantus firmus*, without having to consider the lowest stratum. Still, the two-part composition follows the rules of the generalisation, i.e. that the parts should be set in relation with the lowest stratum.

To better understand this, let's recall once again that in the formalisation for two-part composition, the rules are understood to apply between the counterpoint and the *cantus firmus*. In the three-voice formalisation, the rules apply between the parts and the lowest stratum. One might therefore be tempted to conclude that three-voice composition breaks completely with two-part composition, but that would be too hasty a conclusion. Indeed, on closer inspection, the way the rules work in two-part composition (between counterpoint and *cantus firmus*) is just one particular case of this new vision. In two-part composition, too, the rules apply between the parts and the lowest stratum. But of course, since there were only two voices, the lowest stratum was naturally either counterpoint or *cantus firmus*. This means that when constraints were established between the counterpoint and the *cantus firmus*, those same constraints were *de facto* also established between the highest part and the lowest stratum. Considering the rules as being established between the counterpoint and the *cantus firmus* was just a simplification of reality, although it was perfectly correct. The two-part formalisation is considering a convenient special case, not the general case. This is illustrated in Figures 2.4 and 2.5. As we can see on those pseudo-compositions, it does not change anything to apply the constraints between the counterpoints and the *cantus firmus* or between the parts and the lowest stratum.

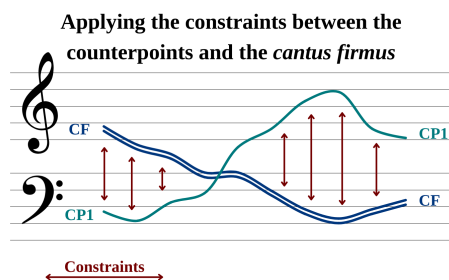


Figure 2.4: Applying the constraints between the counterpoint and the *cantus firmus*

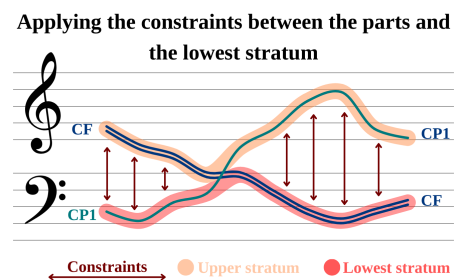


Figure 2.5: Applying the constraints between the parts and the lowest stratum

However, when it comes to generalising the composition of counterpoint for three voices, the same simplification is no longer possible. We are now forced to establish our rules between the parts and the lowest stratum, and no longer between the counterpoints and the *cantus firmus*. In Figures 2.6 and 2.7 it becomes clear that establishing the rules between the counterpoints and the *cantus firmus* is really different from applying them between the various parts and the lowest stratum. In these figures, the parts don't intersect and therefore fit perfectly with the strata, so the constraints are

always applied to the same counterpoint. This was done for the sake of intelligibility of the graphs, but it is of course possible for the parts to cross and for the "target" of the constraints not always to be the same counterpoint.

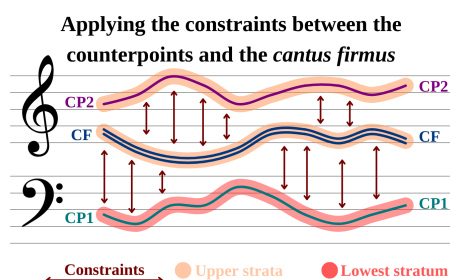


Figure 2.6: Wrong approach: applying the constraints between the counterpoint and the *cantus firmus*.

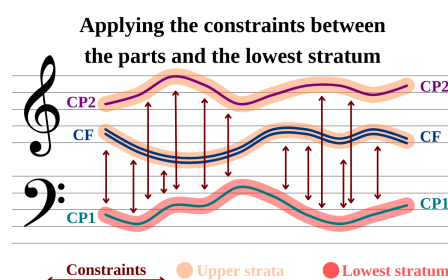


Figure 2.7: Correct approach: applying the constraints between the parts and the lowest stratum.

It is, of course, possible for the *cantus firmus* to be equal to the lowest stratum all along, in which case nothing changes from the perspective we had when composing for two voices. In this particular case, by applying the rules with respect to the *cantus firmus*, we would find ourselves *de facto* applying the rules with respect to the lowest stratum (and we would be back to the situation described above, see Figures 2.6 and 2.7, only that there is now one more part). It is when the *cantus firmus* pitches are higher up than those of the counterpoints that considering the lowest stratum becomes necessary.

A very important detail brought about by this paradigm shift is the following: in the formalisation of two-part composition, the constraints are applied between the counterpoints and the *cantus firmus*, which guarantees that the *cantus firmus* is taken into account in the constraints. But if we now apply the constraints only between the counterpoints and the lowest stratum, there is no longer any guarantee that the *cantus firmus* will be linked to the other voices by any constraints, for example if the *cantus firmus* is not the lowest stratum. Nevertheless, it is important that the relationship between the *cantus firmus* and the lowest stratum is *also* taken into account, not just the relationship between the counterpoints and the lowest stratum. This means that when we apply the constraints between the parts and the lowest stratum, we *must* also apply them to the *cantus firmus* (since the *cantus firmus* is a part, like any of the counterpoints).

2.3 Definitions of the variables used in the formalisation

In this section we define the variables used in the formalisation. Many of these variables were already present in T. Wafflard's work and are reused in this formalisation, with some changes. All these (re)definitions are explained in detail in this section.

Throughout this section, when reference is made to the past ("this variable used to be", "this variable keeps the same definition", ...), it means that reference is made to the previous definition of the variable, which was the one defined in T. Wafflard's work.

Nota bene Please take into consideration that all the rules from T. Wafflard's thesis (which can be found in Appendix C) are fully compatible with the new definitions

of the variables. The reason for this is that the rules from T. Wafflard apply for a specific case of our generalisation (see Section 2.2). They thus also work with the generalisation.

2.3.1 Variables and array notation

First, let's look at how the variables are defined and the notation we use to access them. The variables are defined to capture a compositional reality, such as the notes of the voices, the harmonic intervals between those voices, or many other concepts. They are represented by a letter (N for the notes, H for the harmonic intervals, ...) and are mainly arrays because they have a different value for each beat of the composition. To know which beat we are talking about, we address the variable in a computer notation. For example, $N[i, j]$ means "the note on the i -th beat of the j -th measure". This implies that i ranges from 0 to 3 and that j ranges from 0 to the number of measures (which is written as m).

Once defined, the variables are related to each other according to the formalised rules. The constraint solver searches for all possible values of these variables, according to the constraints, and stops when all variables in N (the pitches) are fixed, as this means that a solution has been reached (the notes of the counterpoints are known, and this is the goal of the solver).

Of course, there are many different solutions for the same *cantus firmus*, and in order to distinguish between two valid solutions (i.e. all solutions that respect all constraints), costs are introduced. Some variables are therefore actually costs, that are intended to convey the preferences expressed by Fux in *Gradus ad Parnassum*. The solver considers a valid solution with a low cost to be better than a valid solution with a high cost. Each individual cost C can be either 0 (no cost), 1 (low cost), 2 (medium cost), 4 (high cost), 8 (last resort) and $64m$ (cost proportional to length). These individual costs C are then combined into a total cost τ that the solver tries to minimise. We will see in Chapter 4 how the costs are actually combined.

2.3.2 Overview of all the variables

We may now take a look at all the variables that are useful throughout this work. You may note that a vast majority of them already exists in T. Wafflard's thesis, but with one big change, each variable is now linked to a voice. To understand this, let's take an example. In a two-part composition, it was obvious that H (the harmonic intervals array) described the intervals between the *cantus firmus* and the only counterpoint. It was also obvious that P (the motions array) described the motions of the single counterpoint. And so it is with all the variables. When writing a three-voice composition, we have many possibilities when we talk about intervals or motions. Intervals between which voices? Movements of which counterpoint? To deal with this, each variable is now related to a voice.

The relationship between a variable and a voice is expressed as a function. $X(v)$ represents the variable X of the voice v . The arguments of the function can be either:

- cf - for linking the variable to the *cantus firmus*.
- cp_1 - for linking the variable to the first counterpoint.
- cp_2 - for linking the variable to the second counterpoint.
- a - for linking the variable to the lowest stratum.
- b - for linking the variable to the intermediate stratum.

- c - for linking the variable to the uppermost stratum.

For example, $X(cf)$ refers to the variable X of the *cantus firmus*.

When a variable is not explicitly linked to a voice, it is implied that the relation expressed for it is true for all *parts*. In other words, if the variable X is written without any precision, it means that we are speaking about the variable X of all parts. Formally:

$$X \equiv \forall v \in \{cf, cp_1, cp_2\} : X(v) \quad (2.4)$$

Note that this *only applies to parts*, not strata. X could still apply to a stratum, but it would then be mentioned as $X(s)$, where s is the layer. This is very important: it allows T. Wafflard's rules for two voices to *remain valid* with the new definition of the variables. It also simplifies the notations.

The following is a summary of the variables used in the formalisation and the type of voices to which they apply. $X(v)$ means that the variable X can be linked to any voice, $X(p)$ means that the variable X only makes sense when linked to a part and $X(s)$ means that the variable X only makes sense when linked to a stratum. The variables are defined in more detail in section 2.3.3.

- $N(v)$ - the notes (pitches) of the voice v . This is the same variable as the variable 'cp' in T. Wafflard's thesis.
- $H(v_1, v_2)$ - the harmonic intervals between voice v_1 and voice v_2 : this variable is particular, as it needs two arguments to be meaningful,
- $M(v)$ - the melodic intervals of the voice v ,
- $P(p)$ - the motions of the part p ,
- $A(p)$ - the boolean array representing whether the part p is the lowest stratum.
- $IsCfB(v)$ - the boolean array representing whether the cantus firmus is lower than the voice v ,
- $IsCons(v)$ - to the boolean array representing whether the voice v is consonant with the lowest stratum or not,
- $S(p)$ - an array specific to the fifth species², representing for each beat what set of rules a note follows. For example $S(p)[0, 0] = 3$ means that the very first note of the part p actually belongs to the third species. This is important since the fifth species is actually a mixture of all the others. As all other variables, there is one per part.

The formalisation also uses some constants, namely:

- $species(p)^3$ - the species of part p — by definition, $species(cf) = 0$,
- $n(p)$ - the number of notes in part p ,
- $s_m(p)$ - the maximum number of notes contained in part p , if all notes were quarter notes, excepted the last one (that is always a whole note),

²The S array is treated in the section about the fifth species 3.6.

³This has nothing to do with $S(p)$. This constant describes the species of the whole part. If p is a fifth species counterpoint, then $species(p) = 5$

- $\text{lb}(p)$ - the lower bound of the range of part p ,
- $\text{ub}(p)$ - the upper bound of the range of part p ,
- $\mathcal{R}(p)$ - the range of part p , i.e. $\mathcal{R}(p) := [\text{lb}(p), \text{ub}(p)]$,
- $\text{borrow}(p)$ - the borrowing scale⁴ of part p ,
- $\mathcal{N}(p)$ - the borrowed notes of part p , where $\mathcal{N}^{\mathcal{R}}(p) = \mathcal{N} \cap \mathcal{R}$
- $\mathcal{B}(p)$ - the set of beats on which a note is played⁵ according to the species of part p ,
- $\text{b}(p)$ - the number of beats⁶ in a measure according to the species of part p ,
- $\text{d}(p)$ - the duration of a note⁷ according to the species of part p ,
- m - the number of measures in the composition, which is the same for all voices,
- **Dis** - the set of all consonances, i.e. $\{1, 2, 5, 6, 10, 11\}$
- **Cons** - the set of all consonances, i.e. $\{0, 3, 4, 7, 8, 9\}$, where Cons_p are the perfect consonances, i.e. $\{0, 7\}$, and Cons_{h_triad} are the consonances of a harmonic triad as defined by Fux, i.e. $\{0, 3, 4, 7\}$.

Please note that the constants can only be linked to the parts, never to a stratum. Indeed, it would have no sense to speak about the species of a stratum or about the extended domain of a stratum.

The formalisation also uses the variables $\mathcal{C}$ (the cost factors) and τ (the total cost). A summary of all the costs can be found in Table B.1. To refer to a given cost, the formalisation uses subset notation. For example, ‘example cost’ is written as $\mathcal{C}_{\text{example}}$.

To make sure that those notations are clear, here are some examples: the notation $N(a)$ corresponds to the variable representing the notes (pitches) of the lowest stratum, whereas $N(cf)$ are the notes of the *cantus firmus*. The species of the second counterpoint is written $\text{species}(cp_2)$. If only N is written, then the equation in which N is located holds true for any possible *part*. That is, the relationship $N[0, 0] = 60$ would mean: the pitch of the first note of *all parts* must be a middle C (whose representation in MIDI is 60).

⁴Remember that Fux mainly uses notes without a flat or sharp, which means that if the composition begins with a C, it will be written in Ionian mode, if it begins with a G, in Mixolydian mode, etc. Borrowing mode is a way for counterpoint to access notes that are not originally in its mode. For example, a counterpoint in E (i.e. Phrygian mode) that has the major mode (E major, i.e. Ionian E) as its borrowing mode will also have access to the notes F#, C#, G# and D#.

⁵To make it clearer: for the first species, the only beat in a measure is $\{0\}$, as there is only a note on the first beat. For the second species, the set of beats is $\{0, 2\}$. For the third species, it is: $\{0, 1, 2, 3\}$. For the fourth species: $\{0, 2\}$. And for the fifth species: $\{0, 1, 2, 3\}$.

⁶Thus, it is always equal to the size of the set $\mathcal{B}(p)$.

⁷For the first species, it is equal to 1, as each note is a whole note. For the second species, it is $\frac{1}{2}$, for the third, it is $\frac{1}{4}$, for the fourth, it is $\frac{1}{2}$, and for the fifth, it is $\frac{1}{4}$. It is always equal to $\frac{1}{b(p)}$.

Note regarding the fourth species

Let's recall that the fourth species behaves in a particular way compared to the other species. First of all, it is exclusively composed of syncopations. Its notes are half notes, always linked two by two from bar to bar, producing a pitch change in the middle of the measure, on the upbeat. This gives the impression of hearing a whole note that is constantly shifted by two beats, in other words: syncopation. Concretely; the syncopation means that the beats of the fourth species should be considered as "shifted": its upbeat should be considered as the downbeat, and its downbeat as the upbeat of the previous measure. This means that in the majority of cases, the equations for the fourth species would have to be rewritten, swapping the 0 and 2 indexes ($H[0, j]$ becomes $H[2, j]$ in the fourth species and $H[2, j]$ becomes $H[0, j+1]$ in the fourth species). To avoid duplication of the equations (a first equation for all species and a second equation for the fourth species) and also to avoid equations that were too complex and difficult to read, it was decided to make the index swap implicit.

Here is an example: $H[0, 0] = 7$ should be understood as $H[2, 0] = 7$ if it concerns a fourth-species counterpoint.

2.3.3 In depth definition of the variables

$N(v)$ notes

N is the array corresponding the pitches of each voice. Its size is s_m . It is the same array as the one named 'cp' in T. Wafflard's thesis, and it got renamed to N (for notes), for the sake of clarity. As we have now three of those arrays (one for the first counterpoint, one for the second counterpoint, and even one for the *cantus firmus*), it needed a less ambiguous name than the one it had before.

The notes in array N are written in MIDI format. This means that a middle C (C_4 in scientific pitch notation) is represented by 60, and each semitone has a value of 1. So, starting from the middle C, we have:

$$\{C_4, D_4, E_4, F_4, G_4, A_4, B_4\} \equiv \{60, 62, 64, 65, 67, 69, 71, 72\}.$$

$H_{(abs)}(v_1, v_2)$ h-intervals h-intervals-abs h-intervals-to-cf ...

This variable is an array of size s_m and represents the harmonic intervals between voice v_1 and voice v_2 . It is the only variable that is associated with two different voices. So $H(v_1, v_2)[i, j]$ represents the intervals between the i th beat of the j th measure of voice v_1 and the *first* beat of the j th measure of voice v_2 . v_1 can be a part and v_2 can be a stratum, since you can calculate harmonic intervals between a part and a stratum. If v_2 is not specified, it is equal to a by default. In other words, $H(v_1)$ represents the intervals between the voice v_1 and the lowest stratum: $H(v_1) \equiv H(v_1, a)$. This default value was chosen because the most relevant harmonic intervals are those between the voices and the lowest stratum.

Some important values the H variable can take: a minor third is 3, a major third is 4, a fifth is 7 and an octave is 12 (or 0 if modulo).

$$\begin{aligned} \forall v_1, v_2 \in \{cf, cp_1, cp_2, a, b, c\}, \quad \forall i \in \mathcal{B}(v_1), \quad \forall j \in [0, m) : \\ H_{abs}(v_1, v_2)[i, j] &= |N(v_1)[i, j] - N(v_2)[0, j]| \\ H(v_1 - v_2)[i, j] &= H_{abs}[i, j] \bmod 12 \\ \text{where } H_{abs}[i, j] &\in [0, 127], H[i, j] \in [0, 11] \end{aligned} \tag{2.5}$$

$M_{(brut)}^{(x)}(v)$ m-intervals-brut

The variable M represents the melodic intervals of a voice. It can be evaluated either on a part or on a stratum, each of these situations leading to different behaviours.

- $M_{brut}^x(p)$ (i.e. when related to a part) represents the melodic intervals of the part p , and its mode of operation remains the same as in T. Wafflard's thesis: $M_{brut}^x[i,j]$ represents the melodic interval between $N[i,j]$ and the note that comes x notes later. For example, if $N[0,0] = 60$ and $N[1,0] = 67$, the interval is 7. More exactly, M^x represents the melodic interval between the current beat and the x th following note, according to $d(p)$ ⁸. M represents the absolute value of M_{brut} . By default, $M \equiv M^1$, thus, $M \equiv \forall p \in \{cf, cp_1, cp_2\}: M^1(p)$.

$\forall x \in \{1, 2\}, \forall i \in \mathcal{B}, \forall j \in [0, m - x):$

$$M_{brut}^x[i, j] = \begin{cases} N[i + xd, j] - N[i, j] & \text{if } i + xd < 4 \\ N[(i + xd) \bmod 4, j + 1] - N[i, j] & \text{if } i + xd \geq 4 \end{cases} \quad (2.6)$$

$$M^x[i, j] = |M_{brut}^x[i, j]|$$

where $M_{brut}^x[i, j] \in [-12, 12]$, $M^x[i, j] \in [0, 11]$

- $M_{brut}(s)$ (i.e. when related to a stratum) has its own way of working, that is defined in the next paragraphs. We focus specifically on $M(a)$, the melodic intervals of the lowest stratum.

Why is it complicated to consider melodic intervals of strata? Since strata don't have melodic intervals *per se* (they actually do have melodic intervals, but it doesn't really make sense to consider them), we need to redefine what we mean when speaking about the melodic intervals of a stratum. If it is not clear why strata have no inherent melodic intervals, remember that strata are an abstract concept that is used only in mathematical relationships (and respective constraints). People who listen to the music hear the different parts (be they different tessitura, different instruments, ...) and the way these parts interact together in melodic movements and harmonic convergences, rendering a beautiful music, or not. Strata are an abstraction of the harmonic interactions between the parts, and because of this, they are a consequence of the parts: they exist because the parts exist, and not the other way round! And since strata are defined according to harmonic principles (as was suggested before, they are successions of vertical alignments), speaking about the proper *melodic* intervals of a stratum makes no sense. One could then conclude that melodic intervals do not apply to strata, and go ahead. Nevertheless, Fux *does* speak about computing the motions between a part and the lowest stratum. And to be able to compute motions, one needs to compare two different melodic intervals. So we need to have a definition for the melodic intervals of a stratum.

Definition of $M(s)$, the melodic intervals of a stratum To understand how we arrive at a definition for the melodic intervals of a stratum, we need to remember that the lowest stratum is just the collection of all the lowest-sounding notes in the composition. It is therefore quite logical to think of the melodic intervals of the lowest stratum as the melodic intervals that lead to all those lowest-sounding notes. We thus define the melodic intervals of the lowest stratum to be: the interval that lead to the note of the lowest stratum in the corresponding part. Let's make this clearer with an example.

⁸If $d(p)=x$, it means that the following note comes in x beats. So if $d=1$, $M^2[0,0]$ represents the interval between $N[0,0]$ and $N[2,0]$.

Let's consider that the lowest stratum consists of the notes $[C_{cp_1}, E_{cf}, G_{cp_2}]$ (where C_{cp_1} indicates that the C belongs to the first counterpoint), that in the *cantus firmus* the interval that lead to the E is a +0 (i.e. staying on the same note), and that in the second counterpoint the interval that lead to the G was a -4 (getting down of two tones). The corresponding melodic intervals array of the lowest stratum would then be $[+0, -4]$. This example has been written again in a more visual way in equation 2.7 to make it easier to understand. To the left of the equation is the pitch array of each voice mentioned. To the right of the equation is the melodic interval array of each voice mentioned. The numbers in bold red are those corresponding to the lowest stratum.

$$\begin{aligned}
 N(cf) &= [64, \textbf{64}, 71] & M_{brut}(cf) &= [+0, +7] \\
 N(cp_1) &= [\textbf{60}, 67, 74] & M_{brut}(cp_1) &= [+7, +7] \\
 N(cp_2) &= [72, 71, \textbf{67}] & M_{brut}(cp_2) &= [-1, \textbf{-4}] \\
 \\
 N(a) &= [\textbf{60}, \textbf{64}, \textbf{67}] & M_{brut}(a) &= [+0, \textbf{-4}]
 \end{aligned} \tag{2.7}$$

The formal definition of the melodic intervals of the lowest stratum is hence as follows: the melodic interval in measure j of the lowest stratum is equal to the last melodic interval in measure j of the part that is the lowest stratum in measure $j + 1$. Remember that this complex definition is needed in order for the computation of the motions to work fine, and that the motions of the lowest stratum, just as the lowest stratum, are an abstract notion that serves only in formulas and constraints. The motions of the lowest stratum *do not intend to represent any concrete motion really happening in the composition, nor does it correspond to the melodic intervals between the pitches of the lowest stratum*. It may be worth noting that if a part is the lowest stratum all the time, then the motions of the lowest stratum will be completely equal to those of that part.

$$\forall j \in [0, m - 1] :$$

$$M_{brut}(a)[j] = \begin{cases} M_{brut}(cf)[0][j] & \text{if } A(cf)[j + 1] \\ M_{brut}(cp_1)[\max(\mathcal{B}(cp_1))][j] & \text{if } A(cp_1)[j + 1] \\ M_{brut}(cp_2)[\max(\mathcal{B}(cp_2))][j] & \text{if } A(cp_2)[j + 1] \end{cases} \tag{2.8}$$

It might be helpful to have a look at Figures 2.8 and 2.9 to understand better how the melodic intervals arrays for the lowest stratum. The melodic intervals of the lowest stratum are those that lead to the notes of the lowest stratum.

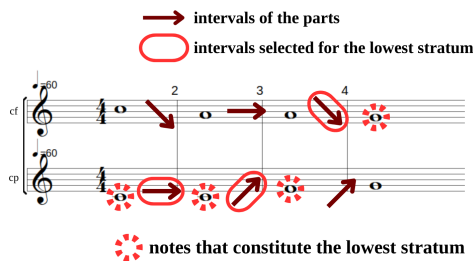


Figure 2.8: Understanding the melodic intervals of the lowest stratum with a first species counterpoint

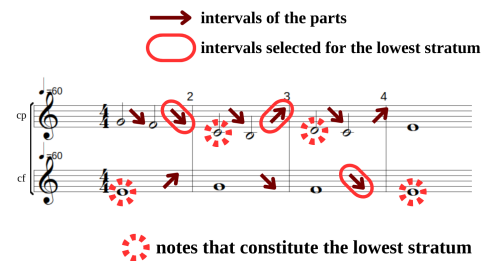


Figure 2.9: Understanding the melodic intervals of the lowest stratum with a second species counterpoint

P(p) motions

The motions array represents the motions⁹ of a part p with respect to the lowest stratum. Of course, to be able to compute the motions between two voices, we must compare their melodic intervals, hence, we must deal with melodic intervals of a stratum. This is not a problem anymore since we have defined what the melodic intervals of the lowest stratum mean in the previous sub-section. However, a problem arises when computing the motions of the part that is also the lowest stratum in some measures. When this happens, we end up calculating motions between a part and itself. Any part is inevitably moving in direct motion with itself, and this situation leads to only direct motions being calculated. This becomes problematic when using the P array in some constraints. To tackle this problem, the motions of a part are equal to -1 when the part is also the lowest stratum (which is denoted $A(p)$, see Section 2.3.3). This value can be considered as a 'non-applicable' value: this part has no motion with respect to the lowest stratum, as it is currently the lowest stratum.

$$\begin{aligned}
 & \forall p \in \{cf, cp_1, cp_2\}, \quad \forall x \in \{1, 2\}, \quad \forall i \in B, \quad \forall j \in [0, m-1), \quad x := b - i \\
 & motion(p)[i, j] = \begin{cases} 0 & \text{if } (M_{brut}^x(p)[i, j] > 0 > M(a)_{brut}[j]) \\ & \quad \vee (M_{brut}^x(p)[i, j] < 0 < M(a)_{brut}[j]) \\ 1 & \text{if } M_{brut}^x(p)[i, j] = 0 \oplus M(a)_{brut}[j] = 0 \\ 2 & \text{if } (M_{brut}^x(p)[i, j] > 0 \wedge M(a)_{brut}[j] > 0) \\ & \quad \vee (M_{brut}^x(p)[i, j] < 0 \wedge M(a)_{brut}[j] < 0) \\ & \quad \vee (M_{brut}^x(p)[i, j] = 0 = M(a)_{brut}[j]) \end{cases} \\
 & P(p)[i, j] = \begin{cases} -1 & \text{if } A(p)[j] \\ motion(p)[i, j] & \text{if } \neg A(p)[j] \end{cases} \tag{2.9}
 \end{aligned}$$

Equation 2.9 may seem daunting, but it is actually very simple (just a little verbose). It works like this:

For each beat in the composition:

- If a part is also the lowest stratum, P is -1 (i.e. non applicable, otherwise we would calculate the motion between the part and itself).
- If the part moves in the opposite direction to the lowest stratum, P is 0.
- If the part stays where it is and the lowest stratum moves (or vice versa), P is 1.
- If the part moves in the same direction as the lowest stratum, P is 2.

A(p) is-lowest

A is an array of boolean variables of size m (the number of measures), where each variable indicates whether the part p is the lowest stratum. In other words, $A(p)$ is true if p is the lowest stratum. The notation " A " was chosen as the uppercase of " a ", which itself represents the lowest stratum. It is also worth to be noted that only one of the parts can be the lowest stratum at the time. This does not mean that two parts

⁹Reminder: there are three types of motion: direct, when both voices move together, contrary, when one voice moves up and the other moves down, and oblique, when one voice doesn't move and the other does

cannot equal the lowest stratum at the same time, it is indeed possible that two parts blend in unison in the final chord, and that both pitches are the lowest sounding notes. It means that only one of those parts is going to be considered to *be* the lowest stratum (and the other one will be the intermediate stratum). See Section 2.1 for more details about this bijection.

Here is the mathematical definition of the A array:

$$\begin{aligned}
& \forall j \in [0, m): \\
& A(cf)[j] = \begin{cases} \top & \text{if } N(cf)[0, j] = N(a)[0, j] \\ \perp & \text{else} \end{cases} \\
& A(cp_1)[j] = \begin{cases} \top & \text{if } (N(cp_1)[0, j] = N(a)[0, j]) \wedge \neg A(cf)[j] \\ \perp & \text{else} \end{cases} \\
& A(cp_2)[j] = \begin{cases} \top & \text{if } \neg A(cf)[j] \wedge \neg A(cp_1)[j] \\ \perp & \text{else} \end{cases}
\end{aligned} \tag{2.10}$$

As can be seen in these equations, only the downbeat of each measure is taken into account when computing the A array. The reason for this is that it is the downbeat note that determines which chord will be the chord of the measure, while the other beats are just fioritures. Another reason for this is that the A variable only serves in contexts where the first note of the measure is relevant.

In practice, there is only an *is-not-bass* array in the code (which is then equal to $\neg A$), as it is almost always more useful to know if a part is *not* the lowest stratum than knowing if it is the lowest one.

IsCfB(p) *is-cf-lower-arr*

This boolean array is true when the *cantus firmus* is lower than the considered part and false otherwise:

$$\begin{aligned}
& \forall p \in \{cp_1, cp_2\} \quad \forall i \in \mathcal{B}(p), \forall j \in [0, m) \\
& IsCfB(p)[i, j] = \begin{cases} \top & \text{if } N(p)[i, j] \geq N(cf)[0, j] \\ \perp & \text{otherwise} \end{cases}
\end{aligned} \tag{2.11}$$

IsCons(p) *is-cons-arr*

This boolean array is true when the harmonic interval with the lowest stratum at a given index is a consonance and false otherwise:

$$\begin{aligned}
& \forall i \in \mathcal{B}, \forall j \in [0, m) \\
& IsCons[i, j]_{(all, p, imp)} = \begin{cases} \top & \text{if } H[i, j] \in Cons_{(all, p, imp)} \\ \perp & \text{otherwise} \end{cases}
\end{aligned} \tag{2.12}$$

Chapter 3

Formal rules for three-part counterpoint

The purpose of this section is to extract all the rules that Fux mentions in his work and to make sure that they are unambiguous.

It consists of seven sections: first, a section for the implicit rules (the rules that Fux doesn't mention, but are present in all his examples), then a section for each species, and finally a short section that considers the interaction between different species. There is no section on rules for all species, yet there *are* rules that apply to all species. In fact, all the rules mentioned in the first species section also apply to the other species. Fux doesn't explicitly mention this point, but it becomes clear when you look at how he teaches and applies these rules in composition. Therefore, **all rules from the first species apply to all species**.

Each species section is divided into two subsections: the first subsection is about setting up Fux's rules and discussing them in English. The second subsection is about translating the rules into formal logic.

Some important notes

- **Concerning the green dot** — All the rules from T. Wafflard's thesis still apply to three-part compositions. The numbering of the rules in this work is consistent with T. Wafflard's work¹. If a rule is defined with the same number as an existing rule for two-part compositions, this means that the corresponding rule from T. Wafflard's work does not apply to three-part composition, and that the new rule should be used instead. To make this clearer, there is a green dot (●) next to the rules from T. Wafflard's thesis that got redefined when used in a three-voice composition.
- **Concerning the [PREF] marker** — Some rules are mandatory and must be followed to find a valid solution, and other rules are just preferences that can be followed to find a *better* solution. The preferences have been marked '[PREF]' throughout the chapter clearly mark which rule is mandatory and which is a preference.
- **Concerning the default costs** — Each time a cost is mentioned in the formalisation, the value corresponding to it is its default value, as defined in Appendix B. Note, however, that in practice the value for each cost can be changed by the user to suit their needs.
- **Concerning the purpose of all the rules** — Some rules are concerned with musicality: composing counterpoint that sounds nice; for example, the rule that forbids dissonance. Some rules are concerned with singability: composing counterpoint that is not too difficult for the human voice to sing; for example, the rule forbidding a melodic leap greater than a sixth.

¹The whole set of rules (two-part composition and three-part composition) can be found in Appendix C.

3.1 Implicit rules

These implicit rules apply to all types of counterpoint. They are never explicitly defined by Fux, but are derived from his many examples. These implicit rules are therefore actually used by Fux, even though he doesn't talk about them.

3.1.1 Formalisation in English

1.H2 and 1.H3 • *First and last notes have not to be perfect consonances anymore.*

Fux doesn't state this in his text, but in many of his examples, we see that when he composes with three voices (and more), the first and last harmonic intervals between the parts and the lowest stratum are not necessary a perfect consonance anymore.

1.H7 and 1.H8 • *The harmonic interval of the penultimate measure must be either a minor third, a perfect fifth, a major sixth, or an octave.*

In two-part composition rules, Fux said that the last harmonic interval had to be either a minor or a major third. This is something he doesn't respect at all in three-part composition, but we can see that he still tries to use minor third and major sixths when possible, in the penultimate measure. The rule from two-part species was thus rewritten in order to be appropriate: either use a perfect consonance, a minor third, or a major sixth.

G8 *The last chord must be composed only of the notes of the harmonic triad.*

Again, this isn't stated explicitly, but we see that all of his examples end with a chord containing exclusively the notes of the harmonic triad.

G9 *The last chord must have the same fundamental as the one of the scale used throughout the composition.*

This rule emanates from an observation of Fux's examples throughout the chapter. The last chord of all his compositions always have the same fundamental as the fundamental of the scale used throughout the composition. When the *cantus firmus* is the lowest stratum, this is not a problem, as the *cantus firmi* always end with the fundamental note of the scale. But when not, it has to be imposed by a constraint, or we may end up with surprising results.

3.1.2 Formalisation into constraints

1.H2 and 1.H3 • *First and last notes have not to be perfect consonances anymore.*

There is no constraint associated with this rule, as it is a relaxation of a rule from the two-part composition rule set.

1.H7 and 1.H8 • *The harmonic interval of the penultimate measure must be either a minor third, a perfect fifth, a major sixth, or an octave.*

$$H[0, m - 2] \in \{0, 3, 7, 9\} \quad (3.1)$$

G8 *The last chord must be composed only of the notes of the harmonic triad.*

$$\forall s \in \{b, c\}: H(s)[0, m - 1] \in \text{Cons}_{h_triad} \quad (3.2)$$

G9 *The last chord must have the same fundamental as the one of the scale used throughout the composition.*

Since the fundamental of the scale is defined by being the first note of the *cantus firmus*, we impose that the last note of the lowest stratum must be equal to the first one of the *cantus firmus* (taking the modulus into account).

$$N(a)[0, m - 1] \mod 12 = N(cf)[0, 0] \mod 12 \quad (3.3)$$

3.2 First species

This section deals with the rules that apply to the first species of counterpoint. As mentioned earlier, these rules apply to *all* species in the context of a three-part composition. In other words, these rules apply whenever $species(p) \in \{0, 1, 2, 3, 4, 5\}$. More specifically, the rules in this section apply to the first beat of each species, except for the fourth species, where they apply to the third beat (see the note on the fourth species 2.3.2).

The first species consists of whole notes only. It is the basis of counterpoint and its simplest case.



Figure 3.1: Example of a first species counterpoint in three-part composition

3.2.1 Formalisation into English

Structural constraints

1.S1 *All notes are whole notes.*

"This species consists of three whole notes in each instance." Mann [28, p.71]

This pretty straightforward rule is the very definition of the first species. It adds nothing in comparison with the rules for the two part comparison. It is hence already implemented by the first species for two voices and does not need any consideration.

Harmonic rules

1.H1 ● *All notes on the downbeat are consonant with the notes of the lowest stratum.*

"This species consists of three notes, the upper two being consonant with the lowest." Mann [28, p.71]

This rule is an update of previous **1.H1** (that previously was saying that *all* intervals must be consonants). Fux states that the upper voices and the lowest one are consonant, and not all voices together.

1.H8 [PREF] *The harmonic triad should be used as much as possible.*

"The harmonic triad should be employed in every measure if there is no special reason against it." Mann [28, p.71]

As a footnote states it [28, footnote, p.71], Fux refers to the "harmonic triad" as being a chord in this position: 1-3-5 (contrary to what is today understood as a harmonic triad). The rule says it is not obligated, but it is preferred, to use the 1-3-5 chord, considering that 1 is the lowest voice.

1.H9 *One might use sixths or octaves.*

"Occasionally, one uses a consonance not properly belonging to the triad, namely, a sixth or an octave." Mann [28, p.72]

Here, Fux explains that when it is not possible to have a harmonic triad, you can use sixths or octaves instead. Remember that the sixths or the octaves are calculated from the lowest stratum. Since the rule **1.H1** obligates the use of a perfect consonance (i.e. a third, a fifth, a sixth or an octave), when the harmonic triad cannot be used, it is already naturally replaced by a third or a sixth, because no other intervals are allowed. It is thus not a new rule but a restatement of rule **1.H1**.

1.H10 *Tenths are prohibited in the last chord.*

"One feels that the degree of perfection and repose which is required of the final chord does not become sufficiently positive with this imperfect consonance [(speaking about a tenth)]." Mann [28, p.77]

When Fux says this, he takes a tenth as an example, but it here understood that the final chord cannot include a tenth (third + octave), nor an eight-teenth (third + two octaves), etc. Nevertheless, "simple" thirds are considered completely valid.

1.H11 [PREF] *Octaves should be preferred over unisons.*

"Unison is less harmonious than the octave." Mann [28, p.79]

This rule does not bring anything new, as there is already a rule stating that two parts cannot blend in unison (rule **1.H5** in appendix). If no unison is possible, then the octaves will always be preferred over the unison (since the latter is not possible).

1.H12 *Last chord cannot include a minor third.*

"The minor third is not capable of giving a sense of conclusion." Mann [28, p.80]

Fux later states that minor modes should not include a third altogether, but that sometimes it is impossible to do without it, so the major third *is* allowed in minor modes.

Melodic rules

The following rules obviously don't apply to the *cantus firmus*, since its melody is already fixed.

1.M3 [PREF] *Steps are preferred to skips.*

"[Each part] follows the natural order closely." Mann [28, p.73]

After having said this, Fux complements his explication by saying the counterpoints should be "moving gracefully, stepwise without any skip". This is clearly a preference, and has already been covered when implementing the first species for two voices (see rule **G8**). It can thus be ignored in the scope of this thesis.

1.M4 [PREF] *The notes of each part should be as diverse as possible.*

"[Each part] follows the principle of variety." Mann [28, p.73]

Fux never clearly defines what he means by the "principle of variety". So we try to define it according to what we can read in his work. The examples he gives are a great help. He first writes an incorrect example and then corrects it, saying that the corrected example is better because it follows the principle of variety more closely. The difference between the two examples is that the number of different pitches has been increased. So we can define the principle of variety as: use as many different notes as possible in a single voice.

The principle of variety is therefore clearly a preference.

This principle is very interesting because it introduces the only rule *across all* rules that has a scope greater than two measures. While without this rule the solver uses constraints that apply only to one measure (think of harmonic constraints) or sometimes to two measures (think of melodic constraints), this constraint is the first to give the solver some "memory" about the composition. And although seven measures may seem like a small range, it means that the constraint covers a large part of the composition (when dealing with compositions of 10 to 15 measures, of course).

1.M5 *Each part should stay in its voice range.*

"One should not exceed the limits of the five lines without grave necessity." Mann [28, p.79]

Fux says here that each part should stay on the musical staff (Fux's "five lines"). Since every staff can be represented differently according to the clef that is used, this rule could be always true. Obviously, Fux meant the staff corresponding to the voice range (treble clef for a soprano, bass clef for a bass, ...).

This is actually something that is already handled when declaring the $N(p)$ arrays, as they are declared with an upper and lower bound ($ub(p)$ and $lb(p)$), corresponding to their voice range.

1.M6 *Melodic intervals cannot be greater or equal to a sixth.*

"The skip of a major sixth is prohibited." Mann [28, p.79]

This rule is only a restatement of rule **1.M2**, saying that melodic intervals cannot exceed a minor sixth interval.

Motion rules

1.P1 • [PREF] *Reaching a perfect consonance by direct motion should be avoided.*

"[Reaching] perfect consonance by direct motion [is allowed if] there is no other possibility." Mann [28, p.77]

This is a new rule as it only applies to three-part composition, but it cancels an already existing rule that used to be applied in two-part composition. The same rule in two-part composition states that it is prohibited to reach a perfect consonance using a direct motion. In three-part composition, this is not prohibited anymore, as not doing it is sometimes impossible, and you may thus derogate from this rule.

1.P4 [PREF] *Successive perfect consonances should be avoided.*

"The necessity of avoiding the succession of two perfect consonances [...]." Mann [28, p.72]

Fux implies here that there should not be two consecutive perfect consonances. He does not specify whether this rule applies to all three parts at once (i.e. if there was a consonance between part 1 and part 2 in measure X, there cannot be one between part 2 and 3 in measure X+1), or whether it applies to each pair of parts separately. However, in his example (Fig. 91 of the English version [28]) we can clearly see that there is a perfect consonance in every measure (parts 1-3, then 1-2, then 1-3, then 2-3, then 1-2). From this we can deduce that *for each pair of parts* it is forbidden for two perfect consonances to follow each other.

However, a closer look at his examples throughout the book reveals that Fux does not respect this rule at all. To name just a few places where this rule does not apply, let's mention figure 108, in the first three measures, between the bass and the *cantus firmus*; figure 109, in the same place, between the *cantus firmus* and the alto; figure 110, in measures 8 and 9, between the bass and the alto. For this reason, this rule must be considered as a preference rather than an absolute constraint.

To make this a preference is still very surprising, since many authors of counterpoint consider the succession of perfect consonances to be completely forbidden [34]. However, we are concentrating here on the Fux formalisation, and the possibility of completely forbidding perfect consonance successions is a choice offered to the user in the interface.

1.P5 *Each part starts distant from the lowest stratum.*

"To allow enough space for the voices to move toward each other by contrary motion, the upper voices begin distant from the bass." Mann [28, p.75]

This preference cannot be made clearer: the voices start distant from the lowest stratum.

1.P6 *It is prohibited that all parts move in the same direction.*

"All voices ascend[ing] [is] a progression which can hardly be managed without awkwardness resulting." Mann [28, p.76]

What Fux is explaining here is simply that the three parts cannot move in the same direction. In other words, if two voices go up, the last one cannot go up. If two voices go down, the last one cannot go down. And if two voices stand still, the last one must move.

1.P7 *It is prohibited to use successive ascending sixths on a direct upwards motion.*

"Ascending sixths on the downbeat sound harsh." Mann [28, p.77]

This rule is quite simple and states that if one harmonic interval is a sixth, then the next harmonic interval cannot also be a sixth.

3.2.2 Formalisation into constraints

Structural constraints

1.S1 *All notes are whole notes.*

This rule needs no special constraint, since it is the very definition of the first species to consist only of whole notes.

Harmonic rules

1.H1 *All notes on the downbeat are consonant with the notes of the lowest stratum.*

The new definition of variable H already captures the change in the rule. This means that the equation of the previous rule **1.H1** defined for two voices stays the same.

1.H8 [PREF] *The harmonic triad should be used as much as possible.*

As this rule is actually a preference and not a mandatory rule, it has been implemented as a cost. If the harmonic triad is used, then the cost is 0. Else, it is 1.

$$\begin{aligned} \forall j \in [0, m-1]: \\ (H(b)[0, j] \notin \{3, 4\}) \vee (H(c)[0, j] \neq 7) \iff C_{prefer-harmonic-triad}[j] = 1 \end{aligned} \quad (3.4)$$

1.H9 *One might use sixths or octaves.*

As discussed in the previous subsection, there is no constraint to add for this rule.

1.H10 *Tenths are prohibited in the last chord.*

$$H_{brut}[0, m-1] > 12 \implies H[0, m-1] \notin \{3, 4\} \quad (3.5)$$

If the harmonic interval is bigger than an octave, then you cannot use thirds anymore.

1.H11 [PREF] *Octaves should be preferred over unisons.*

As discussed before, there is no constraint to add for this rule.

1.H12 *Last chord cannot include a minor third.*

$$H[0, m-1] \neq 3 \quad (3.6)$$

Melodic rules

1.M3 [PREF] *Steps are preferred to skips.*

As discussed in the previous subsection, there is no constraint to add as it already exists (see rule **G8**).

1.M4 [PREF] *The notes of each part should be as diverse as possible.*

As it is not explained either if this has to be true for the whole partition or only for two following notes, it has been chosen as an arbitrary seven successive notes to apply the rule on. This means that the solution is penalized if a note in measure X was already present in measures $[X-3, X+3]$. This amount was chosen because it represents the number of flat notes that exist, pushing for the solver to find a solution that contain all of them.

$$\begin{aligned} \forall p \in \{cp_1, cp_2\}, \quad \forall j \in [0, m), \quad \forall k \in [j+1, \min(j+3, m-1)] : \\ N(p)[0, j] = N(p)[0, j+k] \iff C_{variety}[j+m*k] = 2 \end{aligned} \quad (3.7)$$

1.M5 *Each part should stay in its voice range.*

As was discussed before, there is no constraint associated to this rule, as it is already covered by the definition of the voice range.

1.M6 *Melodic intervals cannot be greater or equal to a sixth.*

As was said before, no constraint must be implemented for this rule as it is a re-statement of rule **1.M2**.

Motion rules

1.P1 • [PREF] *Reaching a perfect consonance by direct motion should be avoided.*

Since there is no way in constraint programming to implement a rule that must be obeyed only if possible other than by using a cost, the initial constraint was rewritten to a new one.

$$\begin{aligned} \forall j \in [0, m-1) : \\ P[0, j] = 2 \wedge H[0, j+1] \in Cons_p \iff C_{direct_move_to_p_cons}[j] = 8 \end{aligned} \quad (3.8)$$

Remember: $P[0, j] = 2$ means that the motion is direct.

1.P4 [PREF] *Successive perfect consonances should be avoided.*

As discussed before, this rule is actually a preference.

$$\begin{aligned} \forall p_1, p_2 \in \{cf, cp_1, cp_2\}, \text{ where } p_1 \neq p_2, \quad \forall j \in [0, m-1) : \\ (H(p_1, p_2)[0, j] \in Cons_p) \wedge (H(p_1, p_2)[0, j+1] \in Cons_p) \\ \implies C_{succ_p_cons} = 2 \end{aligned} \quad (3.9)$$

The cost has been set to two according to the cost hierarchy defined in T. Wafflard's thesis (a cost of two is a medium cost), but it is possible for the user to change this cost. The costs are discussed in detail in Section 4.2.

1.P5 *Each part starts distant from the lowest stratum.*

This is not a strict rule but an indication to make easier for the composer to have contrary motions. Since this is neither a requirement nor a preference, it can simply be added as a heuristic for the solver. This is discussed in Section 4.1.2, on heuristics.

1.P6 *It is prohibited that all parts move in the same direction.*

To prevent this, we need only look at the motions between the parts and the lowest stratum. If one of their motions is contrary, then it is guaranteed that the three voices will not go in the same direction (because at least one is contrary). The same applies if one of the motions is oblique. The problem arises when all the movements are direct,

because this would mean that the three voices are going in the same direction. So at least one motion must be something other than direct. Remember that 0 represents a contrary motion, 1 represents an oblique motion and 2 represents a direct motion.

$$\forall j \in [0, m-1):$$

$$\bigvee_{p \in \{cf, cp_1, cp_2\}} P(p)[0, j] \in \{0, 1\} \quad (3.10)$$

1.P7 *It is prohibited to use successive ascending sixths on a direct upwards motion.*

Either the harmonic interval is not a sixth in any of both positions, or one of them is not moving up.

$$\forall j \in [1, m-1), \quad \forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ where } p_1 \neq p_2, \quad \text{sixth} := \{8, 9\}:$$

$$(H(p_1, p_2)[0, j-1] \notin \text{sixth}) \vee (H(p_1, p_2)[0, j] \notin \text{sixth}) \quad (3.11)$$

$$\vee M(p_1)[0, j] > 0 \vee M(p_2)[0, j] > 0$$

3.3 Second species

The second species consists only of half notes. It introduces more dissonance than was possible with the first species.

All the rules in this section apply only when $species(p) = 2$, with p being the part mentioned in the rule. Note that the rules for the first species also apply to counterpoints of the second species.



Figure 3.2: Example of a second species counterpoint in three-part composition

3.3.1 Formalisation in English

Harmonic rules

2.H4 *Major thirds are now allowed in the last chord.*

"A major third [may] appear in the last chord." Mann [28, p.87]

This is a consequence of now using three voices instead of two. Fux makes explicit two implicit rules we had already defined (**1.H2** and **1.H3** and **G8**). It has thus already been implemented in the first species for two voices.

2.H5 *The half notes must be coherent with respect to the whole notes.*

"The half notes are always concordant with the two whole notes." Mann [28, p.88]

One might ask what Fux meant when he wrote "concordant". Did he mean to say "consonant"? Our take on the question is that he meant that the half notes are written whilst taking the whole notes into account. This interpretation is aligned with the French translation, and even with the Latin original. In other words, Fux just says "there are constraints on the half notes". It is thus not a rule *per se*.

Melodic rules

2.M2 • *It is allowed to ligate the fourth-to-last with the third-to-last or to ligature the third-to-last with the second-to-last.*

"Ligatures have no place in this species [except] in the final cadence."
Mann [28, p.87]

Fux explains that in some cases, you have no other option than ligaturing the fourth-to-last and the third-to-last notes. The reasons he gives for this are all part of the previous mentioned rules (no successive perfect consonances, no unison, ...).

Later on, he also says that the third-to-last and the second-to-last notes can be ligatured (hence producing a whole note).

"A whole note may occasionally be used in the next to last measure."
Mann [28, p.93]

He says that in the chapter about third species, but it seems that this applies even in cases where the second species is not used in combination with the third (see figures 134, 173 and 174 of the English version).

He doesn't state clearly if the three of them can get ligatured, but it seems quite obvious that this is not allowed, as it would introduce a lot of redundancy in the composition. It is hence decided that the rule is: a ligature may happen in one case or in the other, but not in both.

We thus have to relax the already existing constraint from two-part composition stating that no two consecutive notes can be the same, to accept it in some cases.

Motion rules

2.P3 [PREF] *Successive fifths on the downbeat are only allowed when they are separated by a third on the upbeat.*

"A half note may, for the sake of the harmonic triad, occasionally make a succession of two parallel fifth acceptable - which can be effected by the skip of a third." Mann [28, p.86]

Fux didn't speak about prohibiting two parallel (i.e. consecutive) fifths in the second species for two voices. That being said, it is indeed prohibited in three parts composition as you cannot have two successive perfect consonances (see rule **1.P4**). We thus have to relax constraint **1.P4** in order to accept two successive consonances, when the two successive fifths flank a third. And since the rule on successive perfect consonances is actually a preference, this means that the cost of successive perfect consonances is not applied if those two consonances are fifths and there is a third in between.

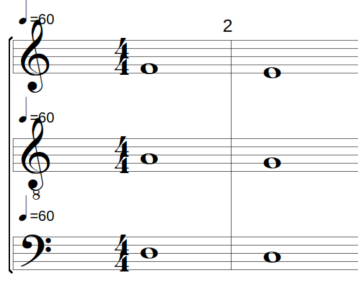


Figure 3.3: Successive fifths - prohibited



Figure 3.4: Successive fifths separated by a third - valid

3.3.2 Formalisation into constraints

Harmonic rules

2.H4 *Major thirds are now allowed in the last chord.*

No need to add a new constraint as this rule is already covered by rules **1.H2** and **1.H3** and **1.H8**.

2.H5 *The half notes must be coherent with respect to the whole notes.*

No need to add a new constraint as this is not an actual rule.

Melodic rules

2.M2 • *It is allowed to ligate the fourth-to-last with the third-to-last or to ligature the third-to-last with the second-to-last.*

This is a relaxation of the two-voice rule **2.M2** "two consecutive notes cannot be the same".

The reason why this rule has been implemented as a constraint relaxation instead than as a cost is because Fux does not say that ligaturing is bad, he just presents it as a new option offered by the three-part composition.

$$\begin{aligned}
 &\forall j \in [1, m), \quad j \neq m - 2 : \\
 &((N[2, j - 1] \neq N[0, j]) \wedge (N[0, j] \neq N[2, j])) \\
 &\wedge \\
 &((N[2, m - 3] \neq N[0, m - 2]) \vee (N[0, m - 2] \neq N[2, m - 2]))
 \end{aligned} \tag{3.12}$$

The first line prohibits ligatures except in the positions where they are allowed, and the second line states that only *one* ligature can occur.

Motion rules

2.P3 [PREF] *Successive fifths on the downbeat are only allowed when they are separated by a third on the upbeat.*

This rule is a relaxation of the cost **1.P4** defined above, and is thus rewritten to correspond to a special case that occurs with the second species.

In the following equation, only p_1 must be a second species counterpoint, p_2 can

be any species.

$$\forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ where } p_1 \neq p_2, \quad \forall j \in [0, m-1] :$$

$$\mathcal{C}_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[0, j] \notin Cons_p) \vee (H(p_1, p_2)[0, j+1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[0, j] = 7) \wedge (H(p_1, p_2)[0, j+1] = 7) \\ & \quad \wedge (H(p_1, p_2)[2, j] = 3) \vee (H(p_1, p_2)[2, j] = 4) \\ 2 & \text{otherwise} \end{cases} \quad (3.13)$$

The meaning of this equation is that $\mathcal{C}_{succ_p_cons}$ is equal to zero if one of the two considered consonances is not perfect (because then we do not have two successive perfect consonances), or if we have two successive fifths with a third in between. Otherwise (when we have perfect consonances), the cost must be set.

3.4 Third species

The third species consists only of quarter notes. It introduces even more dissonance than the second species and opens the space for more variation.

All the rules in this section apply when $species(p) = 3$. Note that the rules for the first species also apply to counterpoints of the third species.

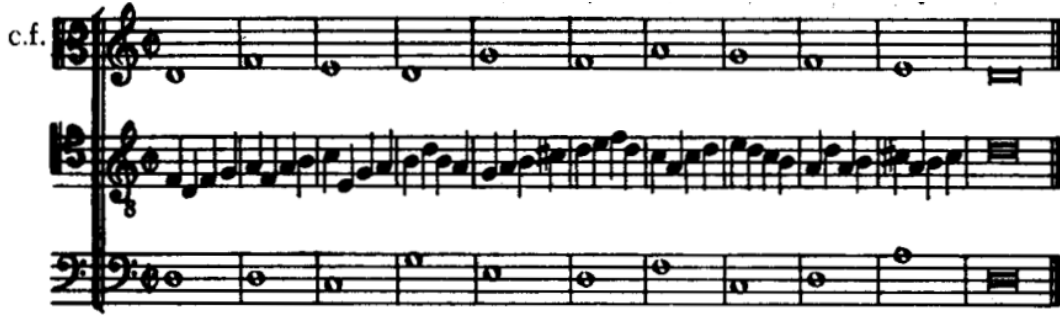


Figure 3.5: Example of a third species counterpoint in three-part composition

3.4.1 Formalisation in English

Harmonic rules

3.H5 *The quarter notes must be coherent with respect to the whole notes.*

"The quarters have to concur with the whole notes of the other voices."
Mann [28, p.91]

When Fux uses the word "concur", he most likely means "are related" and not "are consonant". This is the same as for the rule **2.H5** in the second species, where the half notes had to be *concordant* with the *cantus firmus*. This is not a rule in itself, Fux is just saying that some rules should be followed (i.e. the other constraints).

3.H6 [PREF] *If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.*

"Take care whenever you cannot use the harmonic triad on the first quarter occurring on the upbeat, to use it on the second or third quarters." Mann [28, p.91]

This rule is quite clear and speaks for itself.

Melodic rules

Fux introduces no new melodic constraints for the third species.

Motion rules

Fux introduces no new motion constraints for the third species.

3.4.2 Formalisation into constraints

Harmonic rules

3.H5 *The quarter notes must be coherent with respect to the whole notes.*

As has been discussed in the previous section, there is no constraint to add for this rule, which isn't really a rule.

3.H6 [PREF] *If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.*

This rule is quite clear and speaks for itself. Since this is not a strict rule but an advice, it was treated as a cost.

$$\begin{aligned} \forall j \in [0, m): \\ (H[1, j] \notin \text{Cons}_{h_triad}) \wedge (H[2, j] \notin \text{Cons}_{h_triad}) \\ \iff C_{\text{harmonic-triad-3rd-species}}[j] = 1 \end{aligned} \quad (3.14)$$

3.5 Fourth species

As a reminder, the fourth species consists of syncopations. Each note is played on the upbeat and is ligated to the next note on the downbeat.

All the rules in this section apply when $\text{species}(p) = 4$. Note that the rules for the first species also apply to counterpoints of the third species.



Figure 3.6: Example of a fourth species counterpoint in three-part composition

3.5.1 Formalisation in English

Structural constraints

4.S1 *The fourth species is staggered by two beats.*

"The ligature is nothing but a delaying of the note following." Mann [28, p.95]

Fux here insists on a fact that we have already discussed in 2.3.2. The fourth species behaves as if its upbeats were the downbeats and its downbeats were the upbeats of the previous measure.

4.S2 *All parts can become the lowest stratum somewhere in the composition.*

"The tenor takes the place of the bass - a thing that not only the tenor may do, but also the alto and even the soprano." Mann [28, p.100]

Fux speaks here about our concept of strata. The tenor can become the lowest stratum, just like the alto and the soprano may do. This is a fundamental concept of the generalization of Fux counterpoint to three voices, and has already been extensively discussed before (see Section 2.1).

Harmonic rules

4.H5 [PREF] *Imperfect consonances are preferred over fifth intervals, which in turn are preferred over octaves.*

"The fifth is a perfect consonance, the octave a more perfect one, and the unison the most perfect of all; and the more perfect a consonance, the less harmony it has." Mann [28, p.97]

This rule is as clear as it gets.

Melodic rules

Fux introduces no new melodic constraints for the fourth species.

Motion rules

4.P3 [PREF] *Successive fifths are allowed when using ligatures.*

"[It would be impossible to remove] the ligatures because of another consideration, the immediate succession of several fifths." Mann [28, p.95]

By saying that it exists a rule that prohibits the succession of fifths (which is actually just a particular case of rule **1.P4**, stating that you cannot have two successive perfect consonances) when there is no ligature, Fux is telling us in an indirect way that this rule is not applicable when there are ligatures. He further complements by saying "there is great power in ligatures - the ability to avoid or improve incorrect passages".

The conclusion is that successive fifths are allowed in the fourth species.

4.P4 [PREF] *Resolving to a fifth is preferred over resolving to an octave.*

"A dissonance that resolves to a fifth is more acceptable than a dissonance that resolves to an octave." Mann [28, p.98]

This rule could not be clearer.

4.P5 *Stationary movement in the bass implies dissonance in the fourth species part.*

"If I said that the first note of the ligature must always be consonant, that applies only to the instances in which the lower voice moves from bar to bar, but not the instances in which the bass remains on a pedal point, that is, in the same position. In such a case a ligature involving only dissonances is not only correct but even very beautiful." Mann [28, p.98]

The rule evoked here cancels the rule **4.H1** from two-part composition that states that all notes should be consonant. From now on, if the lowest stratum has a stationary movement, the corresponding delayed note in the fourth species must be a dissonance, instead of a consonance.

4.P6 *A note provoking a hidden fifth gets replaced by a rest.*

"Here a hidden succession of fifths occurs, which is easily perceptible to the ear and should be avoided in three part composition. This may be managed by using a rest." Mann [28, p.98]

Fux's uses the term 'hidden succession of fifths' without any prior definition. It is therefore difficult to be sure of what he meant, since the traditional terms for such progressions are as vague and variable as the traditional rules that govern them. Nevertheless, many authors seem to agree on the following definition of a 'hidden interval': a hidden fifth or hidden octave is when you approach a perfect fifth or perfect octave by direct motion. [35, p.31]. Looking closely at figures 137, 151 and 152 of the English version of *Gradus ad Parnassum*, this definition is consistent with Fux's interpretation.

The point of the rule then is: if a solution leads to a hidden fifth, then the note that provokes the fifth is replaced by a rest. This rule is an *a posteriori* rule: it applies after the solution has been found. The current rule thus complements the rule **4.P3** (about successive fifths in fourth species) and the rule **1.P1** (about direct moves to perfect consonances) without changing them.

See figures 3.7 and 3.8:



Figure 3.7: Invalid solution featuring hidden fifths



Figure 3.8: Valid solution replacing the hidden fifth by a rest.

3.5.2 Formalisation into constraints

Structural constraints

4.S1 *The fourth species is staggered by two beats.*

There is no constraint to add since this rule is the very definition of the fourth species.

4.S2 *All parts can become the lowest stratum somewhere in the composition.*

Again, there is no constraint to add since this rule is already covered by the concept of lowest stratum.

Harmonic rules

4.H5 [PREF] *Imperfect consonances are preferred over fifth intervals, which in turn are preferred over octaves.*

This rule is almost covered by the existing costs (see rule **1.H6**), as a perfect consonance has a higher cost than an imperfect consonance. But Fux says not only that imperfect consonance should be preferred over perfect ones, he says that fifths should be preferred over octaves. This precision in the rule (fifth is better than octave) could be solved by either putting a higher cost to octaves and lower one to fifths, or to put the cost for fifth before the cost for octaves in the lexicographical array of costs, but this is discussed in the parts about costs (see 4.2).

Motion rules

In this subsection, the correct notation is used for the species (see 2.3.2). So $X[0,0]$ actually means $X[0,0]$, and not $X[2,0]$ as in other parts.

4.P3 [PREF] *Successive fifths are allowed when using ligatures.*

The point of this rule is that Fux introduces an exception to **1.P4**: successive fifths are allowed in the fourth species.

We shall then amend the rule **1.P4** (don't forget that it is a cost) to allow successive fifths in any case, and rewrite it as:

$$\begin{aligned} &\forall p_1, p_2 \in \{cf, cp_1, cp_2\}, \quad \text{with } p_1 \neq p_2, \quad \forall j \in [0, m-1]: \\ &\mathcal{C}_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[2, j] \notin Cons_p) \vee (H(p_1, p_2)[2, j+1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[2, j] = 7) \wedge (H(p_1, p_2)[2, j+1] = 7) \\ 2 & \text{otherwise} \end{cases} \end{aligned} \quad (3.15)$$

The meaning of this equation is that the cost is set to zero if the two consecutive intervals are not perfect consonances, or if the consecutive intervals are both fifths. Otherwise, the cost must be set.

4.P4 [PREF] *Resolving to a fifth is preferred over resolving to an octave.*

This is already covered by the rule **4.H5** (prefer fifths over octaves), since preferring fifths over octaves in *all* cases implies preferring to resolve to a fifth rather than to an octave.

4.P5 *Stationary movement in the bass implies dissonance in the fourth species part.*

$$\begin{aligned} &\forall j \in [0, m-1]: \\ &M(a)[0, j] \neq 0 \iff H[2, j] \in Cons \\ &M(a)[0, j] = 0 \iff H[2, j] \in Dis \end{aligned} \quad (3.16)$$

4.P6 *A note provoking a hidden fifth gets replaced by a rest.*

$$\begin{aligned} &\forall j \in [1, m-1]: \\ &H[0, j] = 7 \wedge P[0, j] = 2 \iff N[0, j-1] = \emptyset \end{aligned} \quad (3.17)$$

This rule is very special because it applies after the search, not during the search. More precisely, after the search is started and a result is found, only then does this rule begin to apply. To understand why, remember that the suppressed note is suppressed because it causes a hidden fifth. But the only way to have a hidden fifth is to have an existing note, you cannot have a hidden fifth to a non-existing note. So we don't delete the note during the search, because then we would also delete the hidden fifth (because you can't have a hidden fifth without a note) and so we would change the

solution. You have to find a solution first, and then remove the possible note that causes a hidden fifth.

This is actually a very interesting property, because here we have a note that is not played and yet has an effect on the composition.

3.6 Fifth species

As a reminder, the fifth species is a mix of all previously mentioned species, which means that it combines the rules from all previous species.



Figure 3.9: Example of a fifth species counterpoint in three-part composition

No additional rules (be they harmonic rules, melodic rules or motion rules) were observed by Fux when writing a three-part counterpoint with the fifth species.

However, in order to have some rhythmic variety when composing with two fifth species counterpoints, it was decided to impose a rule that Fux never mentioned. We know that the fifth is just all the other species combined, and the way the solver understands this is that it considers each note of the composition to belong to a particular species. This is represented as $S[i, j]$, where $S[i, j] = x$ means that the i -th note of the j -th measure belongs to the x -th species. So if $S(cp_1)[2, 3] = 3$, then the second note of the third measure of the first counterpoint is constrained by the constraints of the third species. From the solver's point of view, S is just an array like all the others, but it branches on it during the search to guarantee that the composition of the fifth species is composed of as many species as possible.

This gives us a metric for similarity, since two fifth-species counterpoints that have the same S array will be rhythmically redundant. To tackle this problem, we force the solver to find a solution where $S(cp_1)$ must be at least half as different as $S(cp_2)$.

As can be seen in Figure 3.10, the first composition is indeed rhythmically redundant, since both fifth-species counterpoints have (almost) the same rhythm. The second is much less redundant, since it was required that the counterpoints use different species for at least 50% of the beats.

Redundant solution with the same rhythm in both counterpoints:



Less redundant solution with more variety in the counterpoints:



Figure 3.10: Two compositions of two fifth-species counterpoint, one redundant and the other not

$$species(cp_1) = species(cp_2) = 5 \iff \sum_{i=0}^3 \sum_{j=0}^{m-1} (S(cp_1)[i, j] = S(cp_2)[i, j]) < \frac{s_m}{2} \quad (3.18)$$

This equation is only true if both counterpoints are of the fifth species, and its meaning is that the sum of the times $S(cp_1) = S(cp_2)$ must be less than half the number of notes in the composition.

3.7 Writing a three-part composition using various species

In *Gradus ad Parnassum*, Fux almost always writes his counterpoints using a combination of the first species and another species, i.e. he makes the following combinations: 1+1, 1+2, 1+3, 1+4, and 1+5. Only sometimes does he make other combinations, either with two counterpoints of the same type (5+5) or with two different types (2+3).

When creating combinations of different species, Fux doesn't give any specific rules for combining them. It seems that the different species interact with each other as they would do with the first species, taking into account only the first beat of the measure. For example, if we compute a first counterpoint belonging to the 3rd species and a second counterpoint belonging to the second species, the rules that apply between the two counterpoints will be set between all the beats of the first counterpoint and the first beat of the second counterpoint and between all the beats of the second counterpoint and the first beat of the first counterpoint. This corresponds to what we would have done if we had applied the rules between a counterpoint and the *cantus firmus*, where the rules apply between all beats of the *cantus firmus* and the first beat of the counterpoint.

Below is a brief summary of the rules that apply to each species:

- **First species** — rules of the first species,
- **Second species** — rules of the first species + rules of the second species,
- **Third species** — rules of the first species + rules of the third species,
- **Fourth species** — rules of the first species + rules of the fourth species,
- **Fifth species** — rules of all species, selected according to the S array.

This means that in a composition with two counterpoints, one of the 2nd species and one of the 3rd species, the first counterpoint will follow the rules of the 1st and the 2nd species, and the second counterpoint will follow the rules of the 1st and the 3rd species.

Chapter 4

Solution search for three-part counterpoint

Three-part composition is much richer than two-part composition. In fact, it offers many more possibilities than the two-part composition. In constraint solver terms, this means that the search space is much larger, which means two things: the computational complexity of finding a solution is greater, and there are many more possible solutions. The first difficulty is to find a valid solution in this large search space, and the second difficulty is to find the solution that best respects the preferences expressed by Fux.

In this chapter, we first discuss the computational aspect and then turn to the preference management aspect. The computational complexity section covers: the search algorithm used, the heuristics implemented, and some of the factors that influence the speed of finding a solution; whereas the preference management section covers: discussing how to translate Fux's preferences into solver costs, and comparing the different methods that exist for doing so.

4.1 Dealing with the higher computational complexity

Composing a three-part counterpoint is more computationally demanding than composing a two-part counterpoint: the search space has been greatly expanded by adding a whole new set of variables, and the time it takes to find a solution may be too high if you do not think about optimising the search. In addition, adding a third voice to a composition does not bring many new constraints (which would help to discard some potential solutions faster), but it does bring many preferences, which in constraint programming are translated into costs. The solver must be efficient in finding the solution that has the lowest cost (i.e. that is best in terms of musicality).

4.1.1 Using Branch-And-Bound as a search algorithm

To cope with the increased complexity brought about by the three-part composition, it was decided to switch from the Depth First Search algorithm (used in the previous version from FuxCP) to a more efficient Branch and Bound (BAB). This allows us to handle costs properly and to find faster solutions. Moreover, the BAB algorithm can also produce non-optimal results, which is very valuable since finding the best overall solution can be time-consuming. When starting the search for a solution, it is now possible to ask for the next solution (i.e. a better solution than the one found previously, and if none was found previously, then just any valid solution), or for the best solution. In the latter case, the solver will continue to search until it finds the best solution or until it is stopped, returning a better solution each time it finds one.

4.1.2 Heuristics

Main heuristics

When it comes to finding a solution, we obviously need some heuristics to guide the search, because there are so many different possibilities for a three-part composition. To know which heuristics to use, simply think about the most important variable to fix first. In case it is not clear enough, the key to writing counterpoint for many voices is to know what the bass, i.e. the lowest stratum, is doing. This is true whether the composer is a human or a constraint solver. The first heuristic thus follows naturally, and it is: *branch on the lowest stratum array, take the highest constrained variable yet, and try with any value less than the median*. This value branching is used because the possible values for the lowest stratum are those of the notes of the three parts. Since the lowest stratum is designed to always be equal to the lowest sounding note, it doesn't make sense to try to give it a high value. Therefore, lower values are chosen.

The other central heuristic instructs the solver to *branch on the variables of the N arrays (containing the pitches of the voices), choosing as a priority the variables whose domain is small, and try with random values*. This choice is motivated by the fact that in the case of a highly constrained counterpoint (5th species) and a weakly constrained counterpoint (1st species), the counterpoint should not only seek to improve the highly constrained counterpoint, but also the weakly constrained counterpoint. This is why the heuristic chosen is based on the size of the domain and not on the level of constraint. Choosing a random value ensures maximum variety in the final composition and quickly leads to a solution with a variety of notes.

Additional heuristics

A first additional heuristic is to *branch on the array representing the species contained in the fifth species, to ensure a varied composition*. If we are dealing with a counterpoint of the fourth or fifth species, we also branch on the "no ligature cost", so that the solver explores solutions in which the notes are linked, since this is the very nature of the fourth species.

The rule **1.P5** states that the voices should start distant, and as suggested in the section on rules, this should be implemented in a heuristic. However, when we implemented the heuristic that all voices should start distant from the lowest one, we did not see any improvement, neither in search speed nor in solution quality. In fact, it sometimes slowed down the search, so this heuristic was dropped. Furthermore, Fux's advice that the voices should start far apart in order to progress in the opposite direction is only true if the bottom layer moves up. If the bottom stratum moves down, the top strata should move up, so starting far apart becomes a compositional disadvantage in this case (as the voices are limited by their range).

4.1.3 Time to find a solution

In general, the solver is able to find a valid solution fairly quickly (in the order of a second). However, in some cases it is more difficult to find a valid solution and the solver may take a little longer to find one. These cases are generally related to: the species of the counterpoints, and the spacing between the voice ranges of the counterpoints. It is difficult to know which combination of ranges and species will lead to a fast solution, but some general tendencies can be observed:

- **Species of the counterpoint** – The more complex the species of the counterpoints are (think of the 3rd and 5th species, for example), the longer it will

probably take the solver to find a solution. The reason is quite obvious: the more complex the species, the more variables the solver has to play with and the larger the search space.

- **Distance between the voice ranges** – The closer the ranges are, the greater the probability that the solver will take a long time to find a solution. For example, finding a solution by giving the two counterpoints the same range as the *cantus firmus* is more time-consuming than choosing distant ranges. This is because the voices cannot form a unison, and the possibilities for each voice are therefore smaller when their ranges are close together.

There are also cases where the exact combination of two given vocal ranges for two given species makes the search take some time, but by changing the vocal range a little, the solver quite surprisingly finds a solution immediately. It is still unclear why this happens. Our best guess is that there are some particular combinations of *cantus firmus*, voice ranges and species for which the solver has difficulty finding a solution, because of the specific search space of this setting.

It is worth noting that the solver's greatest difficulty (in all cases) is finding a valid solution. Once a valid solution has been found, the solver quickly finds a whole series of solutions, each one better than the previous one (until, of course, it is difficult to find the best solution: then the solver starts taking some time again). The distance between the best solution and the solution found by the solver before reaching a plateau when searching is difficult to estimate. However, it should be remembered that the tool is intended to be used iteratively (the user tries a combination, changes the cost, tries again, etc). In the end, it is human preferences that matter most, and there is no guarantee that the best solution for the user will be the one that the solver thinks is best. This means that the direction the solver takes is more important than the best solution in itself.

4.2 Designing the costs of the solver to be as faithful as possible to the preferences of Fux

A constraint solver cannot determine whether one valid solution is better than another valid solution by itself, and yet we need it to produce the best possible musical solution. We can already say that one solution is better than another if it respects Fux's preferences (see 1.2) more. In order for the solver to understand these preferences and to be able to distinguish the musical quality of two solutions, we give costs to it. The lower the cost, the higher the preference.

Knowing that we are looking for the solution whose cost must be as low as possible, the question arises: how can we calculate the cost in order to best reflect the preferences expressed in *Gradus ad Parnassum*?

The way to translate each preference into a corresponding cost has of course been formalised in the previous sections, but that's not the crux of the matter. The question we face here is: what is the best way to combine all these individual costs to get the most accurate result in terms of what Fux is trying to convey in terms of preferences?

Three main ways of combining the costs have been identified. These are, in order from the simplest to the most complex:

1. a linear combination of all costs,
2. computing the maximum over all costs and minimising it,
3. a lexicographic order search,

We first describe each of these techniques and their respective advantages, and then compare them (and the results they produce).

Throughout this section we talk about some specific costs. All these costs are listed in 4.2.3 with a short explanation. Some more details can be found in Appendix B.3. For the full description of a specific cost, please refer to its corresponding rule.

4.2.1 Linear combination

The first method of calculating our costs is a linear combination. This is the technique used in the previous version of FuxCP. More precisely, it uses a linear combination in which all the weights are equal to one.

Here is a more detailed explanation: there exists a total cost, τ , which is equal to the sum of all individual costs, $\mathcal{C}$. The next step is to minimise τ . Each $\mathcal{C}_i$ is usually itself a sum of sub-costs. Take, for example, the cost of using harmonic fifths, $\mathcal{C}_{fifths} = \sum_j \mathcal{C}_{fifths}[j]$. This cost is the sum of all sub-costs of having harmonic fifths (one per measure). By default, having a harmonic fifth in a measure has a sub-cost of 2. This default values can be changed by the user to be set somewhere on a scale that ranges from 0 to $64m$.

As mentioned at the beginning of this subsection, this procedure can be understood as a linear combination with weights of one only. However, since the cost factors are given different values according to the user's choices, this method is actually more like a regular linear combination, except that the costs are not multiplied by any weights, but the costs are themselves made larger or smaller before the linear combination is calculated.

The linear combination has two major advantages: ease of implementation and high comprehensibility. However, it has a major drawback: since the total cost τ that gets minimised is the sum of all costs $\mathcal{C}$, the best solution might be a solution where one cost is absolutely huge and all the others are small. This might not be a problem if the outstanding cost is not really relevant, but if it is the cost of not using a harmonic triad, it goes completely against the preferences that Fux conveys in his work, making the solution inappropriate. A representation of this situation can be found in the Figure 4.1.

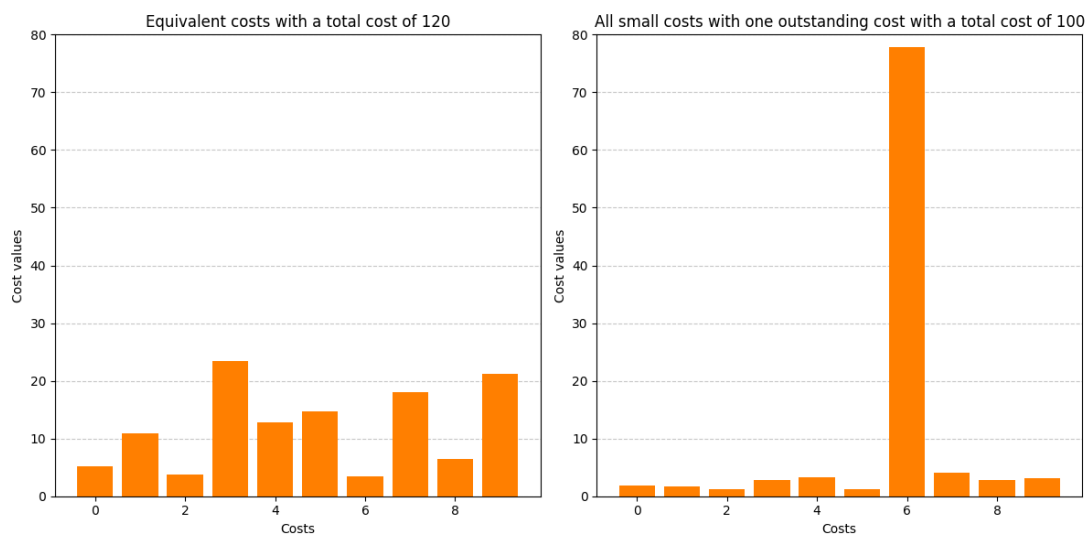


Figure 4.1: Example of a situation where a solution with an outstanding cost is preferred to a solution with equivalent low costs when using a linear combination

Another disadvantage of linear combination is that the result is quite unpredictable: changing the value of the cost may or may not make a difference, and you may need to set huge values to see a real effect. For example, if a composer really wants oblique motion, they may be forced to set the cost of the other types of motion to a huge value, or they may not see the difference between the default solution and their personalised solution. This is due to the fact that all the costs are mixed together and form an indistinguishable soup that the solver considers as a whole, and a small increase in the cost of the direct and contrary motions is very likely to be absorbed into this soup without any change being noticed.

4.2.2 Minimising the maxima

In order to overcome the problem of outstanding costs that we encountered when considering the linear combination solution, one could consider a technique that specifically addresses these outstanding costs: namely, minimising the maximum cost. For example, τ the total cost, could be set equal to the current largest cost. By doing this, the solver would try to find a solution where the focus is on the worst cost and try to reduce it before trying to reduce the other costs.

The problem with this method arises when one cost is significantly higher than the others because it has been defined that way. Let's take the example of a composer wanting as many oblique motions as possible. They will set the cost for direct motion and contrary motion to the highest possible cost and start the search. It is not possible to have only oblique motions, since it would mean that all motions are direct (and thus not oblique). As a result, there will always be either direct or contrary motions, and since the cost for them has been set very high, it would be impossible for the solver to converge to a good solution. The solver will be more focused on lowering costs that are high by design, than to lower all the other costs. This means that those high costs are limiting the search of the solution, creating a bottleneck effect. Once the solver has reached the best potential value of the worst cost, it cannot continue to find better solutions. This situation is illustrated in Figure 4.2.

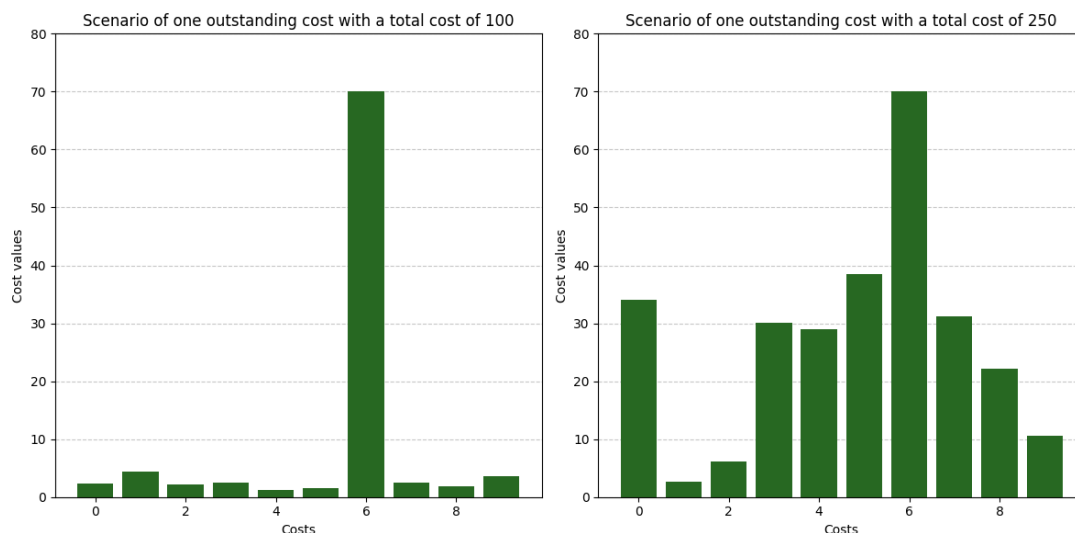


Figure 4.2: Example of two situations where a cost causes a bottleneck in the search because the solver cannot distinguish between left and right situations. The solver will blindly choose one of the two solutions, even if the solution on the left is obviously better.

Furthermore, even when considering a less extreme case (e.g. the default setting), this method requires a normalisation of the costs: there are $3 \times (m - 2)$ sub-costs for the variety cost but only m sub-costs for the octave cost. This means that without normalisation, the variety cost will be on average three times larger than the octave cost, which means that the solver will put three times more effort into minimising the variety cost than the octave cost, which is unfair and unpractical.

4.2.3 Lexicographic order

The third way of dealing with the costs is to arrange them in an array and then perform a lexicographic minimisation. In other words, the costs are arranged by order of importance: from most important to least important. The most important cost to minimise is placed first in this array, and the solver only tries to minimise the other costs if the first cost remained the same or decreased. This method makes a lot of sense when you think about the rules that emanate of *Gradus ad Parnassum*. For example, Fux says that perfect consonance can be achieved by direct motion if there is no other possibility. This means that, all other things being equal, we would prefer not to achieve perfect consonance by direct motion, but that between a bad solution¹ in which perfect consonance is not achieved by direct motion, and a good solution in which perfect consonance is achieved by direct motion, we would choose the good solution.

Some costs are also more important than others in absolute terms. For example, when Fux says that an imperfect consonance is preferred to a fifth, which in turn is preferred to an octave. This amounts to lexicographically ranking the cost of using an octave first (because we really don't want octaves), and then the cost of using a fifth (and there is no cost of using an imperfect consonance, since Fux indicates that this is preferable).

$$\tau = [\underbrace{C_{\text{octaves}}}_{\text{minimise this first}}, C_{\text{fifths}}] \quad (4.1)$$

A second example, which ties in particularly well with the first, is that Fux tells us that the harmonic triad must be used in every measure unless a rule forbids it. In saying this, he places the preference for the harmonic triad above all other preferences, because the only reason that can prevent the use of a harmonic triad is a fixed constraint (and not a preference). You'll notice that the harmonic triad consists of a fifth (which is a perfect consonant), so Fux is telling us that we'd rather use a fifth in a harmonic triad than an imperfect consonant outside a harmonic triad. The lexicographic order search is the only one that allows this kind of concept to be taken into account, because in a linear combination these two preferences would be mutually "exclusive"²: the first preference would add a cost where the second preference would not, and the second preference would add a cost where the first would not.

$$\tau = [\underbrace{C_{\text{harmonic_triad}}}_{\text{minimise this first}}, \underbrace{C_{\text{octaves}}}_{\text{and start minimising this only if it is not possible anymore to minimise the harmonic triad cost}}, C_{\text{fifths}}] \quad (4.2)$$

This way we can keep integrating the different costs until we get a full array τ with all the costs ordered in a lexicographic way.

Of course, it is not always as simple as in the examples above, because it is not always easy to determine which cost has priority over which other. Sometimes Fux is very clear about it (e.g. for the harmonic triad cost, which Fux says has priority

¹A solution in which almost all other costs are high.

²In the sense that their effects would work against each other.

over everything else), and sometimes he isn't (do we prefer no off-key notes, or as much variety as possible?). This is a drawback of this method, because we have to hierarchise the costs, even if the choice is difficult. What's more, once the costs are ranked, their order becomes absolute and the solver loses some of its flexibility.

Knowing this, we came up with a suggested order that should be as close as possible to Fux's preferred order (or at least what we understood him to convey as his preferred order in *Gradus ad Parnassum*). This order should of course be changeable at the composer's discretion³. The default order we have agreed upon is as follows. When a cost is followed by a number in brackets, this means that it only applies if the corresponding species is used.

- | | |
|--|--|
| 1. $C_{\text{no_syncope}}$ ⁴ [4, 5] | 8. $C_{\text{borrowed_notes}}$ ⁶ |
| 2. $C_{\text{successive_p_cons}}$ | 9. C_{variety} |
| 3. $C_{\text{harmonic_triad}}$ | 10. $C_{\text{m2_eq_zero}}$ ⁷ [3, 4, 5] |
| 4. $C_{\text{harmonic_triad_3rd_species}}$ [3] | 11. $C_{\text{not_cambiata}}$ ⁸ [3, 5] |
| 5. C_{octaves} | 12. C_{motions} |
| 6. $C_{\text{penult_thesis_is_fifth}}$ ⁵ [2] | 13. $C_{\text{m_degrees}}$ ⁹ |
| 7. C_{fifths} | 14. $C_{\text{direct_move_to_p_cons}}$ |

Some remarks on the proposed order:

- Two costs come even before the harmonic triad cost: the $C_{\text{no_syncope}}$ cost and the $C_{\text{successive_p_cons}}$ cost. Regarding the $C_{\text{no_syncope}}$ cost: this cost is at the heart of the fourth species, and a fourth species counterpoint without syncopations is not really a fourth species counterpoint. This is why syncopation is considered even more important than the harmonic triad. And concerning the $C_{\text{successive_p_cons}}$ cost: when Fux expresses his preference for the harmonic triad, he says that the harmonic triad should always be used, excepted if there are two successive perfect consonances (see rule 1.H8). This makes successive perfect consonances even more important to avoid than not having a harmonic triad.
- The cost of $C_{\text{penult_thesis_is_fifth}}$ comes before the cost of C_{fifths} , as it is an exception to the latter (similar to the interaction explained above in the section between $C_{\text{harmonic_triad}}$ and C_{fifths}).
- $C_{\text{off_key}}$ was added to its ranking because it is actually an absolute rule not to use off-key notes, but Fux does use some, and so it was decided to put this cost after the very important costs to allow off-key notes to happen.
- The costs C_{variety} , C_{motions} and $C_{\text{m_degrees}}$ were ranked in order from least to most restrictive. First we say that we would prefer the note to change as much as

³Please have a look at Appendix B to see how the composer can personalise the order.

⁴The cost of not using a syncope.

⁵A specific cost for the second species, which applies when a penultimate thesis note does not make a fifth interval with the lowest stratum.

⁶The cost of using sharps or flats.

⁷The cost of having the same note in the downbeat and the upbeat.

⁸The cost of not using a *cambiata*. The *cambiata* can be characterised by the following scheme: consonance - dissonance - consonance.

⁹The cost of using big or small melodic intervals.

possible (with the variety cost), then we indicate our preference for the direction (with the motion cost), and finally we indicate our preference for the size of the motion (with the melodic interval cost). This gives the solver as much flexibility as possible. The other way round would have been more restrictive, since the solver would have minimised the melodic intervals first, setting them all to one, which doesn't leave much room for the motion cost to have an effect, and forcing the variety cost to be high in any case, as with small intervals the melody tends to vary only a little.

- $\mathcal{C}_{m2_eq_zero}$ and $\mathcal{C}_{m_degrees}$ were classified right after the variety cost since they are an in-measure variation of the variety preference.

NB Using the lexicographic order does not *not* mean that the last costs are not taken into account, they are minimised *too* by the solver. It just means that if the solver has to choose between minimising one cost or another, it will minimise the first one in the lexicographic order.

4.2.4 Comparison between the three types of costs.

We have now discussed the advantages and disadvantages of each of the three methods. These are all listed in Table 4.1. As you can see, no one method is definitively better than another, and the only way to know which method is best in practice is to test them in practice to find out which of the methods gives the best results.

Table 4.1: Comparison of the three methods — a green cell can be considered a good feature, a red cell can be considered a bad feature, and an orange cell is a special case

Criteria	Linear combination	Minimising the maximum	Lexicographic order
Outstanding costs	Yes	No	Only for minor costs
Sensitivity ¹⁰	No	Some	Yes
One cost might be a bottleneck	No	Yes	No
Need to normalise costs	No	Yes	No
Possibility to ensure a preference of one cost over another	No	No	Yes
Need for hierarchisation of costs	No	No	Yes
Flexibility	Medium	High	Low

Knowing this, one could think about a combination of all the methods to get rid of their disadvantages. In fact, we could enjoy the advantages of all the methods by combining them and cleverly designing a lexicographic order search in which the cost is a linear combination of a maximum minimisation.

4.3 Combining the three types of costs

To experiment which method gives the best results, we will follow this plan: first compare the result of a linear combination and the result of a purely lexicographic order,

¹⁰In the sense that changing one cost has a big impact on the result.

using the default preference order as defined in 4.2.3. We then analyse the result by looking at what could have been done to manage costs more effectively and, where appropriate, group costs together.

These experiments are carried out using various counterpoint combinations. These setups will increase in complexity, starting with the basic case of three-part counterpoint (two counterpoints of the first species) and moving on to mixed counterpoint.

The advantage of simple species (first, second and fourth species) is that the search for a solution is much faster. In fact, the search for an optimal solution can be quite time-consuming, and this is even more the case when we are talking about complex species such as the third and the fifth, and when they are combined. This means that it is more difficult to grasp the impact of the cost method when using a setup with complex species. What's more, the vast majority of the costs are related to the interaction of the voices in the first beat of the measures: the behaviour we want to observe, i.e. the interaction between the cost method and the resulting composition, will be just as observable with complex species as with simple ones.

The musical analyses in this section are superficial and do not deal with in-depth music theory. They will consist of general surface and impression remarks. They are highly subjective and should not be taken at face value. The aim of this analysis is to provide an initial critical view of the results offered by the solver.

The selected *cantus firmi* were chosen from *Gradus ad Parnassum*.

4.3.1 Comparing the linear combination and the lexicographic order in practice

First experiment: two first-species counterpoints on two different *cantus firmi*

For this first experiment, if the search time exceeds 30 seconds, the search is stopped and the current solution is analysed.



Figure 4.3: Result 1 of the linear combination method with default costs. [Click here to listen.](#)

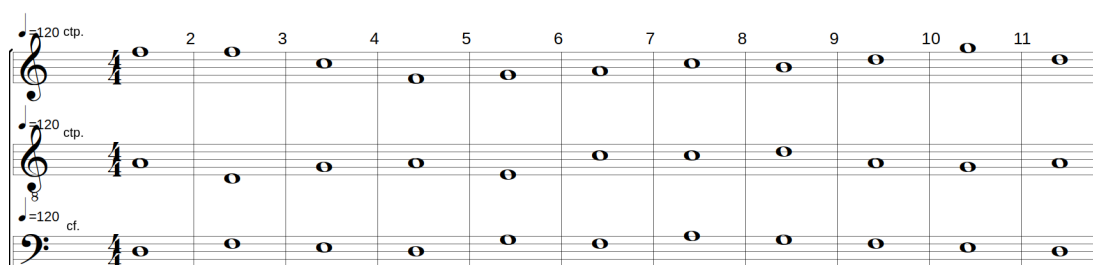


Figure 4.4: Result 1 of the lexicographic order method with default costs. [Click here to listen.](#)

Here are the first two results: 4.3 and 4.4. To the average ear, there's not much difference between these two solutions.

In more technical terms, here are the cost arrays:

- for the linear combination: [2, 20, 0, 3, 4, 6, 4, 14, 0], with a total sum of 53,
- for the lexicographical order: [$\underbrace{0}_{\text{succ_p_cons}}$, $\underbrace{14}_{\text{h_triad}}$, 0, 3, 0, 10, $\underbrace{16}_{\text{motions}}$, 14, $\underbrace{8}_{\text{direct_m_p_cons}}$],
with a total sum of 65.

As we can see, the difference between the two is exactly what we expected: high importance costs are prioritised and low importance costs are neglected in the lexicographic order, while the linear combination tries to minimise everything. A good example of this is the harmonic triads: they are more present in the lexicographic solution than in the linear combination (four versus one). Meanwhile, there are seven direct motions and two oblique motions in the lexicographic solution and only two direct motions in the linear combination solution¹¹.

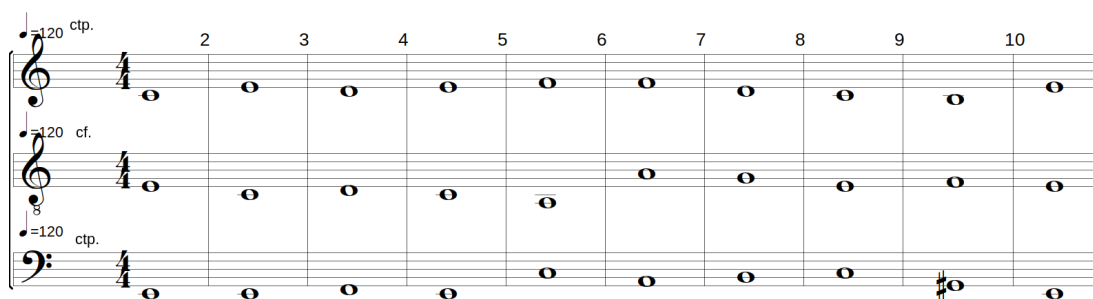


Figure 4.5: Result 2 of the linear combination method with default costs. [Click here to listen.](#)

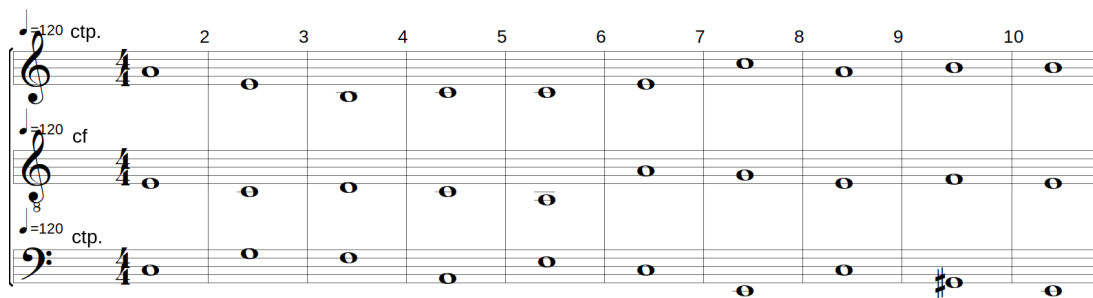


Figure 4.6: Result 2 of the lexicographic order method with default costs. [Click here to listen.](#)

Let's look at the second result (obtained with a different *cantus firmus*), featured in figures 4.5 and 4.6. Once again, the technical results are fairly as one would expect them:

- for the linear combination: [0, 20, 4, 0, 4, 12, 11, 8, 8], with a total sum of 67,
- for the lexicographical order: [0, 16, 0, 2, 4, 8, 10, $\underbrace{19}_{\text{melodic_intervals}}$, 8], with a total sum of 67.

¹¹Reminder: the motions are computed with respect to the lowest sounding note.

For the average listener, it is difficult to establish an absolute preference between these two solutions. We will take this same setting and try to mix the techniques for it in section 4.3.2.

Second experiment: one fourth-species counterpoint and one first-species counterpoint on a single *cantus firmus*

If we now go on, here is a result combining the first and the fourth species, and putting the 4th species at the bottom:

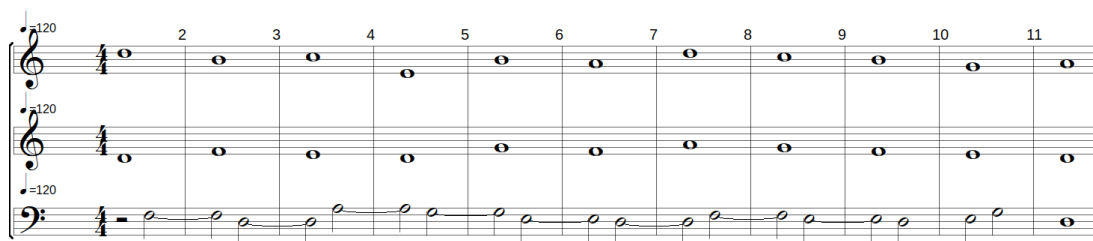


Figure 4.7: Result 3 of the linear combination method with default costs. [Click here to listen.](#)



Figure 4.8: Result 3 of the lexicographic order method with default costs. [Click here to listen.](#)

Looking at the results obtained with this setup (figures 4.7 and 4.8), we come to the following conclusion: in both cases, the melody is a little dull and lacks dynamism. There is no drastic change in quality between the solutions provided by the two search techniques. However, the solution provided by the lexicographic order is somewhat more exciting, since there is more tension in it and it uses more dissonances and resolvings than the linear combination (even if these resolvings are not the most brilliant).

We immediately notice something else with the 4th species on the bass, which is not related to the costs: there are a few dissonances on the downbeat, as the solver doesn't really take into account the harmonic interaction between the notes of the downbeat of the fourth species and the notes of the downbeat of the other species, but rather the harmonic interaction between the upbeat of the fourth species and the downbeat of the others: which leads to a few surprises, as we can see in these examples (that tension mentioned above).

As far as the costs are concerned, one thing is clear: not all the notes of the 4th species obtained with a linear combination are linked, whereas they all are in the solution of the lexicographic order. This is an obvious consequence of using a linear combination, as this technique is not able to prioritise a cost.

Third experiment: one third-species counterpoint and one second-species counterpoint on a single *cantus firmus*

Our first cross-species test will involve a counterpoint of the 2nd species and a counterpoint of the 3rd species. The *cantus firmus* used is the one proposed by Fux in an example in which he uses exactly these two species. The search was given one minute, as the complexity is getting higher than in the previous experiments.

The results are shown in the figures:



Figure 4.9: Result 4 of the linear combination method with default costs. [Click here to listen.](#)



Figure 4.10: Result 4 of the lexicographic order method with default costs. [Click here to listen.](#)

The results are strikingly similar and quite unmelodic. Let's look at these two aspects in turn.

The similarity is probably due to two things: the solver doesn't have much room for manoeuvre, since all the voices are highly constrained, so the costs don't have much of an effect in this very setup.

Concerning the lack of melodic quality: it is probably due in part to bad luck (this *cantus firmus* is perhaps particularly difficult to handle) and in part to the lack of constraints linking the upbeats of the various counterpoints. If you think about it, all the rules proposed by Fux in his chapter on three-voice composition link the beats of one voice (*all its beats*) to the first beat of the other voices. This means that there are constraints between the 2nd, 3rd and 4th beats of one voice and the other voices, but never between these 2nd, 3rd and 4th beats of one voice and the 2nd, 3rd and 4th beats of another voice, always with the 1st beat. Obviously, without rules to ensure that the notes of these beats concur¹², it is more complicated for these beats to concur. Of course, it would be wrong to say that it depends only on chance that these beats concur, because that would mean that the beats are independent. Indeed, one might think so at first, because there are no constraints directly linking them, and yet they are linked by their own connections with the first beat of the other voices. In other words, although the third beat of the first counterpoint is not directly linked to the third beat of the second counterpoint, it is indirectly linked to it through the first beat of the second counterpoint: there are constraints between this third beat of the first counterpoint and the

¹²The word 'concur' is used here in the same sense that Fux uses it: it means that the notes are somehow put in a relationship that makes them sound good together.

first beat of the second counterpoint, and there are also constraints between the first beat of the second counterpoint and the third beat of the second counterpoint. There is therefore a certain dependency and mutual influence between the 3rd beats of the two counterpoints. However, it would be interesting to see if Fux introduces any rules on this subject in his chapter on four-part composition, and if not, it would be interesting to think about what these rules might be in order to maximise the concordance between the notes in the upbeat of the different voices.

Fourth experiment: two fifth-species counterpoints

The fourth experiment is a bit special, as it features two fifth counterpoints *with the same voice range*. This is something Fux doesn't really do in *Gradus ad Parnassum*, but we thought it might be interesting to see how the search method behaves in this situation. Even though Fux doesn't write counterpoints in the same voice range, they are still realistic, for example in the case of a double violin concerto, or any other instrument that has the same voice range and that is played together.

In this experiment, it is interesting to observe how the solver manages the small margin of manoeuvre it has, given that it has to find two counterpoints in the same voice range, all when the counterpoints are forbidden to take the same value (i.e. the same pitch).

The search was run for two minutes, as the fifth species counterpoint is more complex than the others, and a little more time is needed to let the costs have an effect on the solution. The solutions are those in figures 4.11 and 4.12.

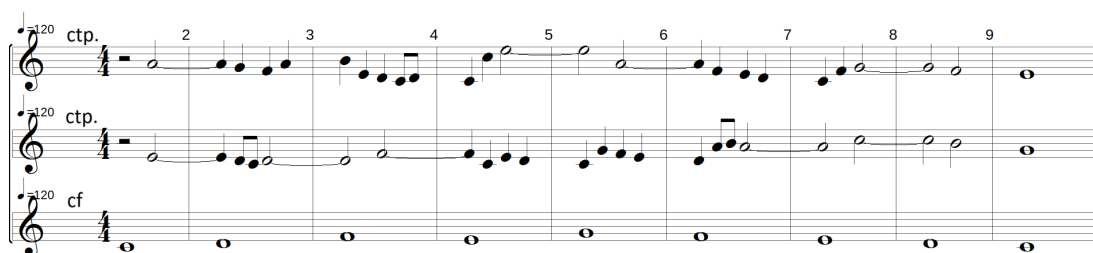


Figure 4.11: Result 5 of the linear combination method with default costs. [Click here to listen.](#)

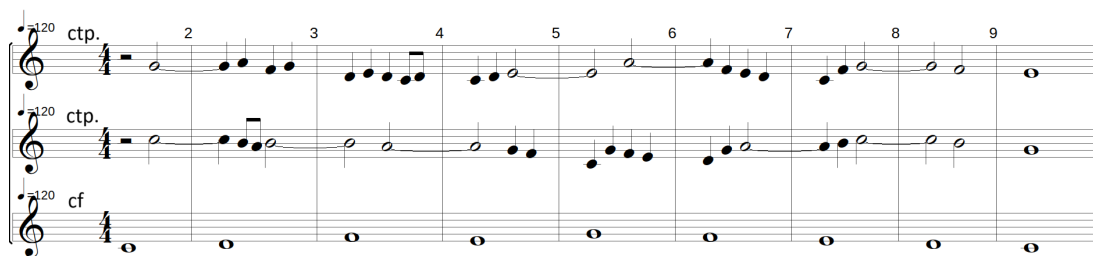


Figure 4.12: Result 5 of the lexicographic order method with default costs. [Click here to listen.](#)

Just as for the second experiment, some interesting things happen in the composition found by the lexicographic order. The intervals are more beautiful and the fact that there are dissonances that sometimes resolve gives more meaning to what's going on.

On the other hand, the fact that the penultimate note of the middle voice resolves on a G instead of a C is a bit frustrating, as the final chord feels like it is not a real ending, but this is a recurring problem in all solutions. The reason for this is probably that there is a rule (rule 4.H5) that makes the solver prefer fifths to octaves, and there is no mention from Fux of deviating from this rule for the last measure.

However, the solver did a surprisingly good job of finding passable solutions with such a small search field. The solutions obtained are far from high art, but you can see the musical intuition behind them.

4.3.2 Mixing the technique of maximum minimisation with lexicographic order

Now what happens when we start to mix the techniques and group some of the costs in the lexicographic order? At each level of the lexicographic order, we calculate the maximum of the costs at that level, which we then try to minimise. All preferences that Fux has not explicitly ordered get packed on the same level, i.e. three levels subsist: the first one, with only the cost for successive perfect consonances, the second one, with only the cost for not using a harmonic triad, and a third one, with all the other costs.

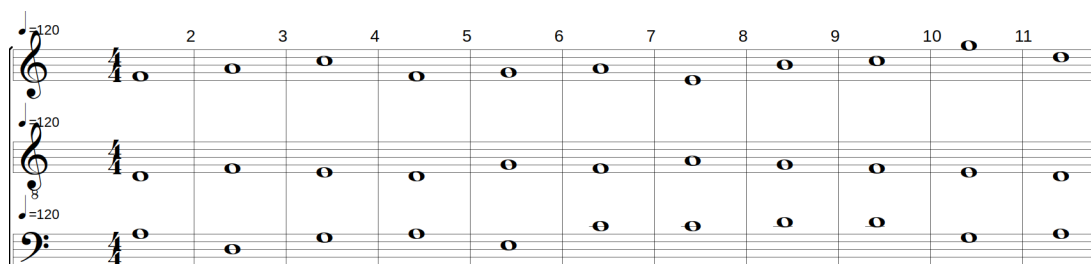


Figure 4.13: Result 1 of a mix between lexicographic and maximum minimisation method. [Click here to listen.](#)

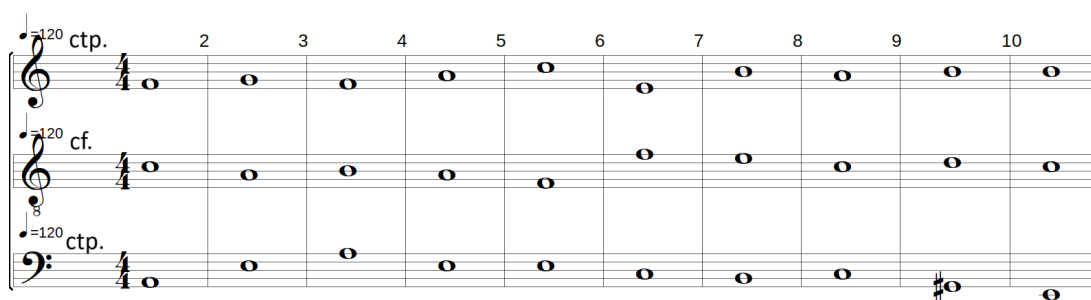


Figure 4.14: Result 2 of a mix between lexicographic and maximum minimisation method. [Click here to listen.](#)

Figures 4.13 and 4.14 show the solutions found by mixing the techniques of lexicographic order and maximum minimisation.

Both solutions handle the ending strangely, but this may be a coincidence, as there is no cost that would obviously cause such an ending.

It would be too bold to say that there is a big difference between the results obtained by this method and those obtained by the others. In other words, to the average ear, the solutions provided by this search technique are not significantly different from those provided by the other search methods.

The good news is that the result, far from being excellent, is different. This means that a composer can set up the tool as they wish and get different results from one setup to the next. They can start with the default settings and then change one parameter after another until they find what suits them best. This is really encouraging, because it means that the solution is not too limited to a few possibilities, but that once a valid solution has been found, there is still plenty of room for personalisation.

Note that the predicted bottleneck effect from Section 4.2.2 was indeed observed in both searches, as the solution stopped improving after only ten seconds of search. After these ten seconds, the solver stopped finding better solutions because the third cost level (the one whose maximum was minimised) had already reached its minimum maximum, and so the solver could not distinguish between two solutions if they had the same maximum, since this search technique doesn't allow it. Please refer to Figure 4.2 for a better understanding of the situation.

4.4 Conclusion on the search methods

As we have stressed several times in this chapter, there is no *best* way of ordering costs. Each technique has its shortcomings, and it is probably by allowing the composer to order their costs as they see fit that the tool will be able to reveal its full potential, since the choice of technique also depends on the musical effect that the composer wishes to achieve.

Nevertheless, the lexicographic method seems to be able to express more character than the cost soup of the linear combination method. The intransigent side of the lexicographic method can be adjusted by combining several costs at the same level of the lexicographic order. This combination can be achieved using the method of minimising maxima, but it should be noted that this is only possible to a certain extent if we want to avoid a cost creating a bottleneck on its own. It may also be preferable to combine costs at one level of the lexicographic order by simple addition, at the risk of making certain costs explode at that level. The best compromise that has been found is therefore to **give the composer as much choice as possible**, by offering them the three possibilities, as well as the combination between the lexicographical order and the other two techniques. This allows the composer to compose their counterpoint iteratively: they make a first attempt that doesn't work well, adjust the costs to direct the search, and so on, until they get the result that suits them best. In practice, this leads to the user interface described in Appendix B, where a composer can choose their preferred order of importance, and then choose whether they want to combine the costs in a linear combination or a maximum minimisation fashion.

Chapter 5

Musicality of the solutions

This chapter deals with the musicality of the solutions. While the aim of the previous chapter was to discuss the costs, this chapter is oriented towards a normal use and what results a composer can achieve. We first look at basic solutions, i.e. compositions where species are not mixed, as in most of Fux's examples, then we will have a look at what happens when we try to leave Fux's writing style by using preferences, and finally we discuss a bit about cross-species compositions.

5.1 Combining first species with another species

Composing with three voices, one of which is the *cantus firmus*, one of which is a counterpoint of the first species, and the last of which is a counterpoint of any species, is the basic situation of writing counterpoint, as Fux explains in his teaching of counterpoint. In this first sub-section, therefore, we shall consider five basic counterpoints to a *cantus firmus* in C.



Figure 5.1: Simple first species composition with three voices. [Click here to listen.](#)



Figure 5.2: Simple second species composition with three voices. [Click here to listen.](#)

Looking at all these examples, we can draw the following conclusions:

- Fux was right when he said that three-part composition is much richer than two-part composition (without even having to mix the species). The addition of the third voice adds depth, dimension and colour, and the problems of monotony

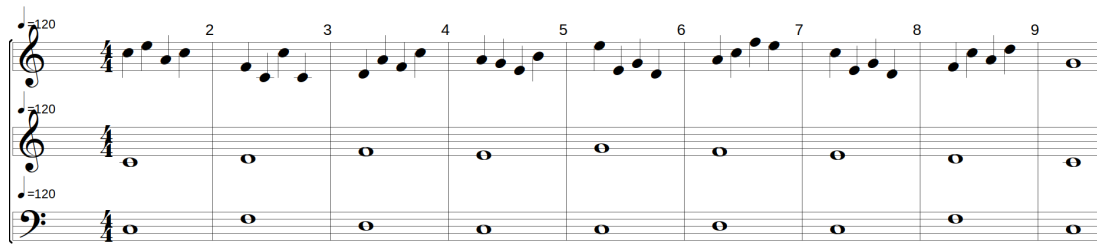


Figure 5.3: Simple third species composition with three voices. [Click here to listen.](#)

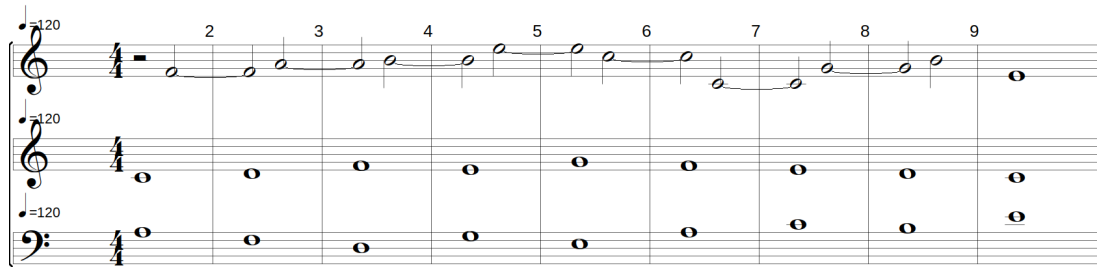


Figure 5.4: Simple fourth species composition with three voices. [Click here to listen.](#)

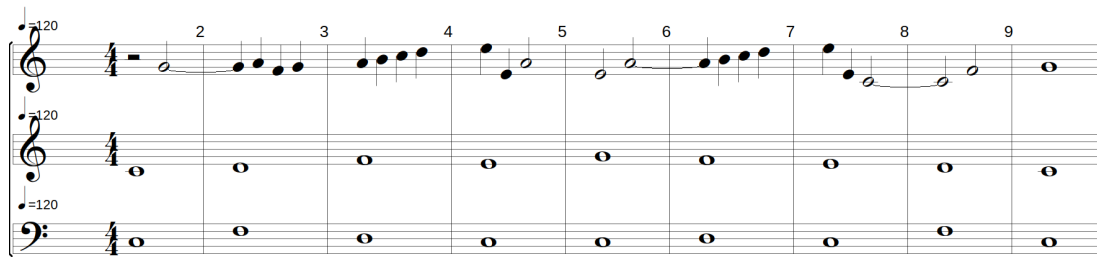


Figure 5.5: Simple fifth species composition with three voices. [Click here to listen.](#)

that arise in the automation of two-part writing seem to have been partially solved. With two voices, if one voice repeated itself, the composition as a whole suffered from repetitiveness. With three voices, if one voice repeats itself, because the other voices change, this creates variation. In fact, we could say more, as this is in keeping with the general idea of counterpoint, which is that patterns repeat and alternate. In this case, of course, the repetitions are due to chance (remember that all the rules, except the preference for variety, apply to a maximum of one measure, so the solver does not have an overall view of the composition), but in the future we could add rules to encourage these repetitions and variations, or even to encourage the voices to respond to each other.

- Too often the final chord is not conclusive. This could be solved by an additional rule to be defined. It is surprising that Fux does not cover this case in his rules. In fact, the only rules he mentions for the final chord are "no minor third" and "no tenth", which is apparently not enough.
- It would be difficult to call these compositions masterpieces. However, they stand up well enough to be used as a basis, and with a few personal improvements on the part of the composer, they can be expanded and improved as much as desired. For example, by changing the few notes that don't go together well, by modifying the harmonies or by adding different rhythms. The result of the

basic counterpoint is therefore more than satisfactory and can be considered a success.

5.2 Using preferences to improve musicality

Below is an example of the difference between a counterpoint of the first species, using Fux's preferences (more precisely, using a lexicographic search with default preferences as defined in Section 4.2.3), and a counterpoint in which personal preferences were expressed. These preferences were: to use as few contrary motions as possible, and as many oblique and direct motions as possible. Prioritising this preference (placing it first in the lexicographic order), then prioritising melodic intervals (maintaining a preference for small melodic intervals, as in *Gradus ad Parnassum*), then prioritising variety, and then placing all the other preferences at the penultimate level of the lexicographic order, with the exception of the preference for no successive perfect consonance, which was placed at the very last level.

The search has not been stopped manually; we leave it running until it finds the best solution according to the defined preferences.

The aim of this experiment is twofold: to show that the preferences can have a big impact on the resulting solution, giving the composer a lot of room for manoeuvre in their composition (this was already partially demonstrated in the previous chapter), and to obtain a solution that differs from Fux's compositions. In particular, these small changes in preferences were made to obtain a more monotonous solution, with fewer twists than the Fux-like solutions (for example, for transitions between two parts of a composition, for more quiet moments, ...).

The results of this experiment can be found in Figures 5.6 and 5.7.

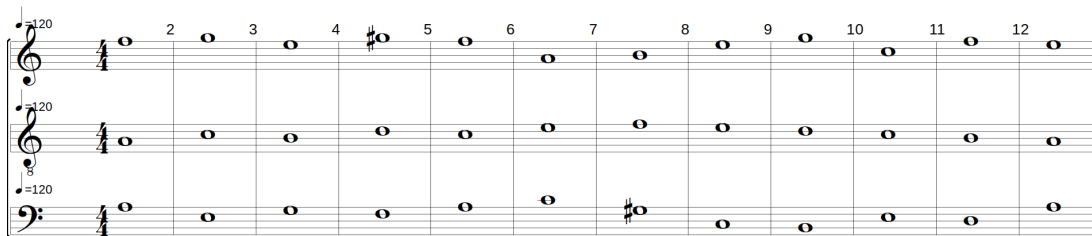


Figure 5.6: Example of a first species counterpoint in three-part composition with Fux's preferences. [Click here to listen.](#)

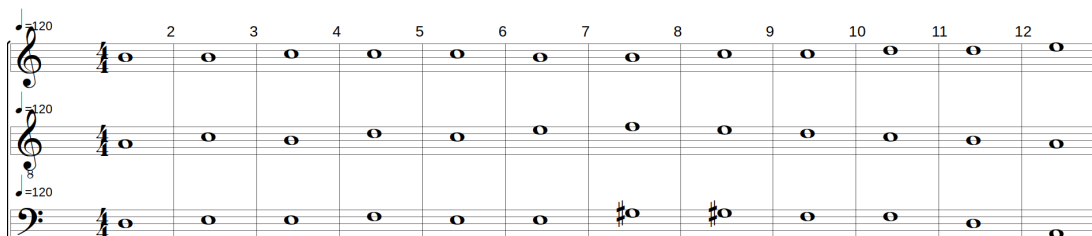


Figure 5.7: Example of a first species counterpoint in three-part composition with custom preferences. [Click here to listen.](#)

The result is striking, as the Fux-like solution simply sounds like... Fux, and the custom solution is completely in line with its aim, which was to create a composition

that is more monotonous and where there is less sense of things happening. This example shows something interesting: with the same set of hard constraints (the mandatory rules of Fux), it is possible, thanks to preferences, to make a variety of personal choices that still sound good but deviate from the traditional style of Fux.

5.3 Combining arbitrary species

One thing is clear about multi-species composition: it is a truly unpredictable art. It has already been discussed in Section 4.1.3 that the different interactions between species and voice ranges can take more or less time in terms of solver efficiency. This observation also applies to the quality of the solutions, with cases where the solver produces perfectly good results and other cases where the solution falls short of expectations. This lack of musical quality in the solutions was not the case when we were composing single species counterpoints (without mixing the species), and is something that emerges when combining species. The most plausible hypothesis is that the solver is unable to produce a solution that is *always* beautiful when combining species precisely because Fux has never given any rules specifying how to combine species. We can only hope that such rules exist in the 3rd chapter of his book, or else we'll have to either extrapolate from existing rules or take inspiration from other authors to create these rules.

Below (Figures 5.8 and 5.9) are two examples of counterpoint generated by FuxCP. Both combine counterpoint of the second kind and counterpoint of the fifth species. The search ran for three minutes before being interrupted. The solutions are a good example of the surprising results that can happen in the second, third and fourth beat.

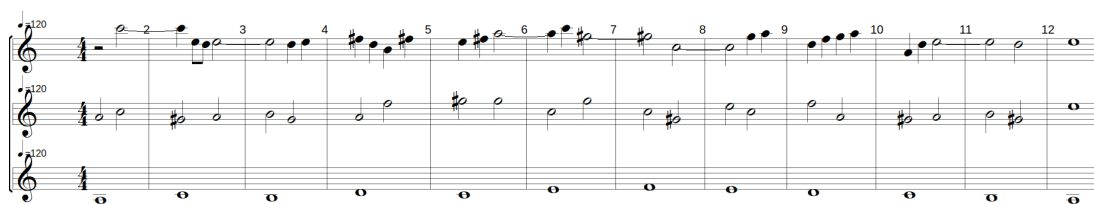


Figure 5.8: Example of a generated counterpoint using a second species counterpoint and a fifth species counterpoint, with an A scale. [Click here to listen.](#)

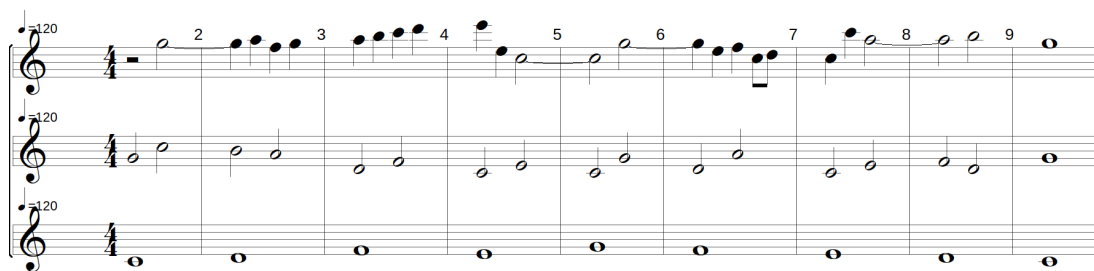


Figure 5.9: Example of a generated counterpoint using a second species counterpoint and a fifth species counterpoint, with a C scale. [Click here to listen.](#)

Chapter 6

Known issues and future improvements

This chapter looks at the current state of the FuxCP implementation and the problems that are currently known to exist. It also discusses possible improvements that should be considered by anyone wishing to take this work forward.

6.1 Known issues about the current state of the work

- As mentioned in Section 4.1.3, some few combinations of species, voice ranges and *cantus firmi* cause the solver to fail to find a solution. The current roundabout way to "solve" this is to change the voice ranges or some other parameter until a structural solution is found. These cases are relatively rare and do not prevent the use of FuxCP.
- If a counterpoint of the fourth species is the lowest stratum, the solver needs more time to find a solution in which all notes are ligated. This is not a problem when combining a fourth species counterpoint with a simple species counterpoint (first or second species), but becomes difficult to handle with more complex species (third or fifth species), as the search time before the solver finds a suitable solution (i.e. with all notes tied) can become very long.
- The current heuristics make the solver branch on the lowest stratum array with a "select the values smaller than $(\min + \max) / 2$ " policy. For some reason, it always seems to take the lowest value (or close to it), which means that the melody of the voice that is the lowest stratum is really redundant, something that even variety preference doesn't compensate for on complex species, since the time needed to find a good solution (i.e. one where the variety cost is low, which again means that many notes are different) is too high compared to the time needed to find a passable solution.

This leads to a second problem, which is that since the downbeat of the part, which is the lowest layer, always consists of the lowest possible note, the following notes (e.g. in the upbeat) will necessarily be higher. This is because the note in the 2nd, 3rd and 4th bars cannot be the same as the note in the 1st bar, but because the 1st bar is already the lowest possible, the notes in the 2nd, 3rd and 4th bars must be higher.

Figure 6.1 illustrates that the second species of counterpoint, in the bass, is composed exclusively of upbeat notes that are higher than the downbeat notes. This is a side effect of the heuristic, and a way should be found to resolve it. Note that in this example the problem is solved very quickly (a matter of seconds) because in this setting the other counterpoint is a first species counterpoint, which is a simple setting

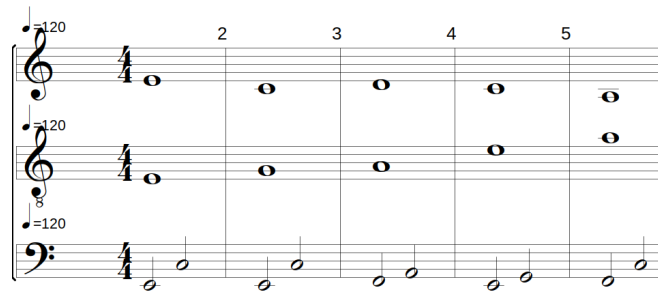


Figure 6.1: Illustration showing the limitation of current heuristics, as the counterpoint of the second species always has a higher upbeat than its downbeat. Note that the composition has been cropped.

6.2 Future improvements

The work towards fully automated counterpoint composition is progressing, but there is still a long way to go before we can claim to have finished the job. There are several ways to improve the current situation for those who want to improve FuxCP. Here are some ideas for improvement:

1. Solution quality

- *Finalise the formalisation of Gradus ad Parnassum* – The first obvious way to improve the tool is simply to complete the formalisation of Fux’s work, which would make it possible to compose in four voices, but also to integrate the additional rules he mentions in his last chapter. This would make it possible to have a complete FuxCP tool, in terms of Fux’s rules, and then to supplement Fux’s rules with additional rules that could come from any influence. This idea of adding external rules to the Fux rules has already been tested to some extent in the work of T. Wafflard, and the results were more than promising. It is therefore clearly a direction to take in the improvement of the FuxCP tool.
- *Relate the notes of the 2nd, 3rd and 4th beats to each other* – As mentioned in one of the preference ordering experiments (see Section 4.3.1), there are no direct constraints between the 2nd, 3rd and 4th beats of each counterpoint. The only way they influence each other is through the transitivity of the constraints: A and B are constrained together, and so are B and C, so A and C are connected in some way. The reason there are no direct constraints is that Fux didn’t mention any. Anyway, this leads to some unmelodicity, and it is definitely a good idea to find some rules (e.g. from other authors) that could deal with this unmelodicity.
- *Allow even more flexibility when setting the costs* – The current interface allows the user to assign an importance to each cost. This importance is used to sort the costs and perform a lexicographic search on them. If two costs have the same importance, they can be combined by either linear combination or maximum minimisation. The choice between linear combination and maximum minimisation applies to all importance levels, but the user should be able to choose which combination to perform for each importance level.
- *Address any of the current limitations* – Section 6.1 discusses some limitations and issues with the current state of the work. Solving any of them would be an improvement for FuxCP.

2. Software architecture

- *Migrate the project to C++* – Gecode is written in C++, and C++ is a language much better suited to managing implementations like FuxCP. GiL works really well, but has shown its limitations more than once: way too verbose, hard to manage objects (which are useful for designing FuxCP) since using Lisp, and lacking some of Gecode's features. These reasons alone are a huge incentive to migrate the whole implementation to C++. This would make it possible to further improve the implementation with more convenience and efficiency.

Here we repeat the words of T. Wafflard, who had already reached the same conclusion:

“Currently, constraints are added to a species via a long function that dispatches the constraints, rather than via class inheritance. Ideally, object-oriented inheritance should be used to represent the different variable arrays and species. All variable arrays (H, M, P, etc.) have something in common, whether in terms of their size relative to the cantus firmus, or in terms of the way certain rules are applied. A relatively abstract class should represent this type of array to enable these commonalities to be brought together.

The same applies to species that share common rules and should have been represented in a class system of their own. It would be logical for species to be children of the first species. Unfortunately, the scope of this work does not allow for a complete overhaul of the architecture. Moreover, in the near future, the entire code may have to be redone in C++ for reasons of performance, features, maintainability, and so on. Also, GiL has reached its limits, both in terms of ease of programming and in terms of possibilities. The Lisp language is not designed for writing mathematics, since each operation requires a different function call. Code readability can become complicated because these calls are all represented by parentheses. At the same time, it is not possible with GiL to combine basic mathematical operations to form a larger one. One has to break down each complex operation into simple intermediate basic operations a bit like writing assembly, which is undesirable for larger projects. Not to mention that branch-and-bound, heuristics, and multithreading seem complicated to implement in GiL.” [1, p.67]

3. Solver performance

- *Bettering the heuristics and reorganising the constraints* – Obviously, increasing the speed at which the solver finds solutions increases the speed at which the solver finds good solutions. It is therefore crucial to continue working on the heuristics to find better and better solutions. At the same time, once the globality of *Gradus ad Parnassum* has been formalised, it might be interesting to rethink the constraints that apply to the composition in an intelligent way, to make the solver's work easier and to have a set of constraints that hold together better.

This can also help the solver to find solutions for the difficult settings. For example, searching for a solution with two counterpoints of the fifth species

sometimes takes a long time.

4. Intended use of the FuxCP tool

- Throughout the work it is noticeable that all the examples given are quite short (fourteen bars at most). This is largely due to Fux himself, as the examples he gives are all of the same length, presumably for pedagogical purposes. He does not mention this explicitly, but there may also be a practical reason for considering only such short compositions. Indeed, these small compositions can be thought of as 'blocks' which can then be arranged to form a whole. Figure 6.2 is a rough outline of what such a composition would look like. The great advantage of this approach is that the counterpoints between the blocks can be of different kinds, allowing the composition to be constantly renewed. A future improvement for FuxCP could be to handle such small blocks, which would allow for long and renewed compositions.

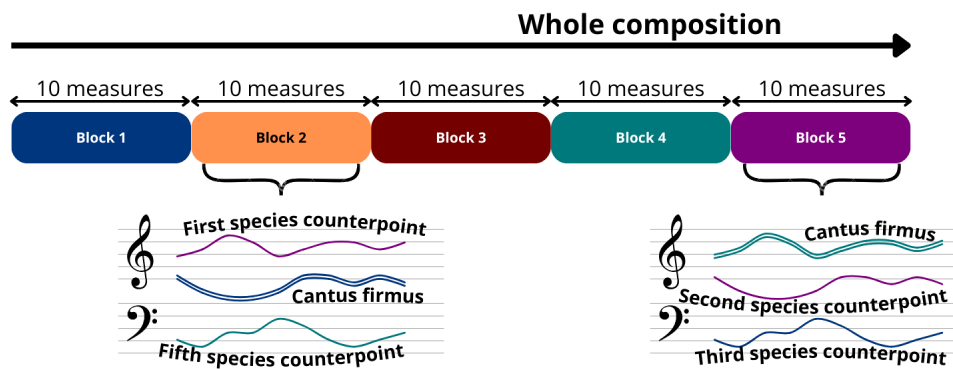


Figure 6.2: Example of what of a composition in blocks could look like

Conclusion

It is time to look back at the work that has been done, to highlight the progress that has been made, but also the shortcomings and gaps that need to be filled by future improvements. We will now discuss some of the key points that emerge from this thesis.

This thesis formalises *Gradus ad Parnassum* by J.J. Fux, a baroque composer who lived in the 18th century, in order to have an exhaustive record of the rules he imparts in his work. These rules concern the composition of counterpoint. More specifically, the aim of this work is to generalise to three voices the existing formalisation for two voices. Once formalised, these rules are implemented in the form of a constraint solving problem, which then makes it possible to obtain automatically generated counterpoints. The result of this work is expressed in the form of a highly customisable tool, in order to provide the user-composer with an easy-to-use tool that can assist them in their composition of counterpoint.

To be able to express Fux's rules in formal logic, it was necessary to create new concepts and variables, and even to change the paradigm: the *cantus firmus* lost its place of honour to the *lowest stratum*, i.e. the collection of the lowest sounding notes at any moment. We can be pleased that the formalisation for three voices maintains full compatibility with the two-part formalisation, and that the new variables and concepts introduced can also be used by the two-part formalised rules.

The implementation, which uses GiL to call the Gecode tool from Lisp, is also functional and allows anyone to generate counterpoints in a matter of seconds. While the musical results of these generations must be nuanced, as they are not always masterpieces, they are a good basis and the current implementation can be considered successful. A major drawback of the current implementation is the cases where the solver has difficulty finding a solution, in which case the waiting time to obtain counterpoints can be long (on the order of several minutes). This is relatively harmless at the moment, but it could become a problem if we continue to add constraints by adding voices. With regard to the possibility of customisation by the composer-user, a great deal of thought has been given to the best way of enabling the constraint solver to understand human preferences. This reflection has led to the conclusion that each user must be as free as possible to manage the costs of the search in order to be able to engage in iterative composition: they make a first attempt, observe the result, adjust the costs and try again, and so on. The interface reflects this policy and allows the user to manipulate the costs with great flexibility.

As far as the practical side of the implementation is concerned, a major limitation in the development of the tool is the restrictive aspect of coding in Lisp a program that would be much easier to code in object-oriented programming. Coding Gecode in Lisp is error prone (as it is much more verbose and less clear) and time consuming. If a potential successor to this work were to change one thing immediately, it would be to continue directly in C++, to make it easier to maintain high quality code, which indirectly improves the quality of the tool itself. Other main improvements may include incorporating four-voice writing and modern counterpoint scholarship to FuxCP, and exploring how constraint relaxations can generate interesting mutations of Fux's style.

In a more global perspective, we can think about the implications of this work for the future. First of all, last year T. Wafflard demonstrated the relevance of using constraint programming in the field of musical composition and more specifically in counterpoint. This work confirms this conclusion. Secondly, by following in the footsteps of previous works on counterpoint, music and constraint programming, and by adding its own contribution to this long line, this work contributes to the creation, perhaps one day, of a tool perfectly capable of generating highly customisable, multi-voice counterpoints on complete musical compositions. Far from replacing composers, this tool would be an excellent tool for beginners, giving them a solid base to build on, and a source of inspiration for the more experienced, who could play with different preferences to produce innovative compositions.

While there is still a considerable journey ahead, the building is steadily taking shape, brick by brick.

Bibliography

- [1] Thibault Wafflard. “FuxCP: a constraint programming based tool formalizing Fux’s musical theory of counterpoint”. Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2023. URL: <http://hdl.handle.net/2078.1/thesis:40739>.
- [2] Damien Sprockeels, Thibault Wafflard, Peter Van Roy, and Karim Haddad. “A Constraint Formalization of Fux’s Counterpoint”. In: *Journées d’Informatique Musicale (JIM)* (2023).
- [3] Klaus-Jürgen Sachs and Carl Dahlhaus. *Counterpoint*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.06690. URL: <https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000006690>.
- [4] Juliet Hess. “Balancing the counterpoint: Exploring musical contexts and relations”. In: *Action, Criticism & Theory for Music Education* 15.2 (2016), p. 50.
- [5] Richard Kramer. “Gradus ad Parnassum: Beethoven, Schubert, and the romance of counterpoint”. In: *19th-Century Music* 11.2 (1987), pp. 107–120.
- [6] Jaime Altozano. *De Pokemon a Bach. Una historia de voces*. Language: Spanish. July 2017. URL: <https://youtu.be/Mr8ICnGutYM?si=6W4XpNx-lTgJtaap>.
- [7] Heinrich Schenker. *Kontrapunkt*. German. Vol. 2: iss.1. Vol. 2 of Neue musikalische Theorien und Phantasien. Stuttgart: J.G. Cotta’sche Buchhandlung Nachfolger, 1906.
- [8] Knud Jeppesen and Glen Haydon. *Counterpoint: The Polyphonic Vocal Style of the Sixteenth Century*. English. Englewood Cliffs, N. J.: Prentice-Hall, 1960.
- [9] David Gaynor Yearsley. *Bach and the Meanings of Counterpoint*. English. Cambridge ; New York: Cambridge University Press, 2002.
- [10] Bill Schottstaedt. *Automatic Species Counterpoint*. Research Report STAN-M-19. Departement of Music, Stanford University, 1984. URL: <https://ccrma.stanford.edu/files/papers/stanm19.pdf>.
- [11] John Polito, Jason M Daida, and Tommaso F Bersano-Begey. “Musica ex machina: Composing 16th-century counterpoint with genetic programming and symbiosis”. In: *Evolutionary Programming VI: 6th International Conference, EP97 Indianapolis, Indiana, USA, April 13–16, 1997 Proceedings* 6. Springer. 1997, pp. 113–123.
- [12] Andres Garay Acevedo. “Fugue composition with counterpoint melody generation using genetic algorithms”. In: *International symposium on computer music modeling and retrieval*. Springer. 2004, pp. 96–106.
- [13] Gabriel Aguilera et al. “Automated generation of contrapuntal musical compositions using probabilistic logic in Derive”. In: *Mathematics and Computers in Simulation* 80.6 (2010), pp. 1200–1211. ISSN: 0378-4754. DOI: <https://doi.org/10.1016/j.matcom.2009.04.012>.
- [14] Dorien Herremans and Kenneth Sörensen. “Composing first species counterpoint with a variable neighbourhood search algorithm”. In: *Journal of Mathematics and the Arts* 6.4 (2012), pp. 169–189. DOI: 10.1080/17513472.2012.738554.

- [15] 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: <https://doi.org/10.1080/17459737.2014.935816>.
- [16] Ars Nova Software. *Counterpointer*. Developed and produced by Ars Nova Software. Address: PO Box 3333, Kirkland, WA 98083-3333, USA. Available on Microsoft Store and Mac App Store. 2019. URL: <https://www.ars-nova.com/counterpointer3.html>.
- [17] Alex Ding, En-Hua Holtz, John Chung, and Orion Bloomfield. *Counterpointer*. Project for Brown University’s CSCI 0320 course. 2021. URL: <https://github.com/counter-pointer/counterpointer>.
- [18] Richard L. Crocker. “Discant, Counterpoint, and Harmony”. In: *Journal of the American Musicological Society* 15.1 (1962), pp. 1–21. ISSN: 00030139, 15473848. URL: <http://www.jstor.org/stable/830051>.
- [19] Francesca Rossi, Peter Van Beek, and Toby Walsh. “Constraint programming”. In: *Foundations of Artificial Intelligence* 3 (2008), pp. 181–211.
- [20] Roman Barták. *Constraint Programming*. First Edition. 1998.
- [21] David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. “Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning”. In: *Discrete Optimization* 19 (2016), pp. 79–102.
- [22] IRCAM STMS Lab. *OpenMusic*. Version 7.2. URL: <https://openmusic-project.github.io>.
- [23] Guido Tack and Mikael Zayenz Lagerkvist. *Generic Constraint Development Environment*. Version 6.2.0. Gecode. URL: <https://www.gecode.org/index.html>.
- [24] James Bielman and Luís Oliveira. *CFFI: The Common Foreign Function Interface*. 2005. URL: <https://cffi.common-lisp.dev/>.
- [25] Baptiste Lapière. “Computer-aided musical composition Constraint programming and music”. Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2020.
- [26] Damien Sprockeels. “Melodizer: A Constraint Programming Tool For Computer-aided Musical Composition”. Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2021.
- [27] Clément Chardon, Amaury Diels, and Federico Gobbi. “Melodizer 2.0: A Constraint Programming Tool For Computer-aided Musical Composition”. Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2022.
- [28] Alfred Mann. *The Study of Counterpoint. From Johann Joseph Fux’s Gradus ad Parnassum*. English. Ed. and trans. Latin by Alfred Mann. Revised Edition. 500 Fifth Avenue, New York, N.Y. 10110: W.W. Norton & Company, 1971. ISBN: 0-393-00277-2. URL: http://www.opus28.co.uk/Fux_Gradus.pdf.
- [29] Simonne Chevalier. *Gradus ad Parnassum. Johann Joseph Fux*. French. Ed. by Gabriel Foucou. Trans. Latin by Simmone Chevalier. 2019. ISBN: 978-2-9556093-6-1.
- [30] Johann Joseph Fux. *Gradus ad Parnassum*. French. Trans. by Pierre Denis. Paris: Diod, Bijoutier, Garnier & Cadet, 1773. URL: https://s9.imslp.org/files/imglnks/usimg/b/b2/IMSLP231222-PMLP187246-fux_traite_de_composition_1773.pdf.

- [31] Johann Joseph Fux. *Gradus ad Parnassum*. German. Ed. and trans. by Lorenz Christoph Mizler. Leipzig: Mizler, 1742. URL: <https://s9.imslp.org/files/imglnks/usimg/6/67/IMSLP273867-PMLP187246-gradusadparnassu00fuxj.pdf>.
- [32] Johann Joseph Fux. *Gradus ad Parnassum*. Reprinted in 1966 by Broude Bros., New York. Vienna: Johann Peter van Ghelen, 1725. URL: https://s9.imslp.org/files/imglnks/usimg/f/fd/IMSLP91138-PMLP187246-Fux_-_Gradus_ad_Parnassum.pdf.
- [33] Brandon McNair. *Understanding Stratum in Geology, Including Types and Examples*. Accessed on December 3, 2023. 2023. URL: <https://geologybase.com/stratum/>.
- [34] Noël Gallon and Marcel Bitsch. *Traité De Contrepoint*. French. Ed. by Durand et Cie. Paris, 1964. URL: <https://artinfuser.com/exercise/md/pdf/Gallon-Bitsch-Contrepointe.pdf>.
- [35] Walter Piston. *Harmony: Fifth Edition*. English. Ed. by Mark DeVoto. 5th. W. W. Norton & Company, 1987, p. 32.

Appendix A

Software Architecture

This appendix summarises the architecture of the software. The first figure (A.1) shows how FuxCP is integrated into the tools it uses, and the second figure (A.2) shows the internal structure of FuxCP.

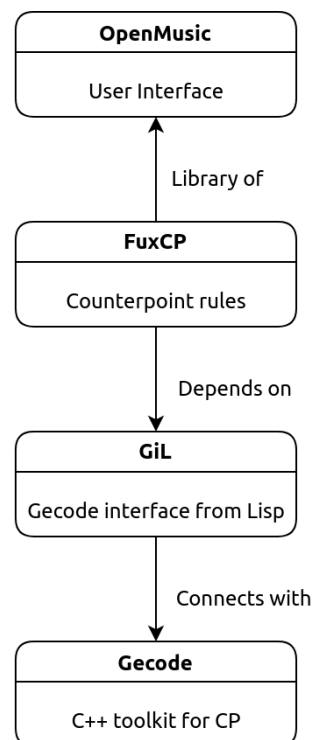


Figure A.1: Integration of FuxCP within the other tools

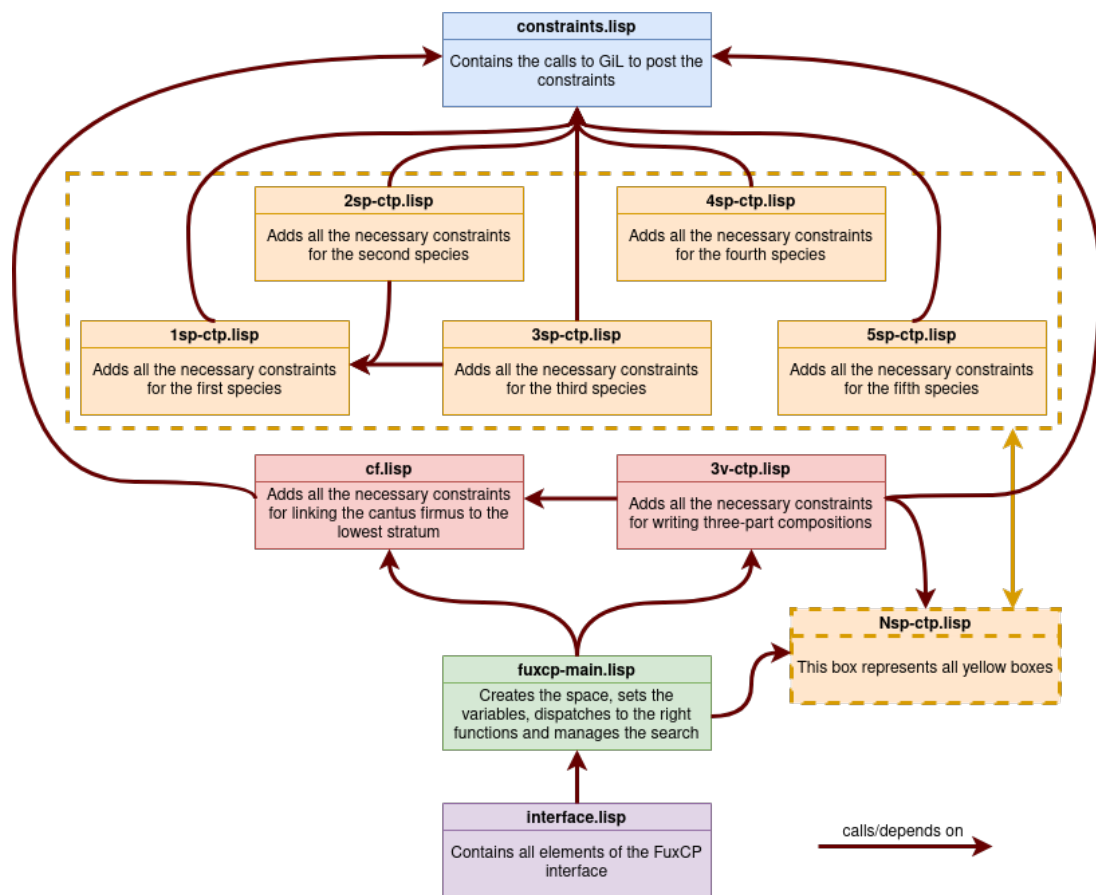


Figure A.2: Internal structure of FuxCP

Appendix B

User Guide

This manual provides an overview of FuxCP, covering the installation process, its use within OpenMusic and a description of the costs displayed in the interface. Although FuxCP is designed to be compatible with all platforms, it relies on GiL, which currently only works on MacOS and Linux. Unfortunately, GiL does not support Windows due to compatibility issues between the 32-bit Lisp licence used by OpenMusic and the 64-bit Gecode Windows version. Although it is technically possible to obtain a 32-bit version of Gecode for Windows, this is not recommended.

B.1 Installing FuxCP

B.1.1 Prerequisites

To use FuxCP you need to download and install the following tools:

- Gecode : <https://www.gecode.org/download.html/>
- OpenMusic : <https://openmusic-project.github.io/openmusic/>

And download the following libraries¹:

- GiL : <https://github.com/sprockelsd/GiLv2.0/>
- FuxCP : <https://github.com/sprockelsd/Melodizer/>

There are other tools available on the latest GitHub, such as Melodizer and Melodizer2.0. For the purposes of this guide, only the FuxCP folder is needed.

B.1.2 Loading FuxCP in OpenMusic

In order to use the above libraries, OpenMusic must be running. When opening any workspace, locate the toolbar at the top of the interface. Click on the "Windows" button, highlighted in the figure B.1, and select "Library" from the drop-down menu. This will bring up a new window. From the toolbar of this window, select 'File' and then 'Add Remote Library'. Navigate through your file system to find the path where the previously downloaded FuxCP and GiL libraries are stored. Once located, the libraries should appear under the "Libraries" folder in the "Library" window, as shown in Figure B.2. Right click on "fuxcp" and select "Load Library". If no errors occur, the setup is complete.

However, if an error occurs, it may be a linking problem with the Gecode library. For MacOS users, a script from the `c++` folder of the GiL library can be used. Edit the path to Gecode within the script to match your system configuration. Linux users should add the Gecode library to the `LD_LIBRARY_PATH` variable. Go to the `/etc/ld.so.conf.d` folder and create a new `.conf` file if one does not already exist.

¹In case the following links are not working, here is a backup link that contains both FuxCP for three voices and the matching GiL version: <https://github.com/PanoLeRamix/FuxCP3/>

In this file, add the full path to the Gecode library, save it, and run `sudo ldconfig` to update the system with the new library. Don't forget to restart OpenMusic and don't lose hope. Following these steps should ensure that FuxCP works properly.

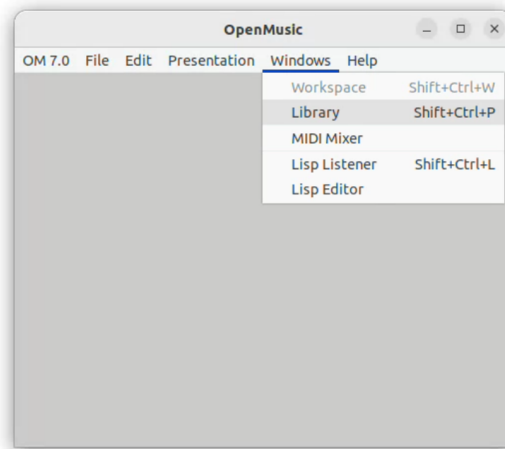


Figure B.1: Opening the "Library" window in OpenMusic.

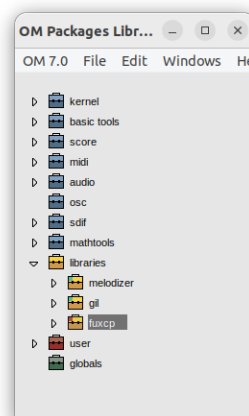


Figure B.2: Loading the "fuxcp" library in OpenMusic.

B.2 Using FuxCP in OpenMusic

Setup

Using FuxCP in OpenMusic is straightforward. There is a single block that contains the entire graphical interface of the tool. This block or class is called `cp-params`.

Using the example patch An example patch, `FuxCP_example.omp`, is provided in the 'examples' folder of FuxCP. This patch is the same as the one shown in the figure B.3. To use it, right-click anywhere in the OpenMusic workspace and select *ImportFile*. Select the sample patch and double-click to open it. Voilà, ready to use!

Setting up your own patch If for some reason the example patch is not available, you can set up your own patch as follows. Right-click anywhere in the OpenMusic workspace and select *New... → NewPatch*. Double-click your patch. Once in your patch, right-click anywhere in the patch and select *Classes → Libraries → FuxCP → Solver → CP – PARAMS*. Alternatively you can just double click anywhere in the patch, type "fuxcp::cp-params" and press enter. This also works for "poly", "voice" and "x-append".

Once this block has appeared, all you have to do is bind an OM voice object, representing the *cantus firmus*, to the second argument of cp-params as shown in figure B.3. Don't forget to block the input voice object and evaluate cp-params so it can detect the new input. Now cp-params can be blocked too. From now on, you could directly use the interface and generate counterpoints using the tool. If you want to retrieve the voice object containing the counterpoint generated by the tool, just bind the third argument on the output side to a voice object. Once bound, it is then possible to evaluate the voice object so that it updates.

If you want to get the whole composition in one object, you have to do some fiddling with OpenMusic. The simplest way to do this is shown in Figure B.3, and works as follows: get the POLY object returned by CP-PARAMS on its third output, split this object in two (the two voices), then get the *cantus firmus*, which is the second output of CP-PARAMS, and put all the voices back together in the desired order using x-append functions.

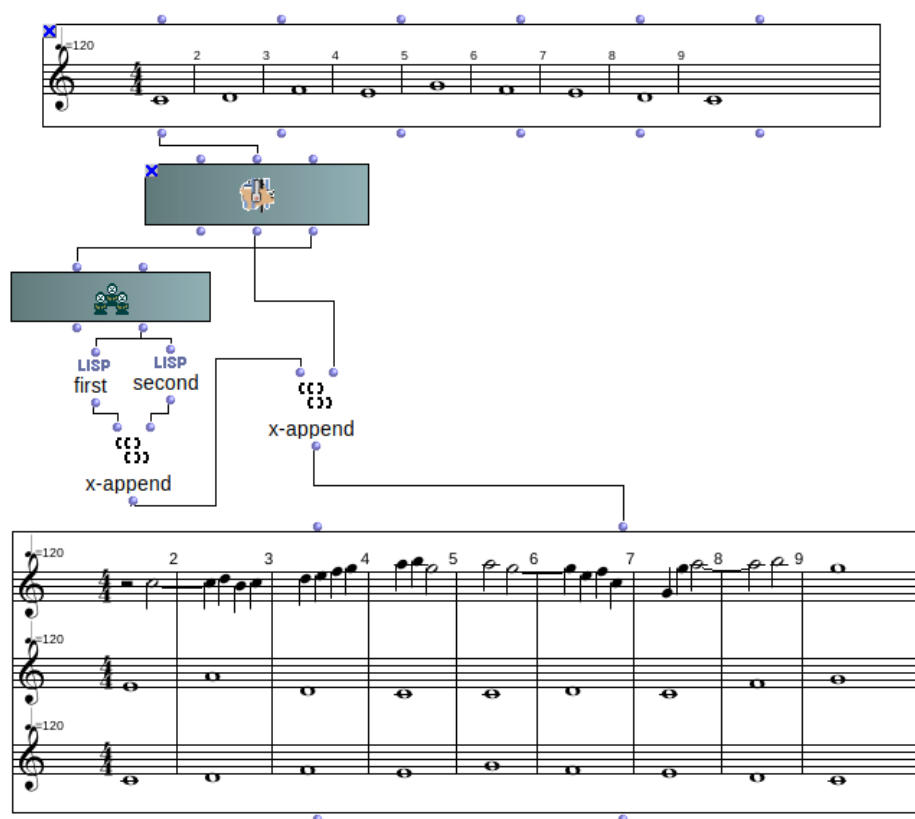


Figure B.3: View of a patch using fuxcp::cp-params in OpenMusic.

Listening to the solution

OpenMusic does not have built-in sound. You will need to use a third party application to listen to the result. Here is a solution that works to listen to the music: having installed TiMidity++², run the following command before opening OpenMusic:

```
timidity -iA -B2,8 -Os
```

Then go to *OpenMusic* → *Preferences* → *MIDI* → *Ports setup* → *Output devices* and select "TiMidity port 0".

Use the interface

But how do you use the interface? Simply double-click on the block to bring it up. The interface is sorted from left to right, so the preferences are divided into several categories: "General preferences", "Melodic preferences", "Species-Specific Preferences", "Solver Configuration" and, in the bottom right corner, "Solver Launcher" (see figure B.4).

Choose the preferences You will notice that there are almost always two settings for each preference. The first is the importance: it corresponds to the priority the solver will give to reducing the value of this cost. An importance of 1 means that it will be the absolute priority of the solver, whereas a preference of 14 means that the solver will minimise this cost if it doesn't affect the other costs. The second setting is the value: it determines the actual value of the cost corresponding to the preference. It is very useful when two preferences are set to have the same importance, in which case their respective cost values will have an effect. For example, if cost *A* and cost *B* both have an importance of 1, but cost *A* has a very high cost compared to *B*, *A* will affect the quality of the solution more than *B*, even though they both have the same importance. If two costs have the same importance, the yellow panel (bottom left) allows you to choose how to combine them: either by linear combination or by maximum minimisation. For more information, see Chapter 4 of this thesis that explains how costs work in detail. The default costs are supposed to represent Fux's preferences.

All costs are explicitly defined in the following section, in Table B.1.

Start the search Pressing the "Next Solution" button will display the solution as a pop-up. What appears on the screen are the two counterpoints. The *cantus firmus* must be added manually by using the 'x-append' functions. To see the complete composition, you must evaluate the 'poly' object. To do so, right click on it, and click 'evaluate'. Alternatively, you can click on it and press 'v'.

The other option is to press the "Best Solution" button. This will start an infinite search that will only stop when the best solution has been found (which can take hours). You can evaluate the output object at any time to see what the best result is so far, and this will not stop the search, so you can see how the solution improves step by step, and stop the search when it has produced something you are happy with.

The button "Stop" allows you to stop the search. This button can take up to 5 seconds to actually stop the search. If a search takes too long, we recommend you to stop the search, change the voice range and start again. Please note that the preferences do not affect the speed of finding the first solution. The first solution is the first valid solution and is not affected by the costs. Only the subsequent solutions can be affected by the preferences.

²TiMidity++ is a software synthesizer that can play MIDI files without a hardware synthesizer, click here to install.

OM 7.2

File

Windows

CP-PARAMS

General preferences

	Importance	Value
Borrowed notes	8	High cost
Harmonic fifths on the downbeat	7	Low cost
Harmonic octaves on the downbeat	5	Low cost
Successive perfect consonances	2	Medium cost
Repeating notes	9	Medium cost
No harmonic triad	3	Medium cost
Direct motion to perf. consonance	14	Last resort
Motion cost	12	
---Direct motion		Medium cost
---Oblique motion		Low cost
---Contrary motion		No cost
Apply specific penultimate note rules		Yes

First choose the importance of each preference (1 being the most important and 14 being the least important). The solver will give priority to the most important preferences. The cost value is taken into account if two costs have the same importance.

If two costs are ranked the same, perform between them a:

Linear combination

Melodic Preferences

	Importance	Value
Melodic cost	13	
---Steps		No cost
---Third skips		Low cost
---Fourth leaps		Low cost
---Tritone leaps		Forbidden
---Fifth leaps		Medium cost
---Sixth leaps		Medium cost
---Seventh leaps		Medium cost
---Octave leaps		Low cost

Solver Configuration

First voice species	1st
First voice range	Really far above
Second voice species	1st
Second voice range	Above
Borrowing mode	Major
Minimum % of skips	

Second species specific pref.

Penultimate downbeat note is a fifth

6

Last resort

Third species specific pref.

No cambiata

11

High cost

Force contrary motion after skip

No

No harmonic triad in 2nd/3rd beat

4

Medium cost

Third and fourth species specific pref.

Same note in downbeat and upbeat

10

Low cost

Fourth species specific pref.

No ligatures

1

Last resort

Fifth species specific pref.

Many quarters (left) or many syncopations (right)

Solver Launcher

Save Config

Stop

Next Solution

Best Solution

Figure B.4: User interface of the fuxcp : : cp-params class in OpenMusic.

B.3 Interface Parameters Description

Table B.1 describes all the parameters available in the interface.

Name	Description	Default value
Borrowed notes	Preference for borrowed notes outside the diatonic scale. A high cost means as few borrowed notes as possible.	High cost
Harmonic fifths on the downbeat	High cost means as few harmonic fifths on the downbeats as possible.	Low cost
Harmonic octaves on the downbeat	High cost means as few harmonic octaves on the downbeats as possible.	Low cost
Successive perfect consonances	High cost means as few successive perfect consonances as possible.	Medium cost
Repeating notes	High cost means as few repeating notes as possible, i.e. as many different notes as possible. This cost corresponds to the variety cost.	Medium cost
No harmonic triad	High cost means as many harmonic triads as possible.	Medium cost
Direct motion to perfect consonance	High cost means as few direct motions to perfect consonances as possible.	Last resort
Direct motion	High cost means as few direct motions as possible.	Medium cost
Oblique motion	High cost means as few oblique motions as possible.	Low cost
Contrary motion	High cost means as few contrary motions as possible.	No cost
Apply specific penultimate note rules	Force all rules on the notes of the penultimate measure. This applies only to two-part composition and refers to the penultimate note having to be a major sixth or a minor third.	Yes
Steps	High cost means as few steps as possible.	No cost
Third skips	High cost means as few third skips as possible.	Low cost
Fourth leaps	High cost means as few fourth leaps as possible.	Low cost
Tritone leaps	High cost means as few tritone leaps as possible.	Forbidden
Fifth leaps	High cost means as few fifth leaps as possible.	Medium cost
Sixth leaps	High cost means as few sixth leaps as possible.	Medium cost
Seventh leaps	High cost means as few seventh leaps as possible.	Medium cost
Octave leaps	High cost means as few octave leaps as possible.	Low cost
2nd: Penultimate downbeat note is a fifth	High cost means trying to ensure that the penultimate downbeat is not a fifth.	Last resort
3rd: No cambiatas	A high cost means as many cambiatas as possible	High cost
3rd: Force contrary motion after skip	Force that a melodic skip or leap is followed by a melodic step in a contrary motion.	No
3rd: No harmonic triad in 2nd/3rd beat	High cost means as many harmonic triads as possible on the 2nd and 3rd beat.	Medium cost
3rd& 4th: Same note in downbeat and upbeat two beats apart	High cost means as many different notes in the downbeat and upbeat.	Low cost
4th: No ligatures	High cost means as few not-ligatured notes, i.e. as many ligatures as possible.	High cost
5th: Many quarters or many syncopations	Determines the minimum percentage of quarter notes or syncopations in the fifth species. Pushing the slider all the way to one side is not recommended.	<center>
Voice species	Determines the type of counterpoint that the tool will generate.	1st and 1st
Voice range	Determines around which pitch the counterpoint will be generated depending on the pitch of the first note of the <i>cantus firmus</i> .	Above and very far above
Minimum % of skips	Determines, depending on the counterpoint size, the percentage of melodic intervals larger than one step.	0%
Borrowing mode	Type of scale from which notes can be borrowed to generate counterpoint. The first note of the <i>cantus firmus</i> determines the tonic of this scale. If none is selected, only natural notes are used. Applies everywhere except the penultimate bar.	Major
Save Config	Saves all established preferences and allows you to start a new search for this configuration later.	-
Next Solution	Starts or continues the search for the previously saved configuration. Displays a new window with the first better solution found. Displays an error message if no solution can be found.	-
Stop	Pause the search. This may take up to 5 seconds to take effect.	-
Best Solution	Starts or continues the search for the previously saved configuration. Does not display a window, but returns the best solution found so far, accessible by evaluating the output of cp-params. Displays an error message if no other solution can be found.	-
Linear combination or maximum minimisation	Choose whether the equally important costs are combined according to a linear combination or according to a maximum minimisation. More details about it in Chapter 4.	Linear combination

Table B.1: Description of the parameters of `fuxcp : cp-params`.

Appendix C

Complete set of rules for two and three part compositions

This appendix contains all the constraints for composing counterpoint for two or three voices. This appendix contains only the formalised equations, the corresponding explanations can be found in the corresponding sections of this thesis and that of T. Wafflard.

All the rules apply to three-voice compositions, but only the rules for two voices apply to two-voice compositions. Rules for three-part compositions are indicated by '3V' at the beginning of the rule.

Some of the rules are different depending on whether the composition is for two or three voices. In these cases, the mathematical relationship to be followed for the rule in question, depending on the situation, is clearly indicated.

Functions

Some rules, formalised by T. Wafflard, use functions that simplify the notation. Here are those functions:

nextm(x) Returns the number of measure(s) to add in 4/4 time signature depending on the number of beat x .

$$\text{nextm}(x) = \begin{cases} 1 + \text{nextm}(x - 4) & \text{if } x \geq 4 \\ 0 & \text{otherwise} \end{cases} \quad (\text{C.1})$$

buildScale(key, scale) Returns the set of notes in the *key* based on the *scale* used. *key* is a value between 0 and 11 such that $0 \equiv C$ and $11 \equiv B$.

$$\begin{aligned} & \forall x \in [-11, 127], \forall \delta := \text{key} + x \in [0, 127] \\ \text{buildScale}(\text{key}, \text{scale}) &= \begin{cases} \bigcup_{\delta \bmod 12 \in \text{key} + \{0, 2, 4, 5, 7, 9, 11\}} \delta & \text{if } \text{scale} = \text{major} \\ \bigcup_{\delta \bmod 12 \in \text{key} + \{0, 2, 3, 5, 7, 8, 10\}} \delta & \text{if } \text{scale} = \text{minor} \\ \bigcup_{\delta \bmod 12 \in \text{key} + \{0, 5, 9, 11\}} \delta & \text{if } \text{scale} = \text{borrowed} \end{cases} \quad (\text{C.2}) \\ & \text{where } \text{key} \in [0, 11], \text{scale} \in \{ \text{"major"}, \text{"minor"}, \text{"borrowed"} \} \end{aligned}$$

N.B.: $\text{buildScale}(\text{key}, \text{"minor"}) \equiv \text{buildScale}([\text{key} + 3] \bmod 12, \text{"major"})$.

positions(upto) Function that returns the set of non-empty positions or indexes ordered depending on the species in such a way that all the positions would follow one another to represent all the beats of that species on a score in a single list.

$$\begin{aligned}
positions(upto) &= \bigcup_{\forall i \in \mathcal{B}, \forall j \in [0, upto)} [i, j] \\
\text{s.t. } \forall x \in [1, 3], \forall y \in [1, upto) & \\
[i, j] <_s [i + x, j] <_s [i, j + y] & \\
\text{where } <_s \text{ means the sorting order} &
\end{aligned} \tag{C.3}$$

By extension, $\rho + z >_s \rho$ such that:

$$\begin{aligned}
\forall z \in \mathbb{N}^+, \forall \rho = [i, j] \in positions(upto) & \\
\rho + z = [i + zd, j + nextm(i + zd)] &
\end{aligned}$$

where $nextm()$ is a function that returns the correct number of measure(s) to add. (C.4)

Implicit General Rules of Counterpoint

G1 *Harmonic intervals are always calculated from the lower note.*

Already handled by making the difference value absolute for the **H** variable.

G2 *The number of measures of the counterpoint must be the same as the number of measures of the cantus firmus.*

Listing C.1: Definition of N in the first species.

```

1  (defvar *notes (list nil nil nil nil))
2  ; ...
3  ;; FIRST SPECIES ;;
4  ; setting the first list of *notes with
5  ;   integer *cf-len as size
6  ;   set *extended-cp-domain as available notes
7  (setf (first *notes)
8        (gil::add-int-var-array-dom *sp* *cf-len *extended-cp-domain))

```

G3 *The counterpoint must have the same time signature and the same tempo as the cantus firmus.*

Listing C.2: Definition of N in the first species.

```

1  (defvar *notes (list nil nil nil nil))
2  ; ...
3  ;; FIRST SPECIES ;;
4  ; setting the first list of *notes with
5  ;   integer *cf-len as size
6  ;   set *extended-cp-domain as available notes
7  (setf (first *notes)
8        (gil::add-int-var-array-dom *sp* *cf-len *extended-cp-domain))

```

G4 *The counterpoint must be in the same key as the cantus firmus.*

G5 This rule is already handled by the creation of the set $\mathcal{N}$. The example of the actual rule given above will clarify the explanations. Let k be the value of the key determined by the key signature, i.e. 60 for C ; and t the tonic of the piece, i.e. $N[0] = 65$. Then:

$$\begin{aligned}
\mathcal{N}_{key} &= \text{buildScale}(k \bmod 12, "major") = \{0, 2, 4, 5, 7, 9, 11, 12, \dots, 127\} \\
\mathcal{N}_{brw} &= \text{buildScale}(t \bmod 12, "borrowed") = \{2, 4, 5, \mathbf{10}, 14, \dots, 125\} \\
\therefore \mathcal{N}_{all} &= \{0, 2, 4, 5, 7, 9, \mathbf{10}, 11, 12, \dots, 127\}
\end{aligned}$$

To ensure that borrowed notes are used sparingly, they must be given a cost to use. Let $Offkey$ be the set of notes outside the key and $Offkey_{costs}$ the list of costs associated with each note. The cost for a note will be $\langle no\ cost \rangle$ or $cost_{Offkey}$ (DFLT: $\langle high\ cost \rangle$).

$$\begin{aligned}
Offkey &= [0, 1, 2, \dots, 127] \setminus \mathcal{N}_{key} \\
&\quad \forall \rho \in \text{positions}(m) \\
Offkey_{costs}[\rho] &= \begin{cases} cost_{Offkey} & \text{if } N[\rho] \in Offkey \\ 0 & \text{otherwise} \end{cases} \quad (C.5) \\
\text{moreover } \mathcal{C} &= \mathcal{C} \cup \sum_{c \in Offkey_{costs}} c
\end{aligned}$$

G6 *The range of the counterpoint must be consistent with the instrument used.*

This rule is already handled by the creation of the set $\mathcal{N}^{\mathcal{R}} = \mathcal{N} \cap \mathcal{R}$. When N is created its domain is set to $\mathcal{N}_{all}^{\mathcal{R}}$ as seen in the code sample C.2: `*extended-cp-domain` refers to the set $\mathcal{N}_{all}^{\mathcal{R}}$.

G7 *Chromatic melodies are forbidden.*

$$\begin{aligned}
&\forall \rho \in \text{positions}(m-2) \\
(M_{brut}[\rho] = 1 \wedge M_{brut}[\rho+1] = 1) &\iff \perp \\
(M_{brut}[\rho] = -1 \wedge M_{brut}[\rho+1] = -1) &\iff \perp
\end{aligned} \quad (C.6)$$

G8 *Melodic intervals should be small.*

$$\begin{aligned}
&\forall \rho \in \text{positions}(m-1) \\
Mdeg_{costs}[\rho] &= \begin{cases} cost_{secondMdeg} & \text{if } M[\rho] \in \{0, 1, 2\} \\ cost_{thirdMdeg} & \text{if } M[\rho] \in \{3, 4\} \\ cost_{fourthMdeg} & \text{if } M[\rho] = 5 \\ cost_{tritoneMdeg} & \text{if } M[\rho] = 6 \\ cost_{fifthMdeg} & \text{if } M[\rho] = 7 \\ cost_{sixthMdeg} & \text{if } M[\rho] \in \{8, 9\} \\ cost_{seventhMdeg} & \text{if } M[\rho] \in \{10, 11\} \\ cost_{octaveMdeg} & \text{if } M[\rho] = 12 \end{cases} \quad (C.7) \\
\text{moreover } \mathcal{C} &= \mathcal{C} \cup \sum_{c \in Mdeg_{costs}} c
\end{aligned}$$

G9 *3V - The last chord must be composed only of the notes of the harmonic triad.*

$$\forall s \in \{b, c\}: H(s)[0, m-1] \in Cons_{h_triad} \quad (C.8)$$

G10 *3V - The last chord must have the same fundamental as the one of the scale used throughout the composition.*

$$N(a)[0, m-1] \bmod 12 = N(cf)[0, 0] \bmod 12 \quad (C.9)$$

Constraints of the First Species

Harmonic Constraints of the First Species

1.H1 *All harmonic intervals must be consonances.*

$$\forall j \in [0, m) \quad H[0, j] \in Cons \quad (C.10)$$

1.H2 *The first harmonic interval must be a perfect consonance.*

- When dealing with two-part composition:

$$H[0, 0] \in Cons_p \quad (C.11)$$

- When dealing with three-part composition: The rule doesn't exist.

1.H3 *The last harmonic intervals must be a perfect consonance.*

- When dealing with two-part composition:

$$H[0, m - 1] \in Cons_p \quad (C.12)$$

- When dealing with three-part composition: The rule doesn't exist.

1.H4 *The key tone is tuned according to the first note of the cantus firmus.*

$$\begin{aligned} \neg IsCfB[0, 0] &\implies H[0, 0] = 0 \\ \neg IsCfB[0, m - 1] &\implies H[0, m - 1] = 0 \end{aligned} \quad (C.13)$$

1.H5 *The voices cannot play the same note at the same time except in the first and last measure.*

$$\forall p_1, p_2 \in \{cf, cp_1, cp_2\}, \text{ with } p_1 \neq p_2 \forall j \in \{0, 1, 2, 3\} \forall j \in [1, m - 1) \quad N(p_1)[i, j] \neq N(p_2)[i, j] \quad (C.14)$$

1.H6 *Imperfect consonances are preferred to perfect consonances.*

$$\begin{aligned} &\forall j \in [0, m) \\ Pcons_{costs}[j] &= \begin{cases} cost_{Pcons} & \text{if } H[0, j] \in Cons_p \\ 0 & \text{otherwise} \end{cases} \\ \text{moreover } \mathcal{C} &= \mathcal{C} \cup \sum_{c \in Pcons_{costs}} c \end{aligned} \quad (C.15)$$

1.H7 and **1.H8** *The harmonic interval of the penultimate note must be a major sixth or a minor third depending on the cantus firmus pitch. When writing with three voices, the harmonic interval must be either a minor third, a perfect fifth, a major sixth or an octave.*

- When dealing with two-part composition:

$$\begin{aligned} \rho &:= \max(positions(m)) - 1 \\ H[\rho] &= \begin{cases} 9 & \text{if } IsCfB[\rho] \\ 3 & \text{otherwise} \end{cases} \end{aligned} \quad (C.16)$$

where ρ represents the penultimate index of any counterpoint.

- When dealing with three-part composition:

$$H[0, m - 1] \in \{0, 3, 7, 9\} \quad (\text{C.17})$$

1.H9 *3V - One might use sixths or octaves.* As discussed in **1.H9**, there is no constraint to add for this rule.

1.H10 *3V - Tenths are prohibited in the last chord.*

$$H_{\text{brut}}[0, m - 1] > 12 \implies H[0, m - 1] \notin \{3, 4\} \quad (\text{C.18})$$

1.H11 *3V - Octaves should be preferred over unisons.* As discussed in **1.H11**, there is no constraint to add for this rule.

1.H12 *3V - Last chord cannot include a minor third.*

$$H[0, m - 1] \neq 3 \quad (\text{C.19})$$

Melodic Constraints of the First Species

1.M1 *Tritone melodic intervals are forbidden.*

$$\begin{aligned} &\forall \rho \in \text{positions}(m - 1) \\ M[\rho] = 6 &\implies M_{\text{degcosts}}[\rho] = \text{cost}_{\text{tritone}} M_{\text{deg}} \end{aligned} \quad (\text{C.20})$$

1.M2 *Melodic intervals cannot exceed a minor sixth interval.*

$$\forall j \in [0, m - 1) \quad M[0, j] \leq 8 \quad (\text{C.21})$$

1.M3 *3V - Steps are preferred to skips.* This rule is a duplicate of rule **G8**.

1.M4 *3V - The notes of each part should be as diverse as possible.*

$$\begin{aligned} &\forall p \in \{cp_1, cp_2\}, \quad \forall j \in [0, m - 1), \quad \forall k \in [j + 1, \min(j + 3, m - 1)] : \\ N(p)[0, j] = N(p)[0, j + k] &\iff \text{cost}_{\text{variety}}[j + m * k] = 1 \end{aligned} \quad (\text{C.22})$$

1.M5 *Each part should stay in its voice range.*

This rule is already covered by the definition of the voice ranges, so no constraint is associated to it.

1.M6 *3V - Melodic intervals cannot be greater or equal to a sixth.* This rule is only a restatement of rule **1.M2**, saying that melodic intervals cannot exceed a minor sixth interval.

Motion Constraints of the First Species

1.P1 *Perfect consonances cannot be reached by direct motion.*

- When dealing with two-part composition:

$$\forall j \in [0, m - 1) \quad H[0, j + 1] \in \text{Cons}_p \implies P[0, j] \neq 2 \quad (\text{C.23})$$

- When dealing with three-part composition:

$$\begin{aligned}
& \forall j \in [0, m-2] : \\
& P[0, j] = 2 \wedge H[0, j+1] \in Cons_p \\
& \iff cost_{direct_move_to_p_cons}[j] = 8
\end{aligned} \tag{C.24}$$

1.P2 *Contrary motions are preferred to oblique motions which are preferred to direct motions.*

- | | | |
|---|--|---|
| <ul style="list-style-type: none"> • $cost_{con}$
DFLT: <no cost> | <ul style="list-style-type: none"> • $cost_{obl}$
DFLT: <low cost> | <ul style="list-style-type: none"> • $cost_{dir}$
DFLT: <medium cost> |
|---|--|---|

$$\begin{aligned}
& \forall j \in [0, m-1) \\
& P_{costs}[j] = \begin{cases} cost_{con} & \text{if } P[0, j] = 0 \\ cost_{obl} & \text{if } P[0, j] = 1 \\ cost_{dir} & \text{if } P[0, j] = 2 \end{cases} \\
& \text{moreover } \mathcal{C} = \mathcal{C} \cup \sum_{c \in P_{costs}} c
\end{aligned} \tag{C.25}$$

1.P3 *At the start of any measure, an octave cannot be reached by the lower voice going up and the upper voice going down more than a third skip.*

$$\begin{aligned}
& i := \max(\mathcal{B}), \forall j \in [0, m-1) \\
& H[0, j+1] = 0 \wedge P[i, j] = 0 \wedge \begin{cases} M_{brut}[i, j] < -4 \wedge IsCfB[i, j] \iff \perp \\ M_{cf}[i, j] < -4 \wedge \neg IsCfB[i, j] \iff \perp \end{cases}
\end{aligned} \tag{C.26}$$

where i stands for the last beat index in a measure.

1.P4 *3V - Successive perfect consonances should be avoided.*

$$\begin{aligned}
& \forall v_1, v_2 \in \{cf, cp_1, cp_2\}, \quad v_1 \neq v_2, \quad \forall j \in [0, m-2): \\
& (H(v_1, v_2)[0, j] \in Cons) \wedge (H(v_1, v_2)[0, j+1] \in Cons_p) \\
& \implies Cost_{succ_p_cons} = 2
\end{aligned} \tag{C.27}$$

1.P5 *3V - Each part starts distant from the lowest stratum.*

This is not a strict rule but an indication to make easier for the composer to have contrary motions. Since this is neither a requirement nor a preference, it can simply be added as a heuristic for the solver. This is discussed in section 4.1.2, on heuristics.

1.P6 *3V - It is prohibited that all parts move in the same direction.*

To prevent this, we need only look at the motions between the parts and the lowest stratum. If one of their motions is contrary, then it is guaranteed that the three voices will not go in the same direction (because at least one is contrary). The same applies if one of the motions is oblique. The problem arises when all the movements are direct, because this would mean that the three voices are going in the same direction. So it was forbidden to have all motions direct at the same time.

$$\begin{aligned}
& \forall j \in [0, m-2): \\
& \bigvee_{p \in \{cf, cp_1, cp_2\}} M(p)[0, j] \neq 2
\end{aligned} \tag{C.28}$$

1.P7 3V - *It is prohibited to use successive ascending sixths on a direct upwards motion.*
 Either the harmonic interval is not a sixth in any of both positions, or one of them is not moving up.

$$\begin{aligned} \forall j \in [1, m-1), \quad \forall v_1, v_2 \in \{cf, cp_1, cp_2\} \text{ where } v_1 \neq v_2, \quad \text{sixth} := \{8, 9\} : \\ (H(v_1, v_2)[0, j-1] \notin \text{sixth}) \vee (H(v_1, v_2)[0, j] \notin \text{sixth}) \\ \vee M(v_1)[0, j] > 0 \vee M(v_2)[0, j] > 0 \end{aligned} \quad (\text{C.29})$$

Constraints of the Second Species

Harmonic Constraints of the Second Species

2.H1 *Thesis harmonies cannot be dissonant.*

There is no constraint to add because it would be a duplicate of rule **1.H1**.

2.H2 *Arsis harmonies cannot be dissonant except if there is a diminution.*

$$\begin{aligned} \forall j \in [0, m-1) \\ IsDim[j] = \begin{cases} \top & \text{if } M^2[0, j] \in \{3, 4\} \wedge M^1[0, j] \in \{1, 2\} \wedge M^1[2, j] \in \{1, 2\} \\ \perp & \text{otherwise} \end{cases} \end{aligned} \quad (\text{C.30})$$

$$\forall j \in [0, m-1) \quad \neg IsCons[2, j] \implies IsDim[j] \quad (\text{C.31})$$

2.H3 and **2.H4** *In the penultimate measure the harmonic interval of perfect fifth must be used for the thesis note if possible. Otherwise, a sixth interval should be used instead.*

$$\begin{aligned} H[0, m-2] \in \{7, 8, 9\} \\ \therefore penulthesis_{cost} = \begin{cases} cost_{penulthesis} & \text{if } H[0, m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (\text{C.32})$$

moreover $\mathcal{C} = \mathcal{C} \cup penulthesis_{cost}$

2.H5 3V - *Major thirds are now allowed in the last chord.*

No need to add a new constraint as this rule is already covered by rules **1.H2** and **1.H3** and **1.H8**.

2.H6 3V - *The half notes must be coherent with respect to the whole notes.*

No need to add a new constraint as this is not an actual rule.

Melodic Constraints of the Second Species

2.M1 *If the two voices are getting so close that there is no contrary motion possible without crossing each other, then the melodic interval of the counterpoint can be an octave leap.*

$$\begin{aligned} \forall j \in [0, m-1), \forall M_{cf}[j] \neq 0 \\ M[0, j] = 12 \implies (H_{abs}[0, j] \leq 4) \wedge (IsCfB[j] \iff M_{cf}[j] > 0) \end{aligned} \quad (\text{C.33})$$

2.M2 *Two consecutive notes cannot be the same. When writing a three-part composition, the 4th-to-last, the 3rd-to-last and the 2nd-to-last may be the same.*

- When dealing with two-part composition:

$$\forall \rho \in \text{positions}(m) \quad N[\rho] \neq N[\rho + 1] \quad (\text{C.34})$$

- When dealing with three-part composition:

$$\begin{aligned} \forall j \in [1, m - 1), \quad j \neq m - 2 : \\ ((N[2, j - 1] \neq N[0, j]) \wedge (N[0, j] \neq N[2, j])) \\ \wedge \\ ((N[2, m - 3] \neq N[0, m - 2]) \vee (N[0, m - 2] \neq N[2, m - 2])) \end{aligned} \quad (\text{C.35})$$

Motion Constraints of the Second Species

2.P1 *If the melodic interval of the counterpoint between the thesis and the arsis is larger than a third, then the motion is perceived based on the arsis note.*

$$\forall j \in [0, m - 1) \quad P_{\text{real}}[j] = \begin{cases} P[2, j] & \text{if } M[0, j] > 4 \\ P[0, j] & \text{otherwise} \end{cases} \quad (\text{C.36})$$

2.P2 *Rule 1.P3 on the battuta octave is adapted such that it focuses on the motion from the note in arsis. This constraint already had an adapted mathematical notation in the chapter of the first species. Note that this constraint would indeed use $P[2]$ and not P_{real} .*

2.P3 *3V - Successive fifths on the downbeat are only allowed when they are separated by a third on the upbeat.*

$$\begin{aligned} \forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ where } p_1 \neq p_2, \quad \forall j \in [0, m - 2) : \\ Cost_{\text{succ}_p\text{-cons}} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[0, j] \notin Cons_p) \vee (H(p_1, p_2)[0, j + 1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[0, j] = 5) \wedge (H(p_1, p_2)[0, j + 1] = 5) \\ & \wedge (H(p_1, p_2)[2, j] = 3) \vee (H(p_1, p_2)[2, j] = 4) \\ 2 & \text{otherwise} \end{cases} \end{aligned} \quad (\text{C.37})$$

Constraints of the Third Species

Harmonic Constraints of the Third Species

3.H1 *If five notes follow each other by joint degrees in the same direction, then the harmonic interval of the third note must be consonant.*

$$\begin{aligned} \forall j \in [0, m - 1) \\ \left(\bigwedge_{i=0}^3 M[i, j] \leq 2 \right) \wedge \left(\bigwedge_{i=0}^3 M_{\text{brut}}[i, j] > 0 \vee \bigwedge_{i=0}^3 M_{\text{brut}}[i, j] < 0 \right) \\ \implies IsCons[2, j] \end{aligned} \quad (\text{C.38})$$

3.H2 *If the third harmonic interval of a measure is dissonant then the second and the fourth interval must be consonant and the third note must be a diminution.*

$$\begin{aligned} & \forall j \in [0, m-1) \\ & IsCons[2, j] \vee (IsCons[1, j] \wedge IsCons[3, j] \wedge IsDim[j]) \end{aligned} \quad (C.39)$$

where $IsDim[j] = \top$ when the 3rd note of the measure j is a diminution.

3.H3 *It is best to avoid the second and third harmonies of a measure to be consonant with a one-degree melodic interval between them.*

$$\begin{aligned} & \forall j \in [0, m-1) \\ & Cambiata_{costs}[j] = \begin{cases} cost_{Cambiata} & \text{if } IsCons[1, j] \wedge IsCons[2, j] \wedge M[1, j] \leq 2 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (C.40)$$

3.H4 *In the penultimate measure, if the cantus firmus is in the upper part, then the harmonic interval of the first note should be a minor third.*

$$\neg IsCfB[m-2] \implies H[0, m-2] = 3 \quad (C.41)$$

3.H5 3V - *The quarter notes must be coherent with respect to the whole notes. There is no constraint to add for this rule, which isn't really a rule.*

3.H6 3V - *If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.*

$$\begin{aligned} & \forall j \in [0, m-1): \\ & (H[1, j] \notin Cons_{h_triad}) \wedge (H[2, j] \notin Cons_{h_triad}) \\ & \iff cost_{harmonic-triad-3rd-species}[j] = 1 \end{aligned} \quad (C.42)$$

Melodic Constraints of the Third Species

3.M1 *Each note and its two beats further peer are preferred to be different.*

$$\begin{aligned} & \forall \rho \in positions(m-2) \\ & MtwoSame_{costs}[i, j] = \begin{cases} cost_{MtwoSame} & \text{if } M^2[\rho] = 0 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (C.43)$$

Motion Constraints of the Third Species

3.P1 *The motion is perceived based on the fourth note.*

This implies that the costs of the motions and the first species constraints on the motions are deducted from $P[3]$.

Constraints of the Fourth Species

Harmonic Constraints of the Fourth Species

4.H1 *Arsis harmonies must be consonant.*

$$\forall j \in [0, m-1) \quad H[2, j] \in Cons \quad (C.44)$$

4.H2 *If the cantus firmus is in the upper part, then no harmonic seventh interval can occur.*

$$\forall j \in [1, m-1) \quad \neg IsCfB[j] \implies H[0, j] \notin \{10, 11\} \quad (C.45)$$

4.H3 and **4.H4** *In the penultimate measure, the harmonic interval of the thesis note must be a major sixth or a minor third depending on the cantus firmus pitch.*

$$H[0, m-2] = \begin{cases} 9 & \text{if } IsCfB[m-2] \\ 3 & \text{otherwise} \end{cases} \quad (C.46)$$

4.H4 *3V - Imperfect consonances are preferred over fifth intervals, which in turn are preferred over octaves.*

This rule is already covered by rule **1.H6** and by the default costs of the search.

Melodic Constraints of the Fourth Species

4.M1 *Arsis half notes should be the same as their next halves in thesis.*

$$\forall j \in [0, m-1) \quad NoSync_{costs} = \begin{cases} cost_{NoSync} & \text{if } M[2, j] \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (C.47)$$

4.M2 *Each arsis note and its two measures further peer are preferred to be different.*

$$\forall j \in [0, m-1) \quad MtwomSame_{costs} = \begin{cases} cost_{MtwomSame} & \text{if } N[2, j] = N[2, j+2] \\ 0 & \text{otherwise} \end{cases} \quad (C.48)$$

Motion Constraints of the Fourth Species

4.P1 *Dissonant harmonies must be followed by the next lower consonant harmony.*

$$\forall j \in [1, m-1) \quad \neg IsCons[0, j] \implies M_{brut}[0, j] \in \{-1, -2\} \quad (C.49)$$

4.P2 *If the cantus firmus is in the lower part then no second harmony can be preceded by a unison/octave harmony.*

$$\forall j \in [1, m-1) \quad IsCfB[j+1] \implies H[2, j] \neq 0 \wedge H[0, j+1] \notin \{1, 2\} \quad (C.50)$$

4.P3 3V - Successive fifths are allowed when using ligatures.

$$\forall v_1, v_2 \in \{cf, cp_1, cp_2\}, \quad \text{with } v_1 \neq v_2, \quad \forall j \in [0, m-2]:$$

$$Cost_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[0, j] \notin Cons_p) \vee (H(p_1, p_2)[0, j+1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[0, j] = 5) \wedge (H(p_1, p_2)[0, j+1] = 5) \\ 2 & \text{otherwise} \end{cases} \quad (C.51)$$

4.P4 3V - Resolving to a fifth is preferred over resolving to an octave.

This is already covered by the rule **4.H5** (prefer fifths over octaves), since preferring fifths over octaves in *all* cases implies preferring to resolve to a fifth rather than to an octave.

4.P5 3V - Stationary movement in the bass implies dissonance in the fourth species part.

$$\forall j \in [0, m-1]:$$

$$M(a)[0, j] \neq 0 \iff H[2, j] \in Cons \quad (C.52)$$

$$M(a)[0, j] = 0 \iff H[2, j] \in Dis$$

4.P6 3V - A note provoking a hidden fifth gets replaced by a rest.

$$\forall j \in [1, m-1]:$$

$$H[0, j] = 7 \wedge P[0, j] = 2 \iff N(0, j-1) = \emptyset \quad (C.53)$$

Constraints of the Fifth Species

The fifth type has a very specific way of working, which cannot be summarised as easily as the other types, as it requires some additional concepts. We here provide the main concepts but for a full understanding of this species' formalisation, please refer to Chapter 7 of T. Wafflard's thesis.

Here is the formal definition of S:

$$\forall \rho \in positions(m)$$

$$S[\rho] = \begin{cases} 0 & \text{if } N[\rho] \text{ is not constrained by any species} \\ 1 & \text{if } N[\rho] \text{ is constrained by the first species} \\ 2 & \text{if } N[\rho] \text{ is constrained by the second species} \\ 3 & \text{if } N[\rho] \text{ is constrained by the third species} \\ 4 & \text{if } N[\rho] \text{ is constrained by the fourth species} \end{cases} \quad (C.54)$$

And the formal definition of IsS_x:

$$\forall x \in \{0, 1, 2, 3, 4\}, \forall \rho \in positions(m)$$

$$IsS_x[\rho] = \begin{cases} \top & \text{if } S[\rho] = x \\ \perp & \text{otherwise} \end{cases} \quad (C.55)$$

5.R1 There must always be a note in thesis and in arsis, except the very first thesis and the very last arsis.

$$\forall j \in [0, m)$$

$$\neg IsS_0[0, j] \quad \text{where } j \neq 0 \quad (C.56)$$

$$\neg IsS_0[2, j] \quad \text{where } j \neq m-1$$

5.R2 The 4th species can only exist in first and third beat.

$$\forall i \in \{1, 3\}, \forall j \in [0, m) \quad \neg IsS_4[i, j] \quad (C.57)$$

5.R3 A 4th species in the third beat necessarily implies a 4th species in the first beat of the following measure and vice versa. The fourth beat should then have no note.

$$\begin{aligned} \forall j \in [0, m - 1) \\ IsS_4[2, j] &\iff IsS_4[0, j + 1] \\ IsS_4[2, j] &\implies IsS_0[3, j] \end{aligned} \quad (C.58)$$

5.R4 A 3rd species cannot be followed by no note.

$$\forall \rho \in positions(m - 1) \quad IsS_3[\rho] \implies \neg IsS_0[\rho + 1] \quad (C.59)$$

5.R5 Only 3rd species and 4th species are used.

$$\forall \rho \in positions(m) \quad \neg IsS_1[\rho] \wedge \neg IsS_2[\rho] \quad (C.60)$$

5.R6 The first and penultimate measures are linked to the 4th species.

$$\begin{aligned} IsS_0[0, 0] \wedge IsS_0[1, 0] \wedge IsS_4[2, 0] \\ IsS_4[0, m - 2] \wedge IsS_0[1, m - 2] \wedge IsS_4[2, m - 2] \end{aligned} \quad (C.61)$$

Generalisation of the Species implications

$$\begin{aligned} \forall x \in \{3, 4\}, \forall cst_x \in Constraints(x), \forall V \in Variables(cst_x) \\ \left(\bigwedge_{\forall v \in V} IsS_x[v_{pos}] \right) \implies cst_x(V) \end{aligned}$$

where $Constraints(x)$ is the set of constraints of the species x ,
and $Variables(cst_x)$ is the set of set of variables concerned by the constraint cst_x ,
and v_{pos} is the position of the v related note in the array N .
(C.62)

Avoiding Multiple Same Final Solutions

$$\forall \rho \in positions(m - 1) \quad IsS_0[\rho] \implies (N[\rho] = N[\rho + 1]) \quad (C.63)$$

Formalisation of Inter-species Rules into Constraints

$$\begin{aligned} \forall j \in [1, m - 1) \\ \neg IsCons[0, j] \wedge IsS_4[0, j] \implies M_{brut}^2[0, j] \in \{-1, -2\} \wedge IsCons[2, j] \end{aligned} \quad (C.64)$$

Parsing of the Species Array in Rhythm

The fifth species is unique as it is the only one that might have a different rythm in each beat. To handle this, a parsing is applied after a solution is found to determine the rhythm of the counterpoint.

This parsing is shown in Figure C.1. The letters correspond to the following: N are the notes the final counterpoint (the one that is returned), Cp are the notes computed by the solver (the array used by the solver), R is the rhythm and S is the S array.

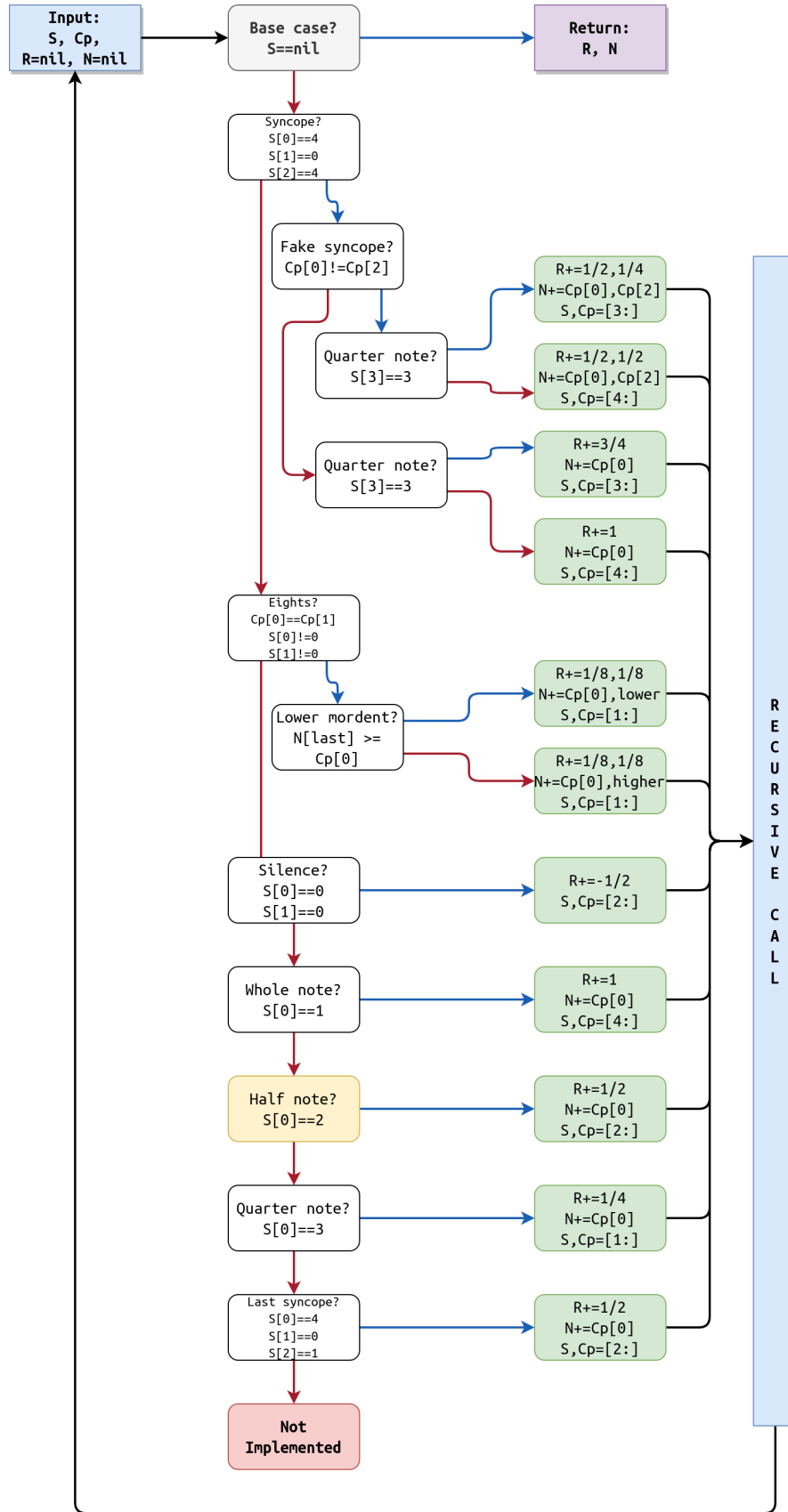


Figure C.1: Rhythm species parser algorithm diagram, 5th species. A red arrow means the test failed while a blue one means it passed.

Appendix D

Code

D.1 FuxCP.lisp

```
1 (in-package :om)
2
3 (defvar *fuxcp-sources-dir* nil)
4 (setf *fuxcp-sources-dir* (make-pathname :directory (append (pathname-directory *
    load-pathname*) '("sources"))))
5
6 (mapc 'compile&load (list
7   (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    package" :type "lisp")
8   (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "utils"
    :type "lisp")
9   (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    constraints" :type "lisp")
10  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "1
    sp-ctp" :type "lisp")
11  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "2
    sp-ctp" :type "lisp")
12  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "3
    sp-ctp" :type "lisp")
13  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "4
    sp-ctp" :type "lisp")
14  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "5
    sp-ctp" :type "lisp")
15  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "3v-ctp
    " :type "lisp")
16  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "cf" :
    type "lisp")
17  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    fuxcp-main" :type "lisp")
18  (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    interface" :type "lisp")
19 ))
20
21
22 (fill-library '(
23   ("Solver" nil (fuxcp::cp-params) nil)
24 ))
25
26 (print "FuxCP Loaded")
```

D.2 package.lisp

```
1 (in-package :om)
2
3 (defvar *FuxCP-path* (make-pathname :directory (append (pathname-directory *
    load-pathname*) (list "FuxCP"))))
4
5 (require-library "GIL")
```

```

6
7 (defpackage :fuxcp
8   (:use "COMMON-LISP" "OM" "CL-USER"))

```

D.3 interface.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard and Anton Lamotte
4 ; Date: June 3, 2023 and January 2024
5 ; This file contains all the cp-params interface.
6 ; That is to say the interface blocks, as well as the global variables updated via
   the interface.
7
8 ;;=====
9 ;;= cp-params OBJECT =
10 ;;=====
11
12 (print "Loading cp-params object...")
13
14 (om::defclass! cp-params ()
15   ;attributes
16   (
17     ; ----- Input cantus firmus -----
18     (cf-voice :accessor cf-voice :initarg :cf-voice :initform nil :documentation "")
19     ; ----- Solver parameters -----
20     (species-param :accessor species-param :initform (list "1st" "1st") :type string
21       :documentation "")
22     (voice-type-param :accessor voice-type-param :initform (list "Really far above"
23       "Above") :type string :documentation "")
24     (min-skips-slider-param :accessor min-skips-slider-param :initform 0 :type
25       integer :documentation "")
26     (borrow-mode-param :accessor borrow-mode-param :initform "Major" :type string :
27       documentation "")
28     ; ----- Output & Stop -----
29     (current-csp :accessor current-csp :initform nil :documentation "")
30     (result-voice :accessor result-voice :initarg :result-voice :initform nil :
31       documentation "")
32     ; ----- Cost order -----
33     (linear-combination :accessor linear-combination :initform "Linear combination"
34       :type string :documentation "")
35   )
36   (:icon 225)
37   (:documentation "This class implements FuxCP.
38     FuxCP is a constraints programming based tool aiming to generate counterpoints
39     based on cantus firmus.")
40 )
41
42 ; the editor for the object
43 (defclass params-editor (om::editorview) ())
44
45 (defmethod om::class-has-editor-p ((self cp-params)) t)
46 (defmethod om::get-editor-class ((self cp-params)) 'params-editor)
47
48 (defmethod om::om-draw-contents ((view params-editor))
49   (let* ((object (om::object view))
50     (om::om-with-focused-view view)
51   )
52 )
53 )

```

```

48 ; this function creates the elements for the main panel
49 (defun make-main-view (editor)
50   ; background colour
51   (om::om-set-bg-color editor om::*om-light-gray-color*)
52 )
53
54 ; To access the melodizer object, (om::object self)
55 (defmethod initialize-instance ((self params-editor) &rest args)
56   ;; do what needs to be done by default
57   (call-next-method) ; start the search by default?
58   (make-interface self)
59 )
60
61 (defun make-interface (editor)
62   (let* (
63     (melodic-subcosts '(
64       (:name "Steps" :value "No cost" :cannot-be-forbidden t :param
65         m-step-cost)
66       (:name "Third skips" :value "Low cost" :param m-third-cost)
67       (:name "Fourth leaps" :value "Low cost" :param m-fourth-cost)
68       (:name "Tritone leaps" :value "Forbidden" :param m-tritone-cost)
69       (:name "Fifth leaps" :value "Medium cost" :param m-fifth-cost)
70       (:name "Sixth leaps" :value "Medium cost" :param m-sixth-cost)
71       (:name "Seventh leaps" :value "Medium cost" :param m-seventh-cost)
72       (:name "Octave leaps" :value "Low cost" :param m-octave-cost)
73     ))
74     (melodic-preferences '( ; care it is a special apostrophe here (needed to
75       evaluate every value that has a comma in this list, and not to take
76       their symbols)
77       (:section "Melodic Preferences" :name "Melodic cost" :display nil :
78         importance "13" :value nil :subcosts ,melodic-subcosts :param
79         m-degrees-cost)
80       ;; Add more cost data as needed
81     ))
82     (motion-subcosts '(
83       (:name "Direct motion" :value "Medium cost" :cannot-be-forbidden t :
84         param dir-motion-cost)
85       (:name "Oblique motion" :value "Low cost" :param obl-motion-cost)
86       (:name "Contrary motion" :value "No cost" :cannot-be-forbidden t :param
87         con-motion-cost)
88     ))
89     (general-preferences '( ; care it is a special apostrophe here (needed to
90       evaluate every value that has a comma in this list, and not to take
91       their symbols)
92       (:section "General preferences" :name "Borrowed notes" :display nil :
93         importance "8" :value "High cost" :param borrow-cost :
94         cannot-be-forbidden t)
95       (:section "General preferences" :name "Harmonic fifths on the downbeat"
96         :display nil :importance "7" :value "Low cost" :param h-fifth-cost :
97         cannot-be-forbidden t)
98       (:section "General preferences" :name "Harmonic octaves on the downbeat"
99         :display nil :importance "5" :value "Low cost" :param h-octave-cost
100        :cannot-be-forbidden t)
101       (:section "General preferences" :name "Successive perfect consonances" :
102         display nil :importance "2" :value "Medium cost" :param
103         succ-p-cons-cost :cannot-be-forbidden t)
104       (:section "General preferences" :name "Repeating notes" :display nil :
105         importance "9" :value "Medium cost" :param variety-cost)
106       (:section "General preferences" :name "No harmonic triad" :display nil :
107         importance "3" :value "Medium cost" :param h-triad-cost :

```

```

92         cannot-be-forbidden t)
93     (:section "General preferences" :name "Direct motion to perf. consonance
        " :display nil :importance "14" :value "Last resort" :param
        direct-move-to-p-cons-cost :cannot-be-forbidden t)
94     (:section "General preferences" :name "Motion cost" :display nil :
        importance "12" :value nil :subcosts ,motion-subcosts :param
        motions-cost)
95     (:section "General preferences" :name "Apply specific penultimate note
        rules" :value "Yes" :special-range ("Yes" "No") :param
        penult-rule-check)
96
97     ;; Add more cost data as needed
98 ))
99
100 (specific-preferences '( ; care it is a special apostrophe here (needed to
        evaluate every value that has a comma in this list, and not to take
        their symbols)
101     (:section "Second species specific pref." :name "Penultimate downbeat
        note is a fifth" :importance "6" :value "Last resort" :param
        penult-sixth-cost :cannot-be-forbidden t)
102     (:section "Third species specific pref." :name "No cambiata" :
        importance "11" :value "High cost" :param non-cambiata-cost :
        cannot-be-forbidden t)
103     (:section "Third species specific pref." :name "Force contrary motion
        after skip" :value "No" :special-range ("Yes" "No") :param
        con-m-after-skip-check)
104     (:section "Third species specific pref." :name "No hamonic triad in 2nd
        /3rd beat" :display nil :importance "4" :value "Medium cost" :param
        h-triad-3rd-species-cost)
105     (:section "Third and fourth species specific pref." :name "Same note in
        downbeat and upbeat" :importance "10" :value "Low cost" :param
        m2-eq-zero-cost)
106     (:section "Fourth species specific pref." :name "No ligatures" :
        importance "1" :value "Last resort" :param no-syncopation-cost :
        cannot-be-forbidden t)
107     (:section "Fifth species specific pref." :name "Many quarters (left) or
        many syncopations (right)" :value 50 :make-slider t :param
        pref-species-slider)
108     ;; Add more cost data as needed
109 ))
110 )
111 ;; Add the cost table to the main view
112 (om::om-add-subviews editor (make-cost-panel editor general-preferences #|
        x-offset:|# 0 #|y-offset:|# 0 #|size:|# 540 #|colour:|# om::*
        azulote*))
113 (om::om-add-subviews editor (make-cost-panel editor melodic-preferences #|
        x-offset:|# 526 #|y-offset:|# 0 #|size:|# 350 #|colour:|# om::*
        azulito*))
114 (om::om-add-subviews editor (make-cost-panel editor specific-preferences #|
        x-offset:|# 1052 #|y-offset:|# 0 #|size:|# 500 #|colour:|# (om::
        make-color-255 230 190 165)))
115 (om::om-add-subviews editor (make-explanation-panel editor #|
        x-offset:|# 0 #|y-offset:|# 541 #|size:|# 160 #|colour:|# (om::
        make-color-255 255 240 120)))
116 (om::om-add-subviews editor (make-search-params-panel editor #|x-offset
        :|# 526 #|y-offset:|# 351 #|size:|# 350 #|colour:|# om::*maq-color*))
117 (om::om-add-subviews editor (make-search-buttons editor #|x-offset
        :|# 1052 #|y-offset:|# 501 #|size:|# 200 #|colour:|# om::*
        workspace-color* melodic-subcosts melodic-preferences motion-subcosts
        general-preferences specific-preferences))
118 )
119

```

```

120 ;; ... (existing code)
121
122
123 editor ; Return the editor
124 )
125
126 (defun make-explanation-panel (editor panel-x-offset panel-y-offset size colour)
127   (let* (
128     ;; Explanation text
129     (explanation-text "First choose the importance of each preference (1 being
                        the most important and 14 being the least important). The solver will
                        give priority to the most important preferences. The cost value is taken
                        into account if two costs have the same importance. ")
130
131     ;; Create a view for the explanation panel
132     (explanation-panel (om::om-make-view 'om::om-view
133                                   :size (om::om-make-point 525 size)
134                                   :position (om::om-make-point panel-x-offset
135                                   panel-y-offset)
136                                   :bg-color colour))
137
138     ;; Create a text element for the explanation
139     (explanation-label (om::om-make-dialog-item 'om::om-static-text
140                                   (om::om-make-point 10 10) (om::om-make-point 500
141                                   800)
142                                   explanation-text
143                                   ))
144
145     ;; Add the text element to the explanation panel
146     (om::om-add-subviews explanation-panel explanation-label)
147
148     (om::om-add-subviews explanation-panel
149       (om::om-make-dialog-item
150         'om::om-static-text
151         (om::om-make-point 10 100)
152         (om::om-make-point 400 20)
153         "If two costs are ranked the same, perform between them a:")
154       (om::om-make-dialog-item
155         'om::pop-up-menu
156         (om::om-make-point 215 120)
157         (om::om-make-point 280 20)
158         "Linear combination"
159         :range (list "Linear combination" "Maximum minimisation")
160         :value (linear-combination (om::object editor))
161         :di-action #'(lambda (cost)
162                       (setf (linear-combination (om::object editor)) (nth (om::
163                                   om-get-selected-item-index cost) (om::om-get-item-list
164                                   cost))))
165       )
166     )
167
168     ;; Add the explanation panel to the main view
169     (om::om-add-subviews editor explanation-panel)
170   )
171
172 (defun make-cost-panel (editor cost-data panel-x-offset panel-y-offset y-size colour)
173   (let* (
174     (cost-table (om::om-make-view 'om::om-view

```

```

175         :size (om::om-make-point 525 y-size)
176         :position (om::om-make-point panel-x-offset panel-y-offset)
177         :bg-color colour))
178
179     (importance-column (om::om-make-dialog-item 'om::om-static-text
180         (om::om-make-point 275 0) (om::om-make-point 150 20)
181         "Importance"
182         :font om::*om-default-font2b*
183     ))
184
185     (value-column (om::om-make-dialog-item 'om::om-static-text
186         (om::om-make-point 400 0) (om::om-make-point 150 20) "
187         Value"
188         :font om::*om-default-font2b*
189     ))
190
191     ;; Add header columns to the cost table
192     (om::om-add-subviews cost-table importance-column value-column)
193
194     ;; Populate the cost data dynamically
195     (let (
196         (current-section nil)
197         (y-offset 0)
198     )
199         (loop for index from 0 below (length cost-data)
200             do
201                 (let* ((cost (nth index cost-data))
202                     (section (getf cost :section))
203                     (y-position (+ y-offset (* 45 index)))
204                     (name (if (getf cost :display) (getf cost :display) (getf cost :
205                         name)))
206                     (importance (getf cost :importance))
207                     (value (getf cost :value))
208                     (is-new-section (not (string= current-section section)))
209                 )
210                 ;; Add subsection header if it's a new section
211                 (when is-new-section
212                     (let ((section-label (om::om-make-dialog-item 'om::
213                         om-static-text
214                             (om::om-make-point 15 y-position) (om::
215                             om-make-point 300 20) section
216                             :font om::*om-default-font2b*)))
217                         (om::om-add-subviews cost-table section-label))
218                     (setf current-section section)
219                     (incf y-offset 35)
220                     (incf y-position 35)
221                 )
222                 ;; Add the row to the cost table
223                 (let* (
224                     (name-label (om::om-make-dialog-item 'om::om-static-text
225                         (om::om-make-point 25 y-position) (om::om-make-point
226                         500 20) name))
227                     (importance-popup (om::om-make-dialog-item 'om::pop-up-menu
228                         (om::om-make-point 275 (- y-position 7)) (om
229                         ::om-make-point 70 20)
230                         (format nil "~A" importance)
231                         :value importance
232                         :range (importance-range)
233                         :di-action #'(lambda (x)
234                             (setf (getf cost :importance) (nth (om::
235                             om-get-selected-item-index x) (om::

```

```

230         om-get-item-list x)))
231         (print (getf cost :importance))
232     )
233 )
234 (value-popup (if (getf cost :make-slider)
235     (make-slider cost y-position)
236     (om::om-make-dialog-item 'om::pop-up-menu
237         (om::om-make-point 345 (- y-position 7)) (om::
238             om-make-point 150 20)
239         (format nil "~A" value)
240         :value value
241         :range (if (getf cost :special-range)
242             (getf cost :special-range)
243             (value-range (getf cost :cannot-be-forbidden)
244                 ))
245         )
246         :di-action #'(lambda (x)
247             (setf (getf cost :value) (nth (om::
248                 om-get-selected-item-index x) (om::
249                     om-get-item-list x)))
250             (print (getf cost :value))
251         ))
252     ))
253 )
254 (cond
255     ((and value importance) (om::om-add-subviews cost-table
256         name-label importance-popup value-popup))
257     (value (om::om-add-subviews cost-table name-label
258         value-popup))
259     (importance (om::om-add-subviews cost-table name-label
260         importance-popup))
261 )
262 ) ; end of row
263 (print (getf cost :subcosts))
264 (print (length (getf cost :subcosts)))
265 (if (getf cost :subcosts)
266     (loop for index from 0 below (length (getf cost :subcosts))
267         do
268             (let* ((cost (nth index (getf cost :subcosts)))
269                 (y-position (+ y-position (* 35 (+ 1 index))))
270                 (name (concatenate 'string "|---" (getf cost :name)))
271                 )
272                 (value (getf cost :value))
273                 )
274             (incf y-offset 35)
275             ;; Add the row to the cost table
276             (let* (
277                 (name-label (om::om-make-dialog-item 'om::
278                     om-static-text
279                     (om::om-make-point 50 y-position) (om::
280                         om-make-point 250 20) name))
281                 (value-popup (om::om-make-dialog-item 'om::
282                     pop-up-menu
283                     (om::om-make-point 345 (- y-position
284                         7)) (om::om-make-point 150 20)
285                     (format nil "~A" value)
286                     :value value
287                     :range (value-range (getf cost :
288                         cannot-be-forbidden))
289                     :di-action #'(lambda (x)

```

```

278                                     (setf (getf cost :value) (nth (
                                     om::
                                     om-get-selected-item-index x
                                     ) (om::om-get-item-list x)))
279                                     (print (getf cost :value))
280                                 )
281                             )
282                         )
283                     )
284                 (om::om-add-subviews cost-table name-label
                    value-popup)
285             ) ; end of subcost row
286         ) ; end of subcost
287     ) ; end of subcost loop
288 )
289 ) ; end of cost
290 ) ; end of loop
291 )
292 cost-table
293 ))
294
295 (defun make-search-params-panel (editor panel-x-offset panel-y-offset y-size colour)
296     (let* (
297         (search-params-panel (om::om-make-view 'om::om-view
298             :size (om::om-make-point 525 y-size)
299             :position (om::om-make-point panel-x-offset panel-y-offset)
300             :bg-color colour))
301         )
302         (om::om-add-subviews
303             search-params-panel
304             (om::om-make-dialog-item
305                 'om::om-static-text
306                 (om::om-make-point 15 0)
307                 (om::om-make-point 200 20)
308                 "Solver Configuration"
309                 :font om::*om-default-font2b*
310             )
311
312             (om::om-make-dialog-item
313                 'om::om-static-text
314                 (om::om-make-point 25 30)
315                 (om::om-make-point 150 20)
316                 "First voice species"
317             )
318
319             (om::om-make-dialog-item
320                 'om::pop-up-menu
321                 (om::om-make-point 275 25)
322                 (om::om-make-point 220 20)
323                 "First voice species"
324                 :range (list "1st" "2nd" "3rd" "4th" "5th")
325                 :value (first (species-param (om::object editor)))
326                 :di-action #'(lambda (cost)
327                     (setf (first (species-param (om::object editor))) (nth (om::
328                         om-get-selected-item-index cost) (om::om-get-item-list cost)))
329                 )
330             )
331
332             (om::om-make-dialog-item
333                 'om::om-static-text
334                 (om::om-make-point 25 80)
335                 (om::om-make-point 150 20)
336                 "First voice range"

```

```

336 )
337
338 (om::om-make-dialog-item
339 'om::pop-up-menu
340 (om::om-make-point 275 75)
341 (om::om-make-point 220 20)
342 "Voice range"
343 :range (list "Really far above" "Far above" "Above" "Same range" "Below"
344           "Far below" "Really far below")
345 :value (first (voice-type-param (om::object editor)))
346 :di-action #'(lambda (cost)
347               (setf (first (voice-type-param (om::object editor))) (nth (om::
348                               om-get-selected-item-index cost) (om::om-get-item-list cost)))
349               )
350 )
351
352 (om::om-make-dialog-item
353 'om::om-static-text
354 (om::om-make-point 25 130)
355 (om::om-make-point 150 20)
356 "Second voice species"
357 )
358
359 (om::om-make-dialog-item
360 'om::pop-up-menu
361 (om::om-make-point 275 125)
362 (om::om-make-point 220 20)
363 "Second voice species"
364 :range (list "None" "1st" "2nd" "3rd" "4th" "5th")
365 :value (second (species-param (om::object editor)))
366 :di-action #'(lambda (cost)
367               (setf (second (species-param (om::object editor))) (nth (om::
368                               om-get-selected-item-index cost) (om::om-get-item-list cost)))
369               )
370 )
371
372 (om::om-make-dialog-item
373 'om::om-static-text
374 (om::om-make-point 25 180)
375 (om::om-make-point 150 20)
376 "Second voice range"
377 )
378
379 (om::om-make-dialog-item
380 'om::pop-up-menu
381 (om::om-make-point 275 175)
382 (om::om-make-point 220 20)
383 "Second voice range"
384 :range (list "Really far above" "Far above" "Above" "Same range" "Below"
385           "Far below" "Really far below")
386 :value (second (voice-type-param (om::object editor)))
387 :di-action #'(lambda (cost)
388               (setf (second (voice-type-param (om::object editor))) (nth (om::
389                               om-get-selected-item-index cost) (om::om-get-item-list cost)))
390               )
391 )
392
393 (om::om-make-dialog-item
394 'om::om-static-text
395 (om::om-make-point 25 230)
396 (om::om-make-point 150 20)
397 "Borrowing mode"
398 )

```

```

394      (om::om-make-dialog-item
395      'om::pop-up-menu
396      (om::om-make-point 275 225)
397      (om::om-make-point 220 20)
398      "Borrowing mode"
399      :range (list "None" "Major" "Minor")
400      :value (borrow-mode-param (om::object editor))
401      :di-action #'(lambda (cost)
402      (setf (borrow-mode-param (om::object editor)) (nth (om::
403      om-get-selected-item-index cost) (om::om-get-item-list cost)))
404      )
405      )
406
407      (om::om-make-dialog-item
408      'om::om-static-text
409      (om::om-make-point 25 280)
410      (om::om-make-point 150 20)
411      "Minimum % of skips"
412      )
413
414      (om::om-make-dialog-item
415      'om::om-slider
416      (om::om-make-point 275 275)
417      (om::om-make-point 220 20)
418      "Minimum % of skips"
419      :range '(0 100)
420      :increment 1
421      :value (min-skips-slider-param (om::object editor))
422      :di-action #'(lambda (s)
423      (setf (min-skips-slider-param (om::object editor)) (om::
424      om-slider-value s))
425      )
426      )
427      search-params-panel
428    )
429  )
430
431  (defun make-search-buttons (editor panel-x-offset panel-y-offset y-size colour
432    melodic-subcosts melodic-preferences motion-subcosts general-preferences
433    specific-preferences)
434    (let* (
435      (search-buttons (om::om-make-view 'om::om-view
436      :size (om::om-make-point 525 y-size)
437      :position (om::om-make-point panel-x-offset panel-y-offset)
438      :bg-color colour))
439      (om::om-add-subviews
440      search-buttons
441      (om::om-make-dialog-item
442      'om::om-static-text
443      (om::om-make-point 187 25)
444      (om::om-make-point 150 20)
445      "Solver Launcher"
446      :font om::*om-default-font3b*
447      )
448
449      (om::om-make-dialog-item
450      'om::om-button
451      (om::om-make-point 97 60) ; position (horizontal, vertical)
452      (om::om-make-point 160 20) ; size (horizontal, vertical)
453      "Save Config"

```

```

453 :di-action #'(lambda (b)
454   (if (null (cf-voice (om::object editor))); if the problem is not
        initialized
455     (error "No voice has been given to the solver. Please set a
        cantus firmus into the second input and try again.")
456   )
457
458   (set-global-cf-variables
459     (cf-voice (om::object editor))
460     (borrow-mode-param (om::object editor))
461   )
462   (defparameter *params* (make-hash-table))
463   ;; set melodic parameters
464   (dolist (subcost melodic-subcosts)
465     (setparam-cost (getf subcost :param) (getf subcost :value))
466   )
467
468   ;; set general costs
469   (dolist (cost general-preferences)
470     (if (equal (getf cost :param) 'motions-cost)
471         nil ; motions-cost is treated by the subcosts
472         (if (equal (getf cost :param) 'penult-rule-check)
473             (setparam-yes-no (getf cost :param) (getf cost :value))
              ; penult-rule-check is a yes no
474             (setparam-cost (getf cost :param) (getf cost :value)) ;
              else
475             )
476         )
477   )
478
479   ;; set motions costs
480   (dolist (subcost motion-subcosts)
481     (setparam-cost (getf subcost :param) (getf subcost :value))
482   )
483
484   ;; set species specific costs
485   (dolist (cost specific-preferences)
486     (if (equal (getf cost :param) 'pref-species-slider)
487         (setparam-slider (getf cost :param) (getf cost :value)) ; it
            is a slider
488         (if (equal (getf cost :param) 'con-m-after-skip-check)
489             (setparam-yes-no (getf cost :param) (getf cost :value))
              ; it is a yes-no
490             (setparam-cost (getf cost :param) (getf cost :value)) ;
              else
491             )
492         )
493   )
494
495   ;; set search parameters
496   (setparam-slider 'min-skips-slider (min-skips-slider-param (om::
        object editor)))
497   (setparam 'borrow-mode (borrow-mode-param (om::object editor)))
498
499
500   ;; preferences for the cost order
501   (defparameter *cost-preferences* (make-hash-table))
502   (dolist (current-list (list general-preferences specific-preferences
        melodic-preferences))
503     (dolist (cost current-list)
504       (if (getf cost :importance)
505           (setf (gethash (getf cost :param) *cost-preferences*) (
        getf cost :importance))
506       )
507     )

```

```

506         )
507     )
508 )
509
510 (if (string= "Linear combination" (linear-combination (om::object
511 editor)))
512     (setf *linear-combination t)
513     (setf *linear-combination nil)
514 )
515
516 (setf species-integer-list (convert-to-species-integer-list (
517 species-param (om::object editor))))
518 (setf *voices-types (convert-to-voice-integer-list (voice-type-param
519 (om::object editor))))
520 (setf (current-csp (om::object editor)) (fux-cp species-integer-list
521 ))
522 )
523 )
524
525 (om::om-make-dialog-item
526 'om::om-button
527 (om::om-make-point 97 100) ; position
528 (om::om-make-point 160 20) ; size
529 "Next Solution"
530 :di-action #'(lambda (b)
531 (if (typep (current-csp (om::object editor)) 'null); if the problem
532 is not initialized
533 (error "The problem has not been initialized. Please set the
534 input and press Start.")
535 )
536 (print "Searching for the next solution")
537 ;reset the boolean because we want to continue the search
538 (setparam 'is-stopped nil)
539 ;get the next solution
540 (mp:process-run-function ; start a new thread for the execution of
541 the next method
542 "solver-thread" ; name of the thread, not necessary but useful
543 for debugging
544 nil ; process initialization keywords, not needed here
545 (lambda () ; function to call
546 (setf
547 (result-voice (om::object editor))
548 (search-next-fux-cp (current-csp (om::object editor))))
549 )
550 (om::openeditorframe ; open a voice window displaying the
551 solution
552 (om::omNG-make-new-instance (result-voice (om::object
553 editor)) "Current solution")
554 )
555 )
556 )
557 )
558 )
559
560 (om::om-make-dialog-item
561 'om::om-button
562 (om::om-make-point 262 100) ; position
563 (om::om-make-point 160 20) ; size
564 "Best Solution"
565 :di-action #'(lambda (b)
566 (if (typep (current-csp (om::object editor)) 'null); if the problem
567 is not initialized

```

```

558         (error "The problem has not been initialized. Please set the
559             input and press Start.")
560     )
561     (print "Searching for the best solution")
562     ;reset the boolean because we want to continue the search
563     (setparam 'is-stopped nil)
564     ;get the next solution
565     (mp:process-run-function ; start a new thread for the execution of
566         the next method
567         "solver-thread" ; name of the thread, not necessary but useful
568         for debugging
569         nil ; process initialization keywords, not needed here
570         (lambda () ; function to call
571             (let ((check 1) (result nil))
572                 (loop while check do
573                     (setf result (search-next-fux-cp (current-csp (om::
574                         object editor))))
575                     (if result (setf (result-voice (om::object editor))
576                         result) (setf check nil))
577                 )
578             )
579         )
580     )
581     ;(om::openeditorframe ; open a voice window displaying the
582     ;    solution
583     ;    (om::omNG-make-new-instance (result-voice (om::object
584     ;        editor)) "Current solution")
585     ;)
586     )
587     )
588     (om::om-make-dialog-item
589     'om::om-button
590     (om::om-make-point 262 60) ; position (horizontal, vertical)
591     (om::om-make-point 160 20) ; size (horizontal, vertical)
592     "Stop"
593     :di-action #'(lambda (b)
594         (setparam 'is-stopped t)
595     )
596     )
597     )
598     search-buttons
599 )
600 )
601
602 (defun make-slider (cost y-position)
603     (om::om-make-dialog-item
604     'om::om-slider
605     (om::om-make-point 350 (- y-position 3))
606     (om::om-make-point 150 20)
607     "5th: Preference to a lot of quarters [left] OR a lot of syncopations [right]"
608     :range '(0 100)
609     :increment 1
610     :value (getf cost :value)
611     :di-action #'(lambda (s)
612         (setf (getf cost :value) (om::om-slider-value s))
613         (print (getf cost :value))
614     )
615     )
616 )
617
618 ; return the list of available costs for the preferences
619 ; @is-required: if true, "Forbidden" is removed

```

```

614 (defun value-range (&optional (is-required nil))
615   (let (
616     (costs (list "No cost" "Low cost" "Medium cost" "High cost" "Last resort" "
        Cost prop. to length" "Forbidden")))
617   )
618   (if is-required
619     (butlast costs)
620     costs
621   )
622 )
623 )
624
625 (defun importance-range ()
626   (mapcar #'(lambda (x) (format nil "~A" x)) (loop for i from 1 to 14 collect i))
627 )
628
629 ; set the value @v in the hash table @h with key @k
630 (defun seth (h k v)
631   (setf (gethash k h) v)
632 )
633
634 ; set the value @v in the parameters with key @k
635 (defun setparam (k v)
636   (seth *params* k v)
637 )
638
639 (defun setparam-yes-no (k v)
640   (let ((converted (if (string= "Yes" v)
641                         t
642                         nil)))
643     (setparam k converted)
644   )
645 )
646
647 ; set the cost-converted value @of v in the parameters with key @k
648 (defun setparam-cost (k v)
649   (setparam k (convert-to-cost-integer v))
650 )
651
652 ; set the species-converted value @of v in the parameters with key @k
653 (defun setparam-species (k v)
654   (setparam k (convert-to-species-integer v))
655 )
656
657 ; set the slider-converted value @of v in the parameters with key @k
658 (defun setparam-slider (k v)
659   (setparam k (convert-to-percent v))
660 )
661
662 ; convert a cost to an integer
663 (defun convert-to-cost-integer (param)
664   (cond
665     ((equal param "No cost") 0)
666     ((equal param "Low cost") 1)
667     ((equal param "Medium cost") 2)
668     ((equal param "High cost") 4)
669     ((equal param "Last resort") 8)
670     ((equal param "Cost prop. to length") (* 2 *cf-len))
671     ((equal param "Forbidden") (* 64 *cf-len))
672   )
673 )
674
675 ; convert a species to an integer

```

```

676 (defun convert-to-species-integer-list (param-list)
677   (let (
678     (species-list '())
679   )
680     (dolist (param param-list)
681       (progn
682         (cond
683           ((equal param "1st") (setf species-list (append species-list '(1))))
684           ((equal param "2nd") (setf species-list (append species-list '(2))))
685           ((equal param "3rd") (setf species-list (append species-list '(3))))
686           ((equal param "4th") (setf species-list (append species-list '(4))))
687           ((equal param "5th") (setf species-list (append species-list '(5))))
688           ((equal param "None") nil)
689         )
690       ))
691     (setq *N-COUNTERPOINTS (length species-list))
692     (setq *N-PARTS (+ 1 (length species-list)))
693     species-list
694   )
695 )
696
697 ;; convert the string for the voice type to an integer
698 ;; belong to {"Really far above" "Far above" "Above" "Same range" "Below" "Far below"
699 ;; "Really far below"}
700 ;; convert to {-3 -2 -1 0 1 2 3}
701 (defun convert-to-voice-integer-list (params)
702   (let ((integer-list (make-list *N-COUNTERPOINTS :initial-element nil))) (loop
703     for i from 0 below *N-COUNTERPOINTS do
704       (cond
705         ((equal (nth i params) "Really far above") (setf (nth i integer-list) 3))
706         )
707         ((equal (nth i params) "Far above") (setf (nth i integer-list) 2))
708         ((equal (nth i params) "Above") (setf (nth i integer-list) 1))
709         ((equal (nth i params) "Same range") (setf (nth i integer-list) 0))
710         ((equal (nth i params) "Below") (setf (nth i integer-list) -1))
711         ((equal (nth i params) "Far below") (setf (nth i integer-list) -2))
712         ((equal (nth i params) "Really far below") (setf (nth i integer-list)
713           -3))
714       )
715     )
716     integer-list
717   )
718 )
719
720 ; convert a slider value to a percentage
721 (defun convert-to-percent (param)
722   (float (/ param 100))
723 )
724
725 ; convert a mode to an integer
726 (defun convert-to-mode-integer (param tone)
727   (cond
728     ((equal param "Major") (mod tone 12))
729     ((equal param "Minor") (mod (+ tone 3) 12))
730     ((equal param "None") nil)
731   )
732 )
733
734 ; define all the global variables
735 (defun set-global-cf-variables (cantus-firmus borrow-mode)
736   (defparameter *prev-sol-check nil)
737   (defparameter rhythmic+pitch nil)
738   (defparameter rhythmic-om nil)

```

```

735 (defparameter pitches-om nil)
736 ; get the tonalite of the cantus firmus
737 (defparameter *tonalite-offset (get-tone-offset cantus-firmus))
738 ; get the *scale of the cantus firmus
739 (defparameter *scale (build-scaleset (get-scale) *tonalite-offset))
740 ; *chromatic *scale
741 (defparameter *chromatic-scale (build-scaleset (get-scale "chromatic") *
    tonalite-offset))
742 ; get the first note of each chord of the cantus firmus
743 (defparameter *cf (mapcar #'first (to-pitch-list (om::chords cantus-firmus))))
744 ; get the tempo of the cantus firmus
745 (defparameter *cf-tempo (om::tempo cantus-firmus))
746 ; get the first note of the cantus firmus ;; just used for the moment
747 (defparameter *tone-pitch-cf (first *cf))
748 ; get the borrowed scale of the cantus firmus, i.e. some notes borrowed from the
    natural scale of the tone (useful for modes)
749 (setq mode-param (convert-to-mode-integer borrow-mode *tone-pitch-cf))
750 (if mode-param
751     (defparameter *borrowed-scale (build-scaleset (get-scale "borrowed")
    mode-param))
752     (defparameter *borrowed-scale (list)))
753 )
754 ; get notes that are not in the natural scale of the tone
755 (defparameter *off-scale (set-difference *chromatic-scale *scale))
756 ; length of the cantus firmus
757 (defparameter *cf-len (length *cf))
758 ; *cf-last-index is the number of melodic intervals in the cantus firmus
759 (defparameter *cf-last-index (- *cf-len 1))
760 ; *cf-penult-index is the number of larger (n -> n+2) melodic intervals in the
    cantus firmus
761 (defparameter *cf-penult-index (- *cf-len 2))
762 ; COST_UB is the upper bound of the cost function
763 (defparameter COST_UB (* *cf-len 20))
764 ; *N-COUNTERPOINTS is the number of counterpoints in the counterpoint
765 (defparameter *N-COUNTERPOINTS -1) ; will be defined when parsing the input
766 )

```

D.4 fuxcp-main.lisp

```

1  (in-package :fuxcp)
2
3  ; Author: Thibault Wafflard and Anton Lamotte
4  ; Date: June 3, 2023 and January 2024
5  ; This file contains the functions that:
6  ;   - dispatch to the right species functions
7  ;   - set the global variables of the CSP
8  ;   - manage the search for solutions
9
10 (print "Loading fux-cp...")
11
12 ; get the value at key @k in the hash table @h as a list
13 (defun geth-dom (h k)
14   (list (gethash k h)))
15 )
16
17 ; get the value at key @k in the parameters table as a list
18 (defun getparam-val (k)
19   (geth-dom *params* k))
20 )
21
22 ; get the value at key @k in the parameters table as a domain

```

```

23 (defun getparam-dom (k)
24   (list 0 (getparam k))
25 )
26
27 ; get the value at key @k in the parameters table
28 (defun getparam (k)
29   (gethash k *params*)
30 )
31
32 ; get if borrow-mode param is allowed
33 (defun is-borrow-allowed ()
34   (not (equal (getparam 'borrow-mode) "None")))
35 )
36
37
38 ; define all the constants that are going to be used
39 (defun define-global-constants ()
40   ; Number of costs added
41   (defparameter *n-cost-added* 0)
42
43   ;; CONSTANTS
44   ; Motion types
45   (defparameter DIRECT 2)
46   (defparameter OBLIQUE 1)
47   (defparameter CONTRARY 0)
48
49   ; Integer constants (to represent costs or intervals)
50   ; 0 in IntVar
51   (defparameter ZERO (gil::add-int-var-dom *sp* (list 0)))
52   ; 1 in IntVar
53   (defparameter ONE (gil::add-int-var-dom *sp* (list 1)))
54   ; 3 in IntVar (minor third)
55   (defparameter THREE (gil::add-int-var-dom *sp* (list 3)))
56   ; 9 in IntVar (major sixth)
57   (defparameter NINE (gil::add-int-var-dom *sp* (list 9)))
58
59   (defparameter CANTUS_FIRMUS (gil::add-int-var-array *sp* *cf-len* 0 120))
60   (dotimes (i *cf-len*)
61     (gil::g-rel *sp* (nth i CANTUS_FIRMUS) gil::IRT_EQ (nth i *cf*)))
62   )
63
64   ; Boolean constants
65   ; 0 in BoolVar
66   (defparameter FALSE (gil::add-bool-var *sp* 0 0))
67   ; 1 in BoolVar
68   (defparameter TRUE (gil::add-bool-var *sp* 1 1))
69
70   ; Intervals constants
71   ; perfect consonances intervals
72   (defparameter P_CONS (list 0 7))
73   ; imperfect consonances intervals
74   (defparameter IMP_CONS (list 3 4 8 9))
75   ; all consonances intervals
76   (defparameter ALL_CONS (union P_CONS IMP_CONS))
77   ; harmonic triad intervals
78   (defparameter H_TRIAD (list 0 3 4 7))
79   ; major harmonic triad intervals
80   (defparameter MAJ_H_TRIAD (list 0 4 7))
81   ; dissonances intervals
82   (defparameter DIS (list 1 2 5 6 10 11))
83   ; penultimate intervals, i.e. minor third and major sixth
84   (defparameter PENULT_CONS (list 0 3 9))
85   ; penultimate thesis intervals, i.e. perfect fifth and sixth

```

```

86 (defparameter PENULT_THESIS (list 0 7 8 9))
87 ; penultimate 1st quarter note intervals, i.e. minor third, major sixth and
    octave/unisson
88 (defparameter PENULT_1Q (list 0 3 8))
89 ; penultimate syncope intervals, i.e. seconds and sevenths
90 (defparameter PENULT_SYNCOPE (list 0 1 2 10 11))
91 ; penultimate intervals for three parts, i.e. minor third and major sixth + the
    perfect consonances
92 (defparameter PENULT_CONS_3P (list 0 3 7 9))
93
94 ; P_CONS in IntVar
95 (defparameter P_CONS_VAR (gil::add-int-var-const-array *sp* P_CONS))
96 ; IMP_CONS in IntVar
97 (defparameter IMP_CONS_VAR (gil::add-int-var-const-array *sp* IMP_CONS))
98 ; ALL_CONS in IntVar
99 (defparameter ALL_CONS_VAR (gil::add-int-var-const-array *sp* ALL_CONS))
100 ; H_TRIAD in IntVar
101 (defparameter H_TRIAD_VAR (gil::add-int-var-const-array *sp* H_TRIAD))
102 ; MAJ_H_TRIAD in IntVar
103 (defparameter MAJ_H_TRIAD_VAR (gil::add-int-var-const-array *sp* MAJ_H_TRIAD))
104 ; DIS in IntVar
105 (defparameter DIS_VAR (gil::add-int-var-const-array *sp* DIS))
106 ; PENULT_CONS in IntVar
107 (defparameter PENULT_CONS_VAR (gil::add-int-var-const-array *sp* PENULT_CONS))
108 ; PENULT_THESIS in IntVar
109 (defparameter PENULT_THESIS_VAR (gil::add-int-var-const-array *sp* PENULT_THESIS
    ))
110 ; PENULT_1Q in IntVar
111 (defparameter PENULT_1Q_VAR (gil::add-int-var-const-array *sp* PENULT_1Q))
112 ; PENULT_SYNCOPE in IntVar
113 (defparameter PENULT_SYNCOPE_VAR (gil::add-int-var-const-array *sp*
    PENULT_SYNCOPE))
114 ; PENULT_CONS_3P in IntVar
115 (defparameter PENULT_CONS_3P_VAR (gil::add-int-var-const-array *sp*
    PENULT_CONS_3P))
116
117 ; *cf-brut-intervals is the list of brut melodic intervals in the cantus firmus
118 (setq *cf-brut-m-intervals (gil::add-int-var-array *sp* *cf-last-index -127 127)
    )
119 ; array representing the brut melodic intervals of the cantus firmus
120 (create-cf-brut-m-intervals *cf *cf-brut-m-intervals)
121
122 ;; COSTS
123 ;; Melodic costs
124 (defparameter *m-step-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-step-cost)))
125 (defparameter *m-third-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-third-cost)))
126 (defparameter *m-fourth-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-fourth-cost)))
127 (defparameter *m-tritone-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-tritone-cost)))
128 (defparameter *m-fifth-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-fifth-cost)))
129 (defparameter *m-sixth-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-sixth-cost)))
130 (defparameter *m-seventh-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-seventh-cost)))
131 (defparameter *m-octave-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m-octave-cost)))
132
133 ;; General costs

```

```

134 (defparameter *borrow-cost* (gil::add-int-var-dom *sp* (getparam-val '
    borrow-cost)))
135 (defparameter *h-fifth-cost* (gil::add-int-var-dom *sp* (getparam-val '
    h-fifth-cost)))
136 (defparameter *h-octave-cost* (gil::add-int-var-dom *sp* (getparam-val '
    h-octave-cost)))
137 (defparameter *con-motion-cost* (gil::add-int-var-dom *sp* (getparam-val '
    con-motion-cost)))
138 (defparameter *obl-motion-cost* (gil::add-int-var-dom *sp* (getparam-val '
    obl-motion-cost)))
139 (defparameter *dir-motion-cost* (gil::add-int-var-dom *sp* (getparam-val '
    dir-motion-cost)))
140 ; 3v general costs
141 (defparameter *succ-p-cons-cost* (gil::add-int-var-dom *sp* (getparam-val '
    succ-p-cons-cost)))
142 (defparameter *variety-cost* (gil::add-int-var-dom *sp* (getparam-val '
    variety-cost)))
143 (defparameter *h-triad-cost* (gil::add-int-var-dom *sp* (getparam-val '
    h-triad-cost)))
144 (defparameter *direct-move-to-p-cons-cost* (gil::add-int-var-dom *sp* (
    getparam-val 'direct-move-to-p-cons-cost)))
145 ;; Species specific costs
146 (defparameter *penult-sixth-cost* (gil::add-int-var-dom *sp* (getparam-val '
    penult-sixth-cost)))
147 (defparameter *non-cambiata-cost* (gil::add-int-var-dom *sp* (getparam-val '
    non-cambiata-cost)))
148 (defparameter *m2-eq-zero-cost* (gil::add-int-var-dom *sp* (getparam-val '
    m2-eq-zero-cost)))
149 (defparameter *no-syncopation-cost* (gil::add-int-var-dom *sp* (getparam-val '
    no-syncopation-cost)))
150 ; 3v species specific costs
151 (defparameter *h-triad-3rd-species-cost* (gil::add-int-var-dom *sp* (
    getparam-val 'h-triad-3rd-species-cost)))
152
153 ;; Params domains
154 (defparameter *motions-domain* ; equal to all possible values of the motions
    cost, plus zero
    (remove-duplicates (append
155        (mapcar (lambda (x) (getparam x))
156            (list 'con-motion-cost 'obl-motion-cost 'dir-motion-cost)
157        )
158        (list 0)
159    ))
160 )
161
162
163 ; To make the code more readable
164 (defparameter 3v-1sp 6)
165 (defparameter 3v-2sp 7)
166 (defparameter 3v-3sp 8)
167 (defparameter 3v-4sp 9)
168 (defparameter 3v-5sp 10)
169 )
170
171 (defclass stratum-class () (
172     ; represents a stratum, it is a simplified part-class
173     (solution-array :accessor solution-array :initarg :solution-array :initform nil)
174     (solution-len :accessor solution-len :initarg :solution-len :initform nil)
175
176     (notes :accessor notes :initarg :notes :initform
177         (list
178             (gil::add-int-var-array *sp* *cf-len 0 120)
179             (gil::add-int-var-array *sp* *cf-len 0 120)
180             (gil::add-int-var-array *sp* *cf-len 0 120)

```

```

181         (gil::add-int-var-array *sp* *cf-len 0 120)
182     )
183     ) ; represents the notes of the counterpoint
184     (h-intervals :accessor h-intervals :initarg :h-intervals :initform
185         (list
186             (gil::add-int-var-array *sp* *cf-len 0 11)
187             (gil::add-int-var-array *sp* *cf-len 0 11)
188             (gil::add-int-var-array *sp* *cf-len 0 11)
189             (gil::add-int-var-array *sp* *cf-len 0 11)
190         )
191     )
192     (is-p-cons-arr :accessor is-p-cons-arr :initarg :is-p-cons-arr :initform nil)
193     (m-intervals :accessor m-intervals :initarg :m-intervals :initform (list (gil::
194         add-int-var-array *sp* *cf-last-index 0 12) nil nil nil))
195     (m-intervals-brut :accessor m-intervals-brut :initarg :m-intervals-brut :
196         initform (list (gil::add-int-var-array *sp* *cf-last-index -12 12) nil nil
197             nil))
198     ;(m-intervals-brut :accessor m-intervals-brut :initarg :m-intervals-brut :
199         initform (list nil nil nil nil))
200     ;(m-intervals :accessor m-intervals :initarg :m-intervals :initform (list nil
201         nil nil nil))
202     (motions :accessor motions :initarg :motions :initform (list nil nil nil nil))
203     (motions-cost :accessor motions-cost :initarg :motions-cost :initform (list nil
204         nil nil nil))
205     (m2-intervals-brut :accessor m2-intervals-brut :initarg :m2-intervals-brut :
206         initform nil)
207     (m2-intervals :accessor m2-intervals :initarg :m2-intervals :initform nil)
208     (cf-brut-m-intervals :accessor cf-brut-m-intervals :initarg :cf-brut-m-intervals
209         :initform nil)
210     (is-cp-off-key-arr :accessor is-cp-off-key-arr :initarg :is-cp-off-key-arr :
211         initform nil)
212     (p-cons-cost :accessor p-cons-cost :initarg :p-cons-cost :initform nil)
213     (fifth-cost :accessor fifth-cost :initarg :fifth-cost :initform nil)
214     (octave-cost :accessor octave-cost :initarg :octave-cost :initform nil)
215     (m-degrees-cost :accessor m-degrees-cost :initarg :m-degrees-cost :initform nil)
216     (m-degrees-type :accessor m-degrees-type :initarg :m-degrees-type :initform nil)
217     (off-key-cost :accessor off-key-cost :initarg :off-key-cost :initform nil)
218     (m-all-intervals :accessor m-all-intervals :initarg :m-all-intervals :initform
219         nil)
220
221     (h-intervals-abs :accessor h-intervals-abs :initarg :h-intervals-abs :initform (
222         list nil nil nil nil))
223     (h-intervals-brut :accessor h-intervals-brut :initarg :h-intervals-brut :
224         initform (list nil nil nil nil))
225 ))
226
227 (defclass part-class () (
228     ; represents a part, i.e. a counterpoint or a cantus firmus
229     ; species
230     (species :accessor species :initarg :species :initform nil) ; 0 for cf, 1 for 1
231         st, 2 for 2nd, 3 for 3rd, 4 for 4th, 5 for 5th
232
233     ; solution-array
234     (solution-array :accessor solution-array :initarg :solution-array :initform nil)
235         ; contains the whole array of notes for this part, all merged together in
236         the good order
237     (solution-len :accessor solution-len :initarg :solution-len :initform nil) ;
238         number of note in this part
239
240     ; voice variables
241     (cp-range :accessor cp-range :initarg :cp-range :initform nil)
242     (cp-domain :accessor cp-domain :initarg :cp-domain :initform nil)

```

```

227 (chromatic-cp-domain :accessor chromatic-cp-domain :initarg :chromatic-cp-domain
    :initform nil)
228 (extended-cp-domain :accessor extended-cp-domain :initarg :extended-cp-domain :
    initform nil)
229 (off-domain :accessor off-domain :initarg :off-domain :initform nil)
230 (voice-type :accessor voice-type :initarg :voice-type :initform nil)
231
232 ; 1st species variables
233 (notes :accessor notes :initarg :notes :initform (list nil nil nil nil)) ;
    represents the notes of the counterpoint; (first notes) are all the notes of
    the first beat, (second notes) of the second etc
234 (h-intervals :accessor h-intervals :initarg :h-intervals :initform (list nil nil
    nil nil)) ; h-intervals to the lowest stratum
235 (m-intervals-brut :accessor m-intervals-brut :initarg :m-intervals-brut :
    initform (list
236 (gil::add-int-var-array *sp* *cf-last-index -12 12)
237 (gil::add-int-var-array *sp* *cf-last-index -12 12)
238 (gil::add-int-var-array *sp* *cf-last-index -12 12)
239 (gil::add-int-var-array *sp* *cf-last-index -12 12))) ; this variable is set
    before the others because we need it earlier
240 (m-intervals :accessor m-intervals :initarg :m-intervals :initform (list nil nil
    nil nil)) ; melodic intervals
241 (motions :accessor motions :initarg :motions :initform (list nil nil nil nil))
242 (motions-cost :accessor motions-cost :initarg :motions-cost :initform (list nil
    nil nil nil))
243 (is-cf-lower-arr :accessor is-cf-lower-arr :initarg :is-cf-lower-arr :initform (
    list nil nil nil nil)) ; true if the cf is lower
244 (m2-intervals-brut :accessor m2-intervals-brut :initarg :m2-intervals-brut :
    initform nil)
245 (m2-intervals :accessor m2-intervals :initarg :m2-intervals :initform nil)
246 (cf-brut-m-intervals :accessor cf-brut-m-intervals :initarg :cf-brut-m-intervals
    :initform nil)
247 (is-p-cons-arr :accessor is-p-cons-arr :initarg :is-p-cons-arr :initform nil)
248 (is-cp-off-key-arr :accessor is-cp-off-key-arr :initarg :is-cp-off-key-arr :
    initform nil)
249 (p-cons-cost :accessor p-cons-cost :initarg :p-cons-cost :initform nil)
250 (fifth-cost :accessor fifth-cost :initarg :fifth-cost :initform nil)
251 (octave-cost :accessor octave-cost :initarg :octave-cost :initform nil)
252 (m-degrees-cost :accessor m-degrees-cost :initarg :m-degrees-cost :initform nil)
253 (m-degrees-type :accessor m-degrees-type :initarg :m-degrees-type :initform nil)
254 (off-key-cost :accessor off-key-cost :initarg :off-key-cost :initform nil)
255 (m-all-intervals :accessor m-all-intervals :initarg :m-all-intervals :initform
    nil)
256
257 ; 2nd species variables
258 (h-intervals-abs :accessor h-intervals-abs :initarg :h-intervals-abs :initform (
    list nil nil nil nil))
259 (h-intervals-brut :accessor h-intervals-brut :initarg :h-intervals-brut :
    initform (list nil nil nil nil))
260 (m-succ-intervals :accessor m-succ-intervals :initarg :m-succ-intervals :
    initform (list nil nil nil nil))
261 (m-succ-intervals-brut :accessor m-succ-intervals-brut :initarg :
    m-succ-intervals-brut :initform (list nil nil nil nil))
262 (m2-len :accessor m2-len :initarg :m2-len :initform nil)
263 (total-m-len :accessor total-m-len :initarg :total-m-len :initform nil)
264 (m-all-intervals-brut :accessor m-all-intervals-brut :initarg :
    m-all-intervals-brut :initform nil)
265 (real-motions :accessor real-motions :initarg :real-motions :initform nil)
266 (real-motions-cost :accessor real-motions-cost :initarg :real-motions-cost :
    initform nil)
267 (is-ta-dim-arr :accessor is-ta-dim-arr :initarg :is-ta-dim-arr :initform nil)
268 (is-nbour-arr :accessor is-nbour-arr :initarg :is-nbour-arr :initform nil)

```

```

269 (penult-thesis-cost :accessor penult-thesis-cost :initarg :penult-thesis-cost :
    initform nil)
270
271 ; 3rd species variables
272 (is-5qn-linked-arr :accessor is-5qn-linked-arr :initarg :is-5qn-linked-arr :
    initform nil)
273 (is-not-cambiata-arr :accessor is-not-cambiata-arr :initarg :is-not-cambiata-arr
    :initform nil)
274 (not-cambiata-cost :accessor not-cambiata-cost :initarg :not-cambiata-cost :
    initform nil)
275 (m2-eq-zero-cost :accessor m2-eq-zero-cost :initarg :m2-eq-zero-cost :initform
    nil)
276 (is-cons-arr :accessor is-cons-arr :initarg :is-cons-arr :initform (list nil nil
    nil nil))
277 (cons-cost :accessor cons-cost :initarg :cons-cost :initform (list nil nil nil
    nil))
278
279 ; 4th species variables
280 (is-no-syncope-arr :accessor is-no-syncope-arr :initarg :is-no-syncope-arr :
    initform nil)
281 (no-syncope-cost :accessor no-syncope-cost :initarg :no-syncope-cost :initform
    nil)
282
283 ; 5th species variables
284 (species-arr :accessor species-arr :initarg :species-arr :initform nil) ; 0: no
    constraint, 1: first species, 2: second species, 3: third species, 4: fourth
    species
285 (sp-arr :accessor sp-arr :initarg :sp-arr :initform nil) ; represents *
    species-arr by position in the measure
286 (is-nth-species-arr :accessor is-nth-species-arr :initarg :is-nth-species-arr :
    initform (list nil nil nil nil nil)) ; if *species-arr is n, then *
    is-nth-species-arr is true
287 (is-3rd-species-arr :accessor is-3rd-species-arr :initarg :is-3rd-species-arr :
    initform (list nil nil nil nil)) ; if *species-arr is 3, then *
    is-3rd-species-arr is true
288 (is-4th-species-arr :accessor is-4th-species-arr :initarg :is-4th-species-arr :
    initform (list nil nil nil nil)) ; if *species-arr is 4, then *
    is-4th-species-arr is true
289 (is-2nd-or-3rd-species-arr :accessor is-2nd-or-3rd-species-arr :initarg :
    is-2nd-or-3rd-species-arr :initform nil) ; if *species-arr is 2 or 3, then *
    is-2nd-or-3rd-species-arr is true
290 (m-ta-intervals :accessor m-ta-intervals :initarg :m-ta-intervals :initform nil)
    ; represents the m-intervals between the thesis note and the arsis note of
    the same measure
291 (m-ta-intervals-brut :accessor m-ta-intervals-brut :initarg :m-ta-intervals-brut
    :initform nil) ; same but without the absolute reduction
292 (is-mostly-3rd-arr :accessor is-mostly-3rd-arr :initarg :is-mostly-3rd-arr :
    initform nil) ; true if second, third and fourth notes are from the 3rd
    species
293 (is-constrained-arr :accessor is-constrained-arr :initarg :is-constrained-arr :
    initform nil) ; represents !(*is-0th-species-arr) i.e. there are species
    constraints
294 (is-cst-arr :accessor is-cst-arr :initarg :is-cst-arr :initform (list nil nil
    nil nil)) ; represents *is-constrained-arr for all beats of the measure
295
296 ; 3v variables
297 (variety-cost :accessor variety-cost :initarg :variety-cost :initform nil)
298 (is-not-lowest :accessor is-not-lowest :initarg :is-not-lowest :initform nil) ;
    represents if the current part is not the lowest stratum
299 (h-intervals-to-cf :accessor h-intervals-to-cf :initarg :h-intervals-to-cf :
    initform (list nil nil nil nil))
300 ))
301

```

```

302 (defun init-cantus-firmus ()
303   ; init a cantus-firmus class "object"
304   (let (
305     (cantus-firmus-notes (gil::add-int-var-array *sp* *cf-len 0 120))
306     )
307     (dotimes (i *cf-len) (gil::g-rel *sp* (nth i cantus-firmus-notes) gil::
308       IRT_EQ (nth i *cf)))
309     (make-instance 'part-class
310       :species 0 ; value 0 = cantus firmus
311       :notes (list cantus-firmus-notes nil nil nil) ; no notes in the
312         second, third and fourth beat
313     )
314   )
315 (defun init-counterpoint (voice-type species)
316   ; initialise a counterpoint 'object'
317   (let (
318     ; Lower bound and upper bound related to the cantus firmus pitch
319     (range-upper-bound (+ 12 (* 6 voice-type)))
320     (range-lower-bound (+ -6 (* 6 voice-type)))
321     )
322     (let (
323       ; set the pitch range of the counterpoint
324       (cp-range (range (+ *tone-pitch-cf range-upper-bound) :min (+ *
325         tone-pitch-cf range-lower-bound))) ; arbitrary range
326       )
327       (let (
328         ; set counterpoint pitch domain
329         (cp-domain (intersection cp-range *scale))
330         ; penultimate (first *cp) note domain
331         (chromatic-cp-domain (intersection cp-range *chromatic-scale))
332         ; set counterpoint extended pitch domain
333         (extended-cp-domain (intersection cp-range (union *scale *
334           borrowed-scale)))
335         ; set the domain of the only borrowed notes
336         (off-domain (intersection cp-range *off-scale))
337         )
338         ; create the instance, by passing it the arguments
339         (setf counterpoint (make-instance 'part-class
340           :cp-range cp-range
341           :cp-domain cp-domain
342           :chromatic-cp-domain
343             chromatic-cp-domain
344           :extended-cp-domain
345             extended-cp-domain
346           :off-domain off-domain
347           :voice-type voice-type
348           :species species
349         ))
350         (case species
351           ((1 2 3) (progn
352             ; set the first beats of all measures of the counterpoints
353             ; to be in the extended-cp-domain
354             (setf (first (notes counterpoint)) (gil::
355               add-int-var-array-dom *sp* *cf-len (extended-cp-domain
356                 counterpoint)))
357             ; then treat the case where species = 1, 2, or 3
358             (case species
359               (1 (if (is-borrow-allowed)
360                 ; then add to the penultimate note more
361                 ; possibilities

```

```

354         (setf (nth *cf-penult-index (first (notes
           counterpoint))) (gil::add-int-var-dom *sp* (
           chromatic-cp-domain counterpoint)))
355     ))
356     (2 (progn
357         ; add the arsis counterpoint array (of [*cf-len - 1]
           length) to the space with the domain cp-domain
358         (setf (third (notes counterpoint)) (gil::
           add-int-var-array-dom *sp* *cf-last-index (
           extended-cp-domain counterpoint)))
359         ; add to the penultimate note more possibilities
360         (if (is-borrow-allowed)
361             (setf (nth *cf-penult-index (third (notes
           counterpoint))) (gil::add-int-var-dom *sp* (
           chromatic-cp-domain counterpoint)))
362             )
363         ))
364     (3 (progn
365         (loop for i from 1 to 3 do
366             ; add all quarter notes to the space with the
           domain (cp-domain counterpoint)
367             (setf (nth i (notes counterpoint)) (gil::
           add-int-var-array-dom *sp* *cf-last-index (
           extended-cp-domain counterpoint)))

368             (if (and (eq i 3) (is-borrow-allowed))
369                 ; then add to the penultimate note more
           possibilities
370                 (setf (nth *cf-penult-index (nth i (notes
           counterpoint))) (gil::add-int-var-dom *
           sp* (chromatic-cp-domain counterpoint)))
371             )
372             )
373         ))
374     ))
375 )
376 ))
377 (4 (progn
378     ; add the arsis counterpoint array (of [*cf-len - 1] length)
           to the space with the domain (cp-domain counterpoint)
379     (setf (third (notes counterpoint)) (gil::
           add-int-var-array-dom *sp* *cf-last-index (
           extended-cp-domain counterpoint)))
380     (setf (first (notes counterpoint)) (gil::
           add-int-var-array-dom *sp* *cf-last-index (
           extended-cp-domain counterpoint)))
381     ; add to the penultimate note more possibilities
382     (if (and (is-borrow-allowed) (/= *N-PARTS 1))
383         (progn
384             (setf (nth *cf-penult-index (third (notes counterpoint))
           ) (gil::add-int-var-dom *sp* (chromatic-cp-domain
           counterpoint)))
385             (setf (nth *cf-penult-index (first (notes counterpoint))
           ) (gil::add-int-var-dom *sp* (chromatic-cp-domain
           counterpoint)))
386             )
387         )
388     ))
389 (5 (progn
390     (loop for i from 0 to 3 do
391         (if (eq i 0)
392             (progn
393                 ; add all quarter notes to the space with the
           domain (notes counterpoint)-domain

```

```

394         (setf (nth i (notes counterpoint)) (gil::
          add-int-var-array-dom *sp* *cf-len (
            extended-cp-domain counterpoint)))
395       ; then add to the penultimate note more
          possibilities
396       (if (is-borrow-allowed)
397         (setf (nth *cf-penult-index (nth i (notes
          counterpoint))) (gil::add-int-var-dom *
            sp* (chromatic-cp-domain counterpoint)))
          )
398     )
399   )
400   (progn
401     ; same as above but 1 note shorter
402     (setf (nth i (notes counterpoint)) (gil::
          add-int-var-array-dom *sp* *cf-last-index (
            extended-cp-domain counterpoint)))
403     (if (is-borrow-allowed)
404       (setf (nth *cf-penult-index (nth i (notes
          counterpoint))) (gil::add-int-var-dom *
            sp* (chromatic-cp-domain counterpoint)))
          )
405     )
406   )
407 )
408 )
409   (setf (third (m-intervals-brut counterpoint)) (gil::
          add-int-var-array *sp* *cf-last-index -16 16))
410   ))
411 )
412 ; 3 voices specific
413 (if (eq *N-PARTS 3) (let ( ; if re-mi-la-si is the last cf note then
          you can use a major third even if it's not in the harmony
414   (tonal (mod (car (last *cf)) 12))
415   )
416   (case tonal ((2 4 9 10)
417     ; using the chromatic domain as it is going to be
          constrained to the harmonic triad by a later constraint
418     (setf (car (last (first (notes counterpoint)))) (gil::
          add-int-var-dom *sp* (chromatic-cp-domain counterpoint))
          )
419     )))
420 )
421 counterpoint
422 )
423 )
424 )
425 )
426
427 (defun fux-cp (species-list)
428   "Dispatches the counterpoint generation to the appropriate function according to
          the species."
429   (print (list "Chosen species for the counterpoints: " species-list))
430   ; THE CSP SPACE
431   (defparameter *sp* (gil::new-space))
432
433   ; re/set global variables
434   (define-global-constants)
435   (setq *species-list species-list) ; corresponds to the species of the
          counterpoints (1 5) means one ctp of 1st sp. and one ctp of 5th sp.
436   (setq *cost-indexes (make-hash-table)) ; a hashmap recording which cost is at
          which position in the cost-factors list
437   (setq *cost-factors (set-cost-factors)) ; the cost-factors list, contains all
          the individual costs
438

```

```

439 ;; CREATE THE PARTS
440 (setq counterpoints (make-list *N-COUNTERPOINTS :initial-element nil)) ; list
    containing the counterpoints
441 (dotimes (i *N-COUNTERPOINTS) (setf (nth i counterpoints) (init-counterpoint (
    nth i *voices-types) (nth i species-list)))) ; init the counterpoints-
442 (setq *cantus-firmus (init-cantus-firmus)) ; init the part 'object' for the
    cantus-firmus
443 (setq *parts (cons *cantus-firmus counterpoints)) ; list containing all the
    parts in this order: (cf, cp1, cp2)
444
445
446 ;; CREATE THE STRATA
447 (setq *upper (make-list *N-COUNTERPOINTS :initial-element nil)) ; list
    containing the upper strata (the middle and the uppermost strata)
448 (dotimes (i *N-COUNTERPOINTS) (setf (nth i *upper) (make-instance 'stratum-class
    ))) ; declare the upper strata
449 (setq *lowest (make-instance 'stratum-class)) ; declare the lowest stratum
450 (setf (first (notes *lowest)) (gil::add-int-var-array *sp* *cf-len 0 120))
451 (create-strata-arrays *parts) ; create the strata arrays
452
453 (case *N-COUNTERPOINTS
454   (1 (progn
455       (fux-cp-cf (first *parts)) ; apply the constraints wrt. the cantus
        firmus
456       (case (first species-list) ; in this case len(species-list) = 1
457         (1 (fux-cp-1st (second *parts)))
458         (2 (fux-cp-2nd (second *parts)))
459         (3 (fux-cp-3rd (second *parts)))
460         (4 (fux-cp-4th (second *parts)))
461         (5 (fux-cp-5th (second *parts)))
462         (otherwise (error "Species ~A not implemented" species)))
463       )
464     )
465     (2 (fux-cp-3v species-list *parts)) ; 3v dispatcher
466     (otherwise (error "Only two additional voices are implemented up to now. You
        asked for ~A." (length species))))
467 )
468 )
469
470 (defun fux-search-engine (the-cp &optional (species '(1)) (voice-type 0))
471   (let (se tstop sopts)
472     (print (list "Starting fux-search-engine with species = " species))
473     ;; COST
474     (reorder-costs)
475     (gil::g-cost *sp* *cost-factors) ; set the cost function
476
477     ;; SPECIFY SOLUTION VARIABLES
478     ; (print "Specifying solution variables...")
479     (gil::g-specify-sol-variables *sp* the-cp)
480     (gil::g-specify-percent-diff *sp* 0)
481
482     ;; BRANCHING
483     ; (print "Branching...")
484     (setq var-branch-type gil::INT_VAR_DEGREE_MAX)
485     (setq val-branch-type gil::INT_VAL_SPLIT_MIN)
486
487     (gil::g-branch *sp* (first (notes *lowest)) gil::INT_VAR_DEGREE_MAX gil::
        INT_VAL_SPLIT_MIN)
488     (dotimes (i *N-COUNTERPOINTS) (progn
489       ; 5th species specific
490       (if (eq (nth i species) 5) ; otherwise there is no species array
491         (gil::g-branch *sp* (species-arr (nth i counterpoints))
            var-branch-type gil::INT_VAL_RND)

```

```

492         )
493
494         ; 5th species specific
495         (if (eq (nth i species) 5) (progn ; otherwise there is no species array
496             (gil::g-branch *sp* (no-syncope-cost (nth i counterpoints))
497                 var-branch-type val-branch-type)
498             (gil::g-branch *sp* (not-cambiata-cost (nth i counterpoints))
499                 var-branch-type val-branch-type)
500         ))
501
502         (if (eq (nth i species) 4)
503             (gil::g-branch *sp* (no-syncope-cost (nth i counterpoints))
504                 var-branch-type gil::INT_VAL_MIN)
505         )
506     ))
507
508     ;; Solution variables branching
509     (gil::g-branch *sp* the-cp gil::INT_VAR_DEGREE_MAX gil::INT_VAL_RND) ;
510     the-cp is all the solution arrays merged together
511
512     ; time stop
513     (setq tstop (gil::t-stop)); create the time stop object
514     (setq timeout 5)
515     (gil::time-stop-init tstop (* timeout 1000)); initialize it (time is
516         expressed in ms)
517
518     ; search options
519     (setq sopts (gil::search-opts)); create the search options object
520     (gil::init-search-opts sopts); initialize it
521     ; (gil::set-n-threads sopts 1)
522     (gil::set-time-stop sopts tstop); set the timestop object to stop the search
523     if it takes too long
524
525     ;; SEARCH ENGINE
526     (print "Search engine...")
527     (setq se (gil::search-engine *sp* (gil::opts sopts) gil::BAB));
528     (print se)
529
530     (print "CSP constructed")
531     (list se the-cp tstop sopts)
532 )
533 )
534
535 ; SEARCH-NEXT-SOLUTION
536 ; <l> is a list containing in that order the search engine for the problem, the
537 ; variables
538 ; this function finds the next solution of the CSP using the search engine given as
539 ; an argument
540 (defun search-next-fux-cp (l)
541     (print "Searching next solution...")
542     (let (
543         (se (first l))
544         (the-cp (second l))
545         (tstop (third l))
546         (sopts (fourth l))
547         (species-list (fifth l))
548         (check t)
549         sol sol-pitches sol-species
550     )
551         (time (om::while check :do

```

```

547         ; reset the tstop timer before launching the search
548         (gil::time-stop-reset tstop)
549         ; try to find a solution
550         (time (setq sol (try-find-solution se)))
551         (if (null sol)
552             ; then check if there are solutions left and if the user wishes to
              continue searching
553             (stopped-or-ended (gil::stopped se) (getparam 'is-stopped))
554             ; else we have found a solution so break fthe loop
555             (setf check nil)
556         )
557     ))
558
559     ; print the solution from GiL
560     (print "Solution: ")
561     (handler-case
562         (progn
563             (print (list "*cost-factors" (gil::g-values sol *cost-factors)))
564             (print (list "sum of all costs = " (reduce #' + (gil::g-values sol *
              cost-factors) :initial-value 0)))
565         )
566         (error (c)
567             (dotimes (i *N-COST-FACTORS)
568                 (handler-case (gil::g-values sol (nth i *cost-factors)) (error (
              c) (print (list "Cost" i "had a problem."))))
569             )
570             (error "All costs are not set correctly. Correct this problem before
              trying to find a solution.")
571         )
572     )
573
574     (print (list "species = " species-list))
575
576     (print "The solution can now be retrieved by evaluating the third output of
        cp-params.")
577     (setq sol-pitches (gil::g-values sol the-cp)) ; store the values of the
        solution
578     (let (
579         (basic-rythmics (get-basic-rythmics species-list *cf-len sol-pitches
            counterpoints sol))
580         (sol-voices (make-list *N-COUNTERPOINTS :initial-element nil))
581     )
582
583         (loop for i from 0 below *N-COUNTERPOINTS do (progn
584             (setq rhythmic+pitches (nth i basic-rythmics)) ; get the rhythmic
              correpsonding to the species
585             (setq rhythmic-om (first rhythmic+pitches))
586             (setq pitches-om (second rhythmic+pitches))
587         )
588
589             (setf (nth i sol-voices) (make-instance 'voice :chords (to-midicent
              pitches-om) :tree (om::mktree rhythmic-om '(4 4)) :tempo *
              cf-tempo))
590         )
591         (make-instance 'poly :voices sol-voices)
592     )
593 )
594 )
595
596 ; try to find a solution, catch errors from GiL and Gecode and restart the search
597 (defun try-find-solution (se)
598     (handler-case

```

```

599     (gil::search-next se) ; search the next solution, sol is the space of the
        solution
600     (error (c)
601       (print "A search was already running. Please start over by saving the
        configuration and starting the search.")
602       ;(try-find-solution se)
603     )
604   )
605 )
606
607 ; determines if the search has been stopped by the solver because there are no more
        solutions or if the user has stopped the search
608 (defun stopped-or-ended (stopped-se stop-user)
609   (print (list "stopped-se" stopped-se "stop-user" stop-user))
610   (if (= stopped-se 0); if the search has not been stopped by the TimeStop object,
        there is no more solutions
611       (error "The search was stopped because no more solution was found. Either
        the best solution was found or none exist.")
612   )
613   ;otherwise, check if the user wants to keep searching or not
614   (if stop-user
615       (error "The search was stopped. Press next to continue the search.")
616   )
617 )
618
619 (defun reorder-costs ()
620   ; this function serves to put the costs in the order asked by the user and
        combines them using either a linear combination or a maximum minimisation
621   ; in the interface the user selects an "importance" for each cost: it is the
        order in which the costs are being sorted.
622   ; advice if someone wants to continue with the implementation: in a OOP language
        just define the order directly in a dictionary or somewhat so there is no
        need to reorder at some point
623
624   ; at the end of this function, 'cost-names-by-order'
625   (setf costs-names-by-order (make-list 14 :initial-element nil)) ; there are
        fourteen types of cost
626   ; take the costs in *cost-preferences* and sort them according to the user
        preferences in list 'costs-names-by-order'
627   (maphash #'(lambda (key value)
628     (setf value (- (parse-integer value) 1))
629     (setf (nth value costs-names-by-order) (append (nth value
        costs-names-by-order) (list key)))
630   )
        *cost-preferences*)
631   ; now 'costs-names-by-order' looks like this (python notation) [[cost1, cost2], [
        cost3], [cost4], [cost5, cost6]]
632   ; which means first level for the lexicographic order are cost1 and cost2
633   ; cost3 comes on the second level, cost4 on the third, and cost5 and cost6 are
        together on the fourth level
634
635   ; reverse the cost order because when passing them between GiL and C++ they are
        reversed again
637   (setf costs-names-by-order (reverse costs-names-by-order))
638   (print "Order of the costs, in reversed order:")
639   (let (
640     (i 0)
641     (n-different-costs 0) ; the amount of cost types that have been encountered;
        not all the costs will be encountered, as some are species-specific (
        like the cost for no syncopation, that occurs only with a 4th species
        ctp)
642     (reordered-costs (make-list *N-COST-FACTORS :initial-element nil))
643   )

```

```

644 (assert costs-names-by-order () "costs-names-by-order is nil, which means no
        preferences were passed. This is an implementation bug.")
645 ; for each level of the ordered cost names (i.e. for each sublist in [[cost1
        , cost2], [cost3], [cost4], [cost5, cost6]])
646 (dolist (preference-level costs-names-by-order)
647   (let
648     (
649       (current-cost-array '()) ; the array of the actual values of the
        costs in the preference-level
650       (current-cost-sum (gil::add-int-var *sp* 0 1000)) ; a variable
        representing the combination of the costs of this level
651     )
652     ; for each cost name in the level
653     (dolist (cost preference-level)
654       (let ((index (gethash cost *cost-indexes)))
655         (if index (progn
656           (loop for index in (gethash cost *cost-indexes) do (
657             ; get the value of the cost and put in into the
        current array of cost values
658             (push (nth index *cost-factors) current-cost-array)
659           ))
660           )
661           ; if index is nil (i.e. if this cost doesn't exist in
        this species)
662           ; it is not a problem, it is the normal way of working
        since some costs don't exist in some species
663           #| debug |# ; (print (list "Cost " cost " was not found
        in this configuration."))
664         )
665       )
666     )
667     (if current-cost-array (progn ; if there was at least one cost on
        this level
668       (if *linear-combination
669         ; if the user asked for linear combination then perform a
        linear combination
670         (gil::g-sum *sp* current-cost-sum current-cost-array)
671         ; else perform a maximum minimisation (take the maximum and
        the solver will minimise it)
672         (gil::g-lmax *sp* current-cost-sum current-cost-array)
673       )
674       ; put our linear combination or maximum minimisation into our
        global cost array
675       (setf (nth n-different-costs reordered-costs) current-cost-sum)
676       (print (list n-different-costs "th cost = " preference-level)) ;
        print the index at which the cost was added, please
        remember that it is reverted wrt. user preferences, as it
        will be reverted again when passing to Gil->C++
677       (incf n-different-costs)
678     ))
679   )
680 )
681 ; if some costs are on the same level, then not all slots were used, get rid
of them
682 (setf reordered-costs (subseq reordered-costs 0 n-different-costs))
683 ; verify that no index of the newly created cost array is nil
684 (dolist (cost reordered-costs) (assert cost () "A cost is nil. Ordered costs
= ~A. This is an implementation bug." reordered-costs))
685
686 ; set the global variable to now be the reordered-costs
687 (setf *cost-factors reordered-costs)
688 )

```

D.5 3v-ctp.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Anton Lamotte
4 ; Date: January 2024
5 ; This file contains the function that dispatches the counterpoints to their
   respective functions and adds all the necessary constraints for having 3 voices
   species.
6
7 ;;=====#
8 ;; Three voices counterpoint handler #
9 ;;=====#
10 (defun fux-cp-3v (species-list parts)
11   (print "##### 3 VOICES #####")
12
13   (setf cantus-firmus (first parts))
14   (setf counterpoint-1 (second parts))
15   (setf counterpoint-2 (third parts))
16
17   ;
18   ;
19   ;
20   ; for each part
21   (dotimes (i *N-PARTS)
22     (case (species (nth i parts))
23       (0 (fux-cp-cf (nth i parts))) ; dispatch to the cantus firmus function
24       (1 (fux-cp-1st (nth i parts) 3v-1sp)) ; dispatch to the first species
25         function
26       (2 (fux-cp-2nd (nth i parts) 3v-2sp)) ; dispatch to the second species
27         function
28       (3 (fux-cp-3rd (nth i parts) 3v-3sp)) ; dispatch to the third species
29         function
30       (4 (fux-cp-4th (nth i parts) 3v-4sp)) ; dispatch to the fourth species
31         function
32       (5 (fux-cp-5th (nth i parts) 3v-5sp)) ; dispatch to the fifth species
33         function
34       (otherwise (error "Unexpected value in the species list, when calling
35         fux-cp-3v."))
36     )
37   )
38
39   ;
40   ;
41   ;
42   ;
43   ;
44   ;
45   ;
46   ;
47   ;
48   ;
49   ;
50   ;
51   ;
52   ;
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   ;
78   ;
79   ;
80   ;
81   ;
82   ;
83   ;
84   ;
85   ;
86   ;
87   ;
88   ;
89   ;
90   ;
91   ;
92   ;
93   ;
94   ;
95   ;
96   ;
97   ;
98   ;
99   ;
100  ;
101  ;
102  ;
103  ;
104  ;
105  ;
106  ;
107  ;
108  ;
109  ;
110  ;
111  ;
112  ;
113  ;
114  ;
115  ;
116  ;
117  ;
118  ;
119  ;
120  ;
121  ;
122  ;
123  ;
124  ;
125  ;
126  ;
127  ;
128  ;
129  ;
130  ;
131  ;
132  ;
133  ;
134  ;
135  ;
136  ;
137  ;
138  ;
139  ;
140  ;
141  ;
142  ;
143  ;
144  ;
145  ;
146  ;
147  ;
148  ;
149  ;
150  ;
151  ;
152  ;
153  ;
154  ;
155  ;
156  ;
157  ;
158  ;
159  ;
160  ;
161  ;
162  ;
163  ;
164  ;
165  ;
166  ;
167  ;
168  ;
169  ;
170  ;
171  ;
172  ;
173  ;
174  ;
175  ;
176  ;
177  ;
178  ;
179  ;
180  ;
181  ;
182  ;
183  ;
184  ;
185  ;
186  ;
187  ;
188  ;
189  ;
190  ;
191  ;
192  ;
193  ;
194  ;
195  ;
196  ;
197  ;
198  ;
199  ;
200  ;
201  ;
202  ;
203  ;
204  ;
205  ;
206  ;
207  ;
208  ;
209  ;
210  ;
211  ;
212  ;
213  ;
214  ;
215  ;
216  ;
217  ;
218  ;
219  ;
220  ;
221  ;
222  ;
223  ;
224  ;
225  ;
226  ;
227  ;
228  ;
229  ;
230  ;
231  ;
232  ;
233  ;
234  ;
235  ;
236  ;
237  ;
238  ;
239  ;
240  ;
241  ;
242  ;
243  ;
244  ;
245  ;
246  ;
247  ;
248  ;
249  ;
250  ;
251  ;
252  ;
253  ;
254  ;
255  ;
256  ;
257  ;
258  ;
259  ;
260  ;
261  ;
262  ;
263  ;
264  ;
265  ;
266  ;
267  ;
268  ;
269  ;
270  ;
271  ;
272  ;
273  ;
274  ;
275  ;
276  ;
277  ;
278  ;
279  ;
280  ;
281  ;
282  ;
283  ;
284  ;
285  ;
286  ;
287  ;
288  ;
289  ;
290  ;
291  ;
292  ;
293  ;
294  ;
295  ;
296  ;
297  ;
298  ;
299  ;
300  ;
301  ;
302  ;
303  ;
304  ;
305  ;
306  ;
307  ;
308  ;
309  ;
310  ;
311  ;
312  ;
313  ;
314  ;
315  ;
316  ;
317  ;
318  ;
319  ;
320  ;
321  ;
322  ;
323  ;
324  ;
325  ;
326  ;
327  ;
328  ;
329  ;
330  ;
331  ;
332  ;
333  ;
334  ;
335  ;
336  ;
337  ;
338  ;
339  ;
340  ;
341  ;
342  ;
343  ;
344  ;
345  ;
346  ;
347  ;
348  ;
349  ;
350  ;
351  ;
352  ;
353  ;
354  ;
355  ;
356  ;
357  ;
358  ;
359  ;
360  ;
361  ;
362  ;
363  ;
364  ;
365  ;
366  ;
367  ;
368  ;
369  ;
370  ;
371  ;
372  ;
373  ;
374  ;
375  ;
376  ;
377  ;
378  ;
379  ;
380  ;
381  ;
382  ;
383  ;
384  ;
385  ;
386  ;
387  ;
388  ;
389  ;
390  ;
391  ;
392  ;
393  ;
394  ;
395  ;
396  ;
397  ;
398  ;
399  ;
400  ;
401  ;
402  ;
403  ;
404  ;
405  ;
406  ;
407  ;
408  ;
409  ;
410  ;
411  ;
412  ;
413  ;
414  ;
415  ;
416  ;
417  ;
418  ;
419  ;
420  ;
421  ;
422  ;
423  ;
424  ;
425  ;
426  ;
427  ;
428  ;
429  ;
430  ;
431  ;
432  ;
433  ;
434  ;
435  ;
436  ;
437  ;
438  ;
439  ;
440  ;
441  ;
442  ;
443  ;
444  ;
445  ;
446  ;
447  ;
448  ;
449  ;
450  ;
451  ;
452  ;
453  ;
454  ;
455  ;
456  ;
457  ;
458  ;
459  ;
460  ;
461  ;
462  ;
463  ;
464  ;
465  ;
466  ;
467  ;
468  ;
469  ;
470  ;
471  ;
472  ;
473  ;
474  ;
475  ;
476  ;
477  ;
478  ;
479  ;
480  ;
481  ;
482  ;
483  ;
484  ;
485  ;
486  ;
487  ;
488  ;
489  ;
490  ;
491  ;
492  ;
493  ;
494  ;
495  ;
496  ;
497  ;
498  ;
499  ;
500  ;
501  ;
502  ;
503  ;
504  ;
505  ;
506  ;
507  ;
508  ;
509  ;
510  ;
511  ;
512  ;
513  ;
514  ;
515  ;
516  ;
517  ;
518  ;
519  ;
520  ;
521  ;
522  ;
523  ;
524  ;
525  ;
526  ;
527  ;
528  ;
529  ;
530  ;
531  ;
532  ;
533  ;
534  ;
535  ;
536  ;
537  ;
538  ;
539  ;
540  ;
541  ;
542  ;
543  ;
544  ;
545  ;
546  ;
547  ;
548  ;
549  ;
550  ;
551  ;
552  ;
553  ;
554  ;
555  ;
556  ;
557  ;
558  ;
559  ;
560  ;
561  ;
562  ;
563  ;
564  ;
565  ;
566  ;
567  ;
568  ;
569  ;
570  ;
571  ;
572  ;
573  ;
574  ;
575  ;
576  ;
577  ;
578  ;
579  ;
580  ;
581  ;
582  ;
583  ;
584  ;
585  ;
586  ;
587  ;
588  ;
589  ;
590  ;
591  ;
592  ;
593  ;
594  ;
595  ;
596  ;
597  ;
598  ;
599  ;
600  ;
601  ;
602  ;
603  ;
604  ;
605  ;
606  ;
607  ;
608  ;
609  ;
610  ;
611  ;
612  ;
613  ;
614  ;
615  ;
616  ;
617  ;
618  ;
619  ;
620  ;
621  ;
622  ;
623  ;
624  ;
625  ;
626  ;
627  ;
628  ;
629  ;
630  ;
631  ;
632  ;
633  ;
634  ;
635  ;
636  ;
637  ;
638  ;
639  ;
640  ;
641  ;
642  ;
643  ;
644  ;
645  ;
646  ;
647  ;
648  ;
649  ;
650  ;
651  ;
652  ;
653  ;
654  ;
655  ;
656  ;
657  ;
658  ;
659  ;
660  ;
661  ;
662  ;
663  ;
664  ;
665  ;
666  ;
667  ;
668  ;
669  ;
670  ;
671  ;
672  ;
673  ;
674  ;
675  ;
676  ;
677  ;
678  ;
679  ;
680  ;
681  ;
682  ;
683  ;
684  ;
685  ;
686  ;
687  ;
688  ;
689  ;
690  ;
691  ;
692  ;
693  ;
694  ;
695  ;
696  ;
697  ;
698  ;
699  ;
700  ;
701  ;
702  ;
703  ;
704  ;
705  ;
706  ;
707  ;
708  ;
709  ;
710  ;
711  ;
712  ;
713  ;
714  ;
715  ;
716  ;
717  ;
718  ;
719  ;
720  ;
721  ;
722  ;
723  ;
724  ;
725  ;
726  ;
727  ;
728  ;
729  ;
730  ;
731  ;
732  ;
733  ;
734  ;
735  ;
736  ;
737  ;
738  ;
739  ;
740  ;
741  ;
742  ;
743  ;
744  ;
745  ;
746  ;
747  ;
748  ;
749  ;
750  ;
751  ;
752  ;
753  ;
754  ;
755  ;
756  ;
757  ;
758  ;
759  ;
760  ;
761  ;
762  ;
763  ;
764  ;
765  ;
766  ;
767  ;
768  ;
769  ;
770  ;
771  ;
772  ;
773  ;
774  ;
775  ;
776  ;
777  ;
778  ;
779  ;
780  ;
781  ;
782  ;
783  ;
784  ;
785  ;
786  ;
787  ;
788  ;
789  ;
790  ;
791  ;
792  ;
793  ;
794  ;
795  ;
796  ;
797  ;
798  ;
799  ;
800  ;
801  ;
802  ;
803  ;
804  ;
805  ;
806  ;
807  ;
808  ;
809  ;
810  ;
811  ;
812  ;
813  ;
814  ;
815  ;
816  ;
817  ;
818  ;
819  ;
820  ;
821  ;
822  ;
823  ;
824  ;
825  ;
826  ;
827  ;
828  ;
829  ;
830  ;
831  ;
832  ;
833  ;
834  ;
835  ;
836  ;
837  ;
838  ;
839  ;
840  ;
841  ;
842  ;
843  ;
844  ;
845  ;
846  ;
847  ;
848  ;
849  ;
850  ;
851  ;
852  ;
853  ;
854  ;
855  ;
856  ;
857  ;
858  ;
859  ;
860  ;
861  ;
862  ;
863  ;
864  ;
865  ;
866  ;
867  ;
868  ;
869  ;
870  ;
871  ;
872  ;
873  ;
874  ;
875  ;
876  ;
877  ;
878  ;
879  ;
880  ;
881  ;
882  ;
883  ;
884  ;
885  ;
886  ;
887  ;
888  ;
889  ;
890  ;
891  ;
892  ;
893  ;
894  ;
895  ;
896  ;
897  ;
898  ;
899  ;
900  ;
901  ;
902  ;
903  ;
904  ;
905  ;
906  ;
907  ;
908  ;
909  ;
910  ;
911  ;
912  ;
913  ;
914  ;
915  ;
916  ;
917  ;
918  ;
919  ;
920  ;
921  ;
922  ;
923  ;
924  ;
925  ;
926  ;
927  ;
928  ;
929  ;
930  ;
931  ;
932  ;
933  ;
934  ;
935  ;
936  ;
937  ;
938  ;
939  ;
940  ;
941  ;
942  ;
943  ;
944  ;
945  ;
946  ;
947  ;
948  ;
949  ;
950  ;
951  ;
952  ;
953  ;
954  ;
955  ;
956  ;
957  ;
958  ;
959  ;
960  ;
961  ;
962  ;
963  ;
964  ;
965  ;
966  ;
967  ;
968  ;
969  ;
970  ;
971  ;
972  ;
973  ;
974  ;
975  ;
976  ;
977  ;
978  ;
979  ;
980  ;
981  ;
982  ;
983  ;
984  ;
985  ;
986  ;
987  ;
988  ;
989  ;
990  ;
991  ;
992  ;
993  ;
994  ;
995  ;
996  ;
997  ;
998  ;
999  ;
1000 ;

```

```

39 (dotimes (i *N-COUNTERPOINTS)
40   (create-h-intervals (first (notes (nth i *upper))) (first (notes *lowest)) (
41     first (h-intervals (nth i *upper))))
42   (setf (h-intervals-abs (nth i *upper)) (gil::add-int-var-array *sp* *cf-len
43     -127 127))
44   (setf (h-intervals-brut (nth i *upper)) (gil::add-int-var-array *sp* *cf-len
45     -127 127))
46   (create-intervals (first (notes *lowest)) (first (notes (nth i *upper))) (
47     h-intervals-abs (nth i *upper)) (h-intervals-brut (nth i *upper)))
48 )
49 ;
50 ; =====;
51 ;
52 ; CONSTRAINTS
53 ;
54 ; =====;
55
56 (loop
57   ; for each possible pair or parts
58   ; for example if we have (cf, cp1 and c2), take (cf and cp1), (cf and cp2)
59   ; and (cp1 and cp2)
60   for v1 in parts
61   for i from 0
62   do (loop for v2 in (nthcdr (1+ i) parts)
63     do (progn
64       ; no unison between the voices
65       (print "No unison between the voices")
66       (dotimes (i 4) (if (eq i 0)
67         ; first beat can be the same on first and last measure
68         (add-no-unison-cst (nth i (notes v1)) (nth i (notes v2)))
69         ; other beats must always be different
70         (add-no-unison-at-all-cst (nth i (notes v1)) (nth i (notes v2))))
71       ))
72   ))
73 )
74
75 ; it is not allowed to have two direct motions
76 (print "No together move")
77 (add-no-together-move-cst (list (first (motions counterpoint-1)) (first (motions
78   counterpoint-2)) (first (motions cantus-firmus))))
79
80 (print "Last chord cannot be minor")
81 (dotimes (i *N-COUNTERPOINTS)
82   (add-no-minor-third-cst (lastone (first (h-intervals (nth i *upper)))))
83 )
84
85 (print "Last chord cannot include a tenth")
86 (dotimes (i *N-COUNTERPOINTS)
87   (add-no-tenth-in-last-chord-cst (first (h-intervals (nth i *upper))) (
88     h-intervals-brut (nth i *upper)))
89 )
90
91 (print "Last chord must be a harmonic triad")
92 (add-last-chord-h-triad-cst (first (h-intervals (first *upper))) (first (
93   h-intervals (second *upper))))
94
95 (print "The last lowest note must be the same as the root note of the key")
96 (last-lowest-note-same-as-root-note-cst)
97
98 ; two fifth species counterpoints only
99 (if (equal species-list '(5 5)) (progn

```

```

89     (print "The rhythms of the two fifth-species counterpoints must be as
        different as possible")
90     (add-make-fifth-species-different-cst parts)
91   ))
92
93   ;
94   =====;
95
96   ; COSTS
97   ;
98   ; =====;
99
100  ; Cost #1 : no successive perfect consonances
101  (setf succ-p-cons-cost (gil::add-int-var-array-dom *sp* (* 3 *cf-last-index) (
    append '(0) (getparam-val 'succ-p-cons-cost))))
102  (setf succ-p-cons-cost-index 0)
103  (loop
104    ; for each possible pair or parts
105    ; for example if we have (cf, cp1 and c2), take (cf and cp1), (cf and cp2)
106    ; and (cp1 and cp2)
107    for v1 in parts
108    for i from 0
109    do (loop for v2 in (nthcdr (1+ i) parts)
110      do (progn
111        (print "As few successive perfect consonances as possible")
112        (let (
113          (h-intervals-1-2 (gil::add-int-var-array *sp* *cf-len 0 11)) ; the
114            h-intervals between p1 and p2
115          (is-p-cons-arr-1-2 (gil::add-bool-var-array *sp* *cf-len 0 1)) ; the
116            is h-intervals-1-2 a perfect consonance
117          (current-cost (subseq succ-p-cons-cost succ-p-cons-cost-index)) ;
118            succ-p-cons-cost is a long array of size 3m, each slice of m
119            being dedicated for the costs between a pair of parts
120          )
121          (incf succ-p-cons-cost-index *cf-last-index) ; set the index to the
122            next slice
123
124          (if (member 4 (list (species v1) (species v2)))
125            (progn ; first case, we have a fourth species counterpoint in
126              the composition
127              (if (eq (species v1) 4)
128                (if (eq (species v2) 4)
129                  ; both are of fourth species, compute using the
130                  third beat for each
131                  (create-h-intervals (third (notes v1)) (third (notes
132                    v2)) h-intervals-1-2)
133                  ; only the first is of fourth species, compute using
134                  the third beat for it
135                  (create-h-intervals (third (notes v1)) (first (notes
136                    v2)) h-intervals-1-2)
137                )
138              ; only the second is of fourth species, compute using
139              the third beat for it
140              (create-h-intervals (first (notes v1)) (third (notes v2)
141                ) h-intervals-1-2)
142              )
143          ; if one voice is of the fourth species the last chord was
144          not created yet, due to the delaying of the fourth
145          species
146          (create-h-intervals (last (first (notes v1))) (last (first (
147            notes v2))) (last h-intervals-1-2))
148          )
149        )
150      )
151    )

```

```

129         (progn ; "normal" case: compute using the first beat for all the
130             parts
131             (create-h-intervals (first (notes v1)) (first (notes v2))
132                 h-intervals-1-2)
133         )
134     )
135     (create-is-p-cons-arr h-intervals-1-2 is-p-cons-arr-1-2)
136     (cond
137         ((and (/= 2 (species v1)) (/= 2 (species v2)) (/= 4 (species v1))
138             (/= 4 (species v2))) ; if both voices are not from the 2nd
139             nor from the 4th species
140             (add-no-successive-p-cons-cst is-p-cons-arr-1-2 current-cost
141                 ) ; for all species except the fourth and the second,
142                 successive perfect consonances are avoided
143         )
144         ((= 2 (species v1))
145             (add-no-successive-p-cons-2nd-species-cst is-p-cons-arr-1-2
146                 h-intervals-1-2 (first (m-succ-intervals v1))
147                 current-cost) ; for the second species, successive
148                 fifths are allowed if there is a third in between
149         )
150         ((= 2 (species v2))
151             (add-no-successive-p-cons-2nd-species-cst is-p-cons-arr-1-2
152                 h-intervals-1-2 (first (m-succ-intervals v2))
153                 current-cost) ; for the second species, successive
154                 fifths are allowed if there is a third in between
155         )
156         ((or (eq 4 (species v1)) (eq 4 (species v2)))
157             (add-no-successive-p-cons-4th-species-cst is-p-cons-arr-1-2
158                 h-intervals-1-2 current-cost) ; for the fourth species,
159                 successive fifths are allowed, but no other successive
160                 perfect consonances
161         )
162     )
163 )
164 )
165 )
166 )
167 )
168 )
169 )
170 )
171 )
172 )
173 )
174 )
175 )
176 )
177 )
178 )
179 )
180 )
181 )
182 )
183 )
184 )
185 )
186 )
187 )
188 )
189 )
190 )
191 )
192 )
193 )
194 )
195 )
196 )
197 )
198 )
199 )
200 )
201 )
202 )
203 )
204 )
205 )
206 )
207 )
208 )
209 )
210 )
211 )
212 )
213 )
214 )
215 )
216 )
217 )
218 )
219 )
220 )
221 )
222 )
223 )
224 )
225 )
226 )
227 )
228 )
229 )
230 )
231 )
232 )
233 )
234 )
235 )
236 )
237 )
238 )
239 )
240 )
241 )
242 )
243 )
244 )
245 )
246 )
247 )
248 )
249 )
250 )
251 )
252 )
253 )
254 )
255 )
256 )
257 )
258 )
259 )
260 )
261 )
262 )
263 )
264 )
265 )
266 )
267 )
268 )
269 )
270 )
271 )
272 )
273 )
274 )
275 )
276 )
277 )
278 )
279 )
280 )
281 )
282 )
283 )
284 )
285 )
286 )
287 )
288 )
289 )
290 )
291 )
292 )
293 )
294 )
295 )
296 )
297 )
298 )
299 )
300 )
301 )
302 )
303 )
304 )
305 )
306 )
307 )
308 )
309 )
310 )
311 )
312 )
313 )
314 )
315 )
316 )
317 )
318 )
319 )
320 )
321 )
322 )
323 )
324 )
325 )
326 )
327 )
328 )
329 )
330 )
331 )
332 )
333 )
334 )
335 )
336 )
337 )
338 )
339 )
340 )
341 )
342 )
343 )
344 )
345 )
346 )
347 )
348 )
349 )
350 )
351 )
352 )
353 )
354 )
355 )
356 )
357 )
358 )
359 )
360 )
361 )
362 )
363 )
364 )
365 )
366 )
367 )
368 )
369 )
370 )
371 )
372 )
373 )
374 )
375 )
376 )
377 )
378 )
379 )
380 )
381 )
382 )
383 )
384 )
385 )
386 )
387 )
388 )
389 )
390 )
391 )
392 )
393 )
394 )
395 )
396 )
397 )
398 )
399 )
400 )
401 )
402 )
403 )
404 )
405 )
406 )
407 )
408 )
409 )
410 )
411 )
412 )
413 )
414 )
415 )
416 )
417 )
418 )
419 )
420 )
421 )
422 )
423 )
424 )
425 )
426 )
427 )
428 )
429 )
430 )
431 )
432 )
433 )
434 )
435 )
436 )
437 )
438 )
439 )
440 )
441 )
442 )
443 )
444 )
445 )
446 )
447 )
448 )
449 )
450 )
451 )
452 )
453 )
454 )
455 )
456 )
457 )
458 )
459 )
460 )
461 )
462 )
463 )
464 )
465 )
466 )
467 )
468 )
469 )
470 )
471 )
472 )
473 )
474 )
475 )
476 )
477 )
478 )
479 )
480 )
481 )
482 )
483 )
484 )
485 )
486 )
487 )
488 )
489 )
490 )
491 )
492 )
493 )
494 )
495 )
496 )
497 )
498 )
499 )
500 )
501 )
502 )
503 )
504 )
505 )
506 )
507 )
508 )
509 )
510 )
511 )
512 )
513 )
514 )
515 )
516 )
517 )
518 )
519 )
520 )
521 )
522 )
523 )
524 )
525 )
526 )
527 )
528 )
529 )
530 )
531 )
532 )
533 )
534 )
535 )
536 )
537 )
538 )
539 )
540 )
541 )
542 )
543 )
544 )
545 )
546 )
547 )
548 )
549 )
550 )
551 )
552 )
553 )
554 )
555 )
556 )
557 )
558 )
559 )
560 )
561 )
562 )
563 )
564 )
565 )
566 )
567 )
568 )
569 )
570 )
571 )
572 )
573 )
574 )
575 )
576 )
577 )
578 )
579 )
580 )
581 )
582 )
583 )
584 )
585 )
586 )
587 )
588 )
589 )
590 )
591 )
592 )
593 )
594 )
595 )
596 )
597 )
598 )
599 )
600 )
601 )
602 )
603 )
604 )
605 )
606 )
607 )
608 )
609 )
610 )
611 )
612 )
613 )
614 )
615 )
616 )
617 )
618 )
619 )
620 )
621 )
622 )
623 )
624 )
625 )
626 )
627 )
628 )
629 )
630 )
631 )
632 )
633 )
634 )
635 )
636 )
637 )
638 )
639 )
640 )
641 )
642 )
643 )
644 )
645 )
646 )
647 )
648 )
649 )
650 )
651 )
652 )
653 )
654 )
655 )
656 )
657 )
658 )
659 )
660 )
661 )
662 )
663 )
664 )
665 )
666 )
667 )
668 )
669 )
670 )
671 )
672 )
673 )
674 )
675 )
676 )
677 )
678 )
679 )
680 )
681 )
682 )
683 )
684 )
685 )
686 )
687 )
688 )
689 )
690 )
691 )
692 )
693 )
694 )
695 )
696 )
697 )
698 )
699 )
700 )
701 )
702 )
703 )
704 )
705 )
706 )
707 )
708 )
709 )
710 )
711 )
712 )
713 )
714 )
715 )
716 )
717 )
718 )
719 )
720 )
721 )
722 )
723 )
724 )
725 )
726 )
727 )
728 )
729 )
730 )
731 )
732 )
733 )
734 )
735 )
736 )
737 )
738 )
739 )
740 )
741 )
742 )
743 )
744 )
745 )
746 )
747 )
748 )
749 )
750 )
751 )
752 )
753 )
754 )
755 )
756 )
757 )
758 )
759 )
760 )
761 )
762 )
763 )
764 )
765 )
766 )
767 )
768 )
769 )
770 )
771 )
772 )
773 )
774 )
775 )
776 )
777 )
778 )
779 )
780 )
781 )
782 )
783 )
784 )
785 )
786 )
787 )
788 )
789 )
790 )
791 )
792 )
793 )
794 )
795 )
796 )
797 )
798 )
799 )
800 )
801 )
802 )
803 )
804 )
805 )
806 )
807 )
808 )
809 )
810 )
811 )
812 )
813 )
814 )
815 )
816 )
817 )
818 )
819 )
820 )
821 )
822 )
823 )
824 )
825 )
826 )
827 )
828 )
829 )
830 )
831 )
832 )
833 )
834 )
835 )
836 )
837 )
838 )
839 )
840 )
841 )
842 )
843 )
844 )
845 )
846 )
847 )
848 )
849 )
850 )
851 )
852 )
853 )
854 )
855 )
856 )
857 )
858 )
859 )
860 )
861 )
862 )
863 )
864 )
865 )
866 )
867 )
868 )
869 )
870 )
871 )
872 )
873 )
874 )
875 )
876 )
877 )
878 )
879 )
880 )
881 )
882 )
883 )
884 )
885 )
886 )
887 )
888 )
889 )
890 )
891 )
892 )
893 )
894 )
895 )
896 )
897 )
898 )
899 )
900 )
901 )
902 )
903 )
904 )
905 )
906 )
907 )
908 )
909 )
910 )
911 )
912 )
913 )
914 )
915 )
916 )
917 )
918 )
919 )
920 )
921 )
922 )
923 )
924 )
925 )
926 )
927 )
928 )
929 )
930 )
931 )
932 )
933 )
934 )
935 )
936 )
937 )
938 )
939 )
940 )
941 )
942 )
943 )
944 )
945 )
946 )
947 )
948 )
949 )
950 )
951 )
952 )
953 )
954 )
955 )
956 )
957 )
958 )
959 )
960 )
961 )
962 )
963 )
964 )
965 )
966 )
967 )
968 )
969 )
970 )
971 )
972 )
973 )
974 )
975 )
976 )
977 )
978 )
979 )
980 )
981 )
982 )
983 )
984 )
985 )
986 )
987 )
988 )
989 )
990 )
991 )
992 )
993 )
994 )
995 )
996 )
997 )
998 )
999 )
1000 )

```

```

168         ))
169     )
170     (add-cost-to-factors direct-move-to-p-cons-cost '
        direct-move-to-p-cons-cost)
171 )
172 )
173
174 ; Cost #3: as many different notes as possible
175 (print "As many different notes as possible")
176 (if (eq (species part) 0)
177     nil ; this cost has no sense for the cantus firmus as its notes are
        already fixed
178     (let ( ; for all the counterpoints
179         (variety-cost (gil::add-int-var-array-dom *sp* (* 3 (- (length (
        first (notes part))) 2)) (append '(0)(getparam-val 'variety-cost
        ))))
180     )
181     (compute-variety-cost (first (notes part)) variety-cost)
182     (add-cost-to-factors variety-cost 'variety-cost)
183 )
184 )
185 ))
186
187 ; Cost #4
188 (print "Prefer the use of the harmonic triad")
189 (if (member 4 species-list)
190     (progn ; first case, we have a fourth species counterpoint in the
        composition
191         (setq h-triad-cost (gil::add-int-var-array-dom *sp* *cf-last-index (
        append '(0) (getparam-val 'h-triad-cost)))) ; length of the cost is
        m-1 because there is m-1 notes on the third beat
192         (if (eq (species counterpoint-1) 4)
193             (if (eq (species counterpoint-2) 4)
194                 ; both are of fourth species, compute using the third beat for
        each
195                 (compute-h-triad-cost (third (h-intervals counterpoint-1)) (
        third (h-intervals counterpoint-2)) h-triad-cost)
196                 ; only the first is of fourth species, compute using the third
        beat for it
197                 (compute-h-triad-cost (third (h-intervals counterpoint-1)) (
        first (h-intervals counterpoint-2)) h-triad-cost)
198             )
199             ; only the second is of fourth species, compute using the third beat
        for it
200             (compute-h-triad-cost (first (h-intervals counterpoint-1)) (third (
        h-intervals counterpoint-2)) h-triad-cost)
201         )
202     )
203     (progn ; "normal" case: compute using the first beat for all the parts
204         (setq h-triad-cost (gil::add-int-var-array-dom *sp* *cf-len (append '(0)
        (getparam-val 'h-triad-cost)))) ; length is m as usual
205         (compute-h-triad-cost (first (h-intervals counterpoint-1)) (first (
        h-intervals counterpoint-2)) h-triad-cost)
206     )
207 )
208 (add-cost-to-factors h-triad-cost 'h-triad-cost)
209
210 (dotimes (i *N-PARTS)
211     ; Cost #5, only for 3rd species: if harmonic triad isn't achieved on the
        downbeat, it shall be on the second or third one
212     (if (or (eq (species (nth i parts)) 3) (eq (species (nth i parts)) 5)) (let
213         (

```

```

214      (h-triad-3rd-species-cost (gil::add-int-var-array-dom *sp* (* *
      cf-last-index 2) (append '(0) (getparam-val '
      h-triad-3rd-species-cost))))
215    )
216    (dotimes (j 2) (progn
217      (compute-h-triad-3rd-species-cost
218        (nth (+ j 1) (h-intervals (nth i parts))) ; 2nd or 3rd beat (=j)
219        (subseq h-triad-3rd-species-cost (* j *cf-last-index) (* (+ j 1)
        *cf-last-index))) ; these are the costs corresponding to
        the 2nd or 3rd beat (=j)
220      ))
221      (add-cost-to-factors h-triad-3rd-species-cost 'h-triad-3rd-species-cost)
222    ))
223  )
224
225  ;
  =====;
226  ;
  RETURN
  ;
227  ;
  =====;

228  (append (fux-search-engine solution-array species-list) (list species-list))
229 )

```

D.6 cf.lisp

```

1  (in-package :fuxcp)
2
3  ; Author: Anton Lamotte
4  ; Date: January 2024
5  ; This file contains the function that adds all the necessary constraints to link
6  ; the cantus firmus to the lowest stratum.
7
8  ;;=====#
9  ;; CANTUS FIRMUS          #
10 ;;=====#
11 (defun fux-cp-cf (cantus-firmus &optional (species 0))
12   (print "##### CANTUS FIRMUS RULES #####")
13   "Create the CSP for the cantus-firmus."
14
15   ;===== CREATING GIL ARRAYS
16   ;=====
17   ;; initialize the variables
18   (print "Initializing variables...")
19
20   ; creating harmonic intervals array
21   (print "Creating harmonic intervals array...")
22
23   ; array of IntVar representing the absolute intervals % 12 between the cantus
24   ; firmus and the cantus-firmus
25   (setf (first (h-intervals cantus-firmus)) (gil::add-int-var-array *sp* *cf-len 0
26   11))
27   (create-h-intervals (first (notes cantus-firmus)) (first (notes *lowest)) (first
28   (h-intervals cantus-firmus)))
29
30   ; creating melodic intervals array
31   (print "Creating melodic intervals array...")
32
33   ; array of IntVar representing the absolute intervals between two notes in a row
34   ; of the cantus-firmus

```

```

28 (setf (first (m-intervals cantus-firmus)) (gil::add-int-var-array *sp* *
    cf-last-index 0 12))
29 ;(setf (first (m-intervals-brut cantus-firmus)) (gil::add-int-var-array *sp* *
    cf-last-index -12 12))
30 (create-m-intervals-self (first (notes cantus-firmus)) (first (m-intervals
    cantus-firmus)) (first (m-intervals-brut cantus-firmus)))
31
32 ; creating perfect consonances boolean array
33 (print "Creating perfect consonances boolean array...")
34 ; array of BoolVar representing if the interval between the cantus firmus and
    the cantus-firmus is a perfect consonance
35 (setf (is-p-cons-arr cantus-firmus) (gil::add-bool-var-array *sp* *cf-len 0 1))
36 (create-is-p-cons-arr (first (h-intervals cantus-firmus)) (is-p-cons-arr
    cantus-firmus))
37
38
39 ; creating motion array
40 (print "Creating motion array...")
41 (setf (first (motions cantus-firmus)) (gil::add-int-var-array *sp* *
    cf-last-index -1 2)) ; 0 = contrary, 1 = oblique, 2 = direct/parallel
42 (setf (first (motions-cost cantus-firmus)) (gil::add-int-var-array-dom *sp* *
    cf-last-index *motions-domain*))
43 (create-motions (first (m-intervals-brut cantus-firmus)) (first (
    m-intervals-brut *lowest)) (first (motions cantus-firmus)) (first (
    motions-cost cantus-firmus)) (is-not-lowest cantus-firmus))
44 ;===== HARMONIC CONSTRAINTS
    =====
45 (print "Posting constraints...")
46
47 ; for all intervals between the cantus firmus and the cantus-firmus, the
    interval must be a consonance
48 (print "Harmonic consonances...")
49 (add-h-cons-cst *cf-len *cf-penult-index (first (h-intervals cantus-firmus)))
50
51 (if (= *N-PARTS 2) (progn
52     ; must start with a perfect consonance
53     (print "Perfect consonance at the beginning...")
54     (add-p-cons-start-cst (first (h-intervals cantus-firmus)))
55
56     ; must end with a perfect consonance
57     (print "Perfect consonance at the end...")
58     (add-p-cons-end-cst (first (h-intervals cantus-firmus)))
59
60     (print "Penultimate measure...")
61     (add-penult-cons-1sp-and-cf-cst (penult (is-not-lowest cantus-firmus)) (
        penult (first (h-intervals cantus-firmus))) 0)
62 )
63 ; else (if 3 parts)
64 (progn
65     (print "Penultimate measure...")
66     (gil::g-member *sp* PENULT_CONS_3P_VAR (penult (first (h-intervals
        cantus-firmus)))))
67 )
68 )
69
70 ;===== MELODIC CONSTRAINTS
    =====
71 ; There are no melodic constraints for the cantus firmus, as its notes are
    already fixed
72
73 ;===== MOTION CONSTRAINTS
    =====
74 (print "Motion constraints...")

```

```

75 (if (= *N-PARTS 2)
76     (add-no-direct-move-to-p-cons-cst (first (motions cantus-firmus)) (
77         is-p-cons-arr cantus-firmus) (is-not-lowest cantus-firmus))
78 )
79 ;===== COST FACTORS
80 ;=====
81 (print "Cost function...")
82 ; 1, 2) imperfect consonances are preferred to perfect consonances
83 (print "Imperfect consonances are preferred to perfect consonances...")
84 (add-p-cons-cost-cst (h-intervals cantus-firmus) (is-not-lowest cantus-firmus))
85 ; 3) motion costs
86 (print "add motion costs")
87 (add-cost-to-factors (first (motions-cost cantus-firmus)) 'motions-cost)
88 )

```

D.7 1sp-ctp.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard, adapted by Anton Lamotte
4 ; Date: June 3, 2023, adapted January 2024
5 ; This file contains the function that adds all the necessary constraints to the
6 ; first species.
7
8 ;;=====#
9 ;; FIRST SPECIES #
10 ;;=====#
11 (defun fux-cp-1st (counterpoint &optional (species 1))
12     (print "##### FIRST SPECIES #####")
13     "Create the CSP for the first species of Fux's counterpoint."
14
15     ;===== CREATING GIL ARRAYS
16     ;=====
17     ;; initialize the variables
18     (print "Initializing variables...")
19
20     ; creating harmonic intervals array
21     (print "Creating harmonic intervals array...")
22
23     ; array of IntVar representing the absolute intervals % 12 between the cantus
24     ; firmus and the counterpoint
25     (setf (first (h-intervals counterpoint)) (gil::add-int-var-array *sp* *cf-len 0
26         11))
27     (create-h-intervals (first (notes counterpoint)) (first (notes *lowest)) (first
28         (h-intervals counterpoint)))
29
30     ; creating melodic intervals array
31     (print "Creating melodic intervals array...")
32     ; array of IntVar representing the absolute intervals between two notes in a row
33     ; of the counterpoint
34     (setf (first (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
35         cf-last-index 0 12))
36
37     #| next line defined in init-counterpoint |#
38     ; (setf (first (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
39         cf-last-index -12 12))
40     (create-m-intervals-self (first (notes counterpoint)) (first (m-intervals
41         counterpoint)) (first (m-intervals-brut counterpoint)))

```

```

34 (case species ((1 3v-1st) ; only for the first species
35 ; then
36 (progn
37 ; creating melodic intervals array between the note n and n+2
38 (setf (m2-intervals counterpoint) (gil::add-int-var-array *sp* *
39 cf-penult-index 0 12))
40 (setf (m2-intervals-brut counterpoint) (gil::add-int-var-array *sp* *
41 cf-penult-index -12 12))
42 (create-m2-intervals (first (notes counterpoint)) (m2-intervals
43 counterpoint) (m2-intervals-brut counterpoint))
44
45 ; creating boolean is counterpoint off key array
46 (print "Creating is counterpoint off key array...")
47 (setf (is-cp-off-key-arr counterpoint) (gil::add-bool-var-array *sp* *
48 cf-len 0 1))
49 (create-is-member-arr (first (notes counterpoint)) (is-cp-off-key-arr
50 counterpoint) (off-domain counterpoint))
51
52 ))
53
54 ; creating perfect consonances boolean array
55 (print "Creating perfect consonances boolean array...")
56 ; array of BoolVar representing if the interval between the cantus firmus and
57 the counterpoint is a perfect consonance
58 (setf (is-p-cons-arr counterpoint) (gil::add-bool-var-array *sp* *cf-len 0 1))
59 (create-is-p-cons-arr (first (h-intervals counterpoint)) (is-p-cons-arr
60 counterpoint))
61
62
63 ; creating order/role of pitch array (if cantus firmus is higher or lower than
64 counterpoint)
65 ; 0 for being the bass, 1 for being above
66 (print "Creating order of pitch array...")
67 (setf (first (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
68 cf-len 0 1))
69 (create-is-cf-lower-arr (first (notes counterpoint)) *cf* (first (is-cf-lower-arr
70 counterpoint)))
71
72
73 ; creating motion array
74 (print "Creating motion array...")
75 (setf (first (motions counterpoint)) (gil::add-int-var-array *sp* *cf-last-index
76 -1 2)) ; 0 = contrary, 1 = oblique, 2 = direct/parallel
77 (setf (first (motions-cost counterpoint)) (gil::add-int-var-array-dom *sp* *
78 cf-last-index *motions-domain*))
79 (create-motions (first (m-intervals-brut counterpoint)) (first (m-intervals-brut
80 *lowest)) (first (motions counterpoint)) (first (motions-cost counterpoint)
81 ) (is-not-lowest counterpoint))
82
83
84 ;===== HARMONIC CONSTRAINTS
85 =====
86 (print "Posting constraints...")
87
88 ; for all intervals between the cantus firmus and the counterpoint, the interval
89 must be a consonance
90 (print "Harmonic consonances...")
91 (case species
92 ((1 3v-1sp) (add-h-cons-cst *cf-len *cf-penult-index (first (h-intervals
93 counterpoint))))
94 ((2 3v-2sp) (add-h-cons-cst *cf-len *cf-penult-index (first (h-intervals
95 counterpoint)) PENULT-THESIS-VAR))

```

```

78      ((3 3v-3sp) (add-h-cons-cst *cf-len *cf-penult-index (first (h-intervals
79      counterpoint)) PENULT_1Q_VAR))
80    )
81    ; no unison between the cantus firmus and the counterpoint unless it is the
82    first note or the last note
83    (print "No unison...")
84    (add-no-unison-cst (first (notes counterpoint)) *cf)
85
86    (case species ((1 2)
87      ; then
88      (progn
89        ; must start with a perfect consonance
90        (print "Perfect consonance at the beginning...")
91        (add-p-cons-start-cst (first (h-intervals counterpoint)))
92
93        ; must end with a perfect consonance
94        (print "Perfect consonance at the end...")
95        (add-p-cons-end-cst (first (h-intervals counterpoint)))
96      )
97    ))
98
99    ; if penultimate measure, a major sixth or a minor third must be used
100    ; depending if the cantus firmus is at the bass or on the top part
101    (print "Penultimate measure...")
102    (case species
103      ((1) (add-penult-cons-1sp-and-cf-cst (penult (is-not-lowest counterpoint)) (
104      penult (first (h-intervals counterpoint))) 1))
105      ((3v-1sp) (gil::g-member *sp* PENULT_CONS_3P_VAR (penult (first (h-intervals
106      counterpoint))))))
107    )
108    ;===== MELODIC CONSTRAINTS
109    =====
110    ; NOTE: with the degree iii in penultimate *cf measure -> no solution bc there
111    is a *tritone between I#(minor third) and V.
112    (print "Melodic constraints...")
113    (case species ((1 3v-1sp)
114      ; then
115      (progn
116        ; no more than minor sixth melodic interval
117        (print "No more than minor sixth...")
118        (add-no-m-jump-cst (first (m-intervals counterpoint)))
119
120        ; no chromatic motion between three consecutive notes
121        (print "No chromatic motion...")
122        (add-no-chromatic-m-cst (first (m-intervals-brut counterpoint)) (
123        m2-intervals-brut counterpoint))
124
125        ;===== MOTION CONSTRAINTS
126        =====
127        (print "Motion constraints...")
128        (if (eq species 1) ; for the 3v-1st species, it isn't a constraint but a
129        cost
130          ; no direct motion to reach a perfect consonance
131          (progn
132            (print "No direct motion to reach a perfect consonance...")
133            (add-no-direct-move-to-p-cons-cst (first (motions counterpoint))
134            (is-p-cons-arr counterpoint) (is-not-lowest counterpoint))
135          )
136        )
137        ; no battuta kind of motion
138        ; i.e. contrary motion to an *octave, lower voice up, higher voice down,
139        counterpoint melodic interval < -4

```

```

130         (print "No battuta kind of motion...")
131         (add-no-battuta-cst (first (motions counterpoint)) (first (h-intervals
            counterpoint)) (first (m-intervals-brut counterpoint)) (
            is-not-lowest counterpoint))
132     )
133 ))
134
135 ;===== COST FACTORS
136 ;=====
137 (print "Cost function...")
138 (case species ((1 3v-1st)
139     ; then
140     (progn
141         (setf (m-all-intervals counterpoint) (first (m-intervals counterpoint)))
142         ; 1, 2) imperfect consonances are preferred to perfect consonances
143         (print "Imperfect consonances are preferred to perfect consonances...")
144         (add-p-cons-cost-cst (h-intervals counterpoint) (is-not-lowest
            counterpoint))
145
146         ; 3, 4) add off-key cost, m-degrees cost and tritons cost
147         (print "add off-key cost, m-degrees cost and tritons cost")
148         (set-general-costs-cst counterpoint *cf-len)
149
150         ; 5) motion costs
151         (print "add motion costs")
152         (add-cost-to-factors (first (motions-cost counterpoint)) 'motions-cost)
153     )
154 ))
155
156 (setf (solution-array counterpoint) (first (notes counterpoint)))
157 (setf (solution-len counterpoint) *cf-len)
158
159 ; RETURN
160 (case species
161     (1 (append (fux-search-engine (solution-array counterpoint)) (list (list 1))
        ))
162     (otherwise nil) ; if 3v don't return a search engine, just apply the
        constraints
163 )
164 )

```

D.8 2sp-ctp.lisp

```

1  (in-package :fuxcp)
2
3  ; Author: Thibault Wafflard, adapted by Anton Lamotte
4  ; Date: June 3, 2023, adapted January 2024
5  ; This file contains the function that adds all the necessary constraints to the
    second species.
6
7  ;;=====#
8  ;; SECOND SPECIES      #
9  ;;=====#
10 ; Note: fux-cp-2nd execute the first species algorithm without some constraints.
11 ; In this function, all the variable names without the arsis-suffix refers to
    thesis notes AKA the first species notes.
12 ; All the variable names with the arsis-suffix refers to arsis notes AKA notes on
    the upbeat.
13 (defun fux-cp-2nd (counterpoint &optional (species 2))

```

```

14 "Create the CSP for the 2nd species of Fux's counterpoint, with the cantus
    firmus as input"
15 (print "##### SECOND SPECIES #####")
16
17 ;; ADD FIRST SPECIES CONSTRAINTS
18 (fux-cp-1st counterpoint species)
19 ;===== CREATION OF GIL ARRAYS
    =====
20 (print "Initializing variables...")
21
22 ; merging cp and cp-arsis into one array
23 (setf (solution-len counterpoint) (+ *cf-len *cf-last-index))
24 (setf (solution-array counterpoint) (gil::add-int-var-array *sp* (solution-len
    counterpoint) 0 127)) ; array of IntVar representing thesis and arsis notes
    combined
25 (merge-cp (list (first (notes counterpoint)) (third (notes counterpoint))) (
    solution-array counterpoint)) ; merge the two counterpoint arrays into one
26
27 ; creating harmonic intervals array
28 (print "Creating harmonic intervals array...")
29 ; array of IntVar representing the absolute intervals % 12 between the cantus
    firmus and the counterpoint (arsis notes)
30 (setf (third (h-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 0 11))
31 (create-h-intervals (third (notes counterpoint)) (butlast (first (notes *lowest)
    )) (third (h-intervals counterpoint)))
32 ; array of IntVar representing the absolute intervals (not % 12) and brut (just
    p - q)
33 ; between the cantus firmus and the counterpoint (thesis notes)
34 (setf (h-intervals-abs counterpoint) (gil::add-int-var-array *sp* *cf-len 0 127)
    )
35 (setf (h-intervals-brut counterpoint) (gil::add-int-var-array *sp* *cf-len -127
    127))
36 (create-intervals (first (notes *lowest)) (first (notes counterpoint)) (
    h-intervals-abs counterpoint) (h-intervals-brut counterpoint))
37
38
39 ; creating melodic intervals array
40 (print "Creating melodic intervals array...")
41 ; array of IntVar representing the melodic intervals between arsis note and next
    thesis note of the counterpoint
42 (setf (third (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 0 12))
43 #| next line defined in init-counterpoint |#
44 ;(setf (third (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index -12 12)); same without absolute reduction
45 (create-m-intervals-next-meas (third (notes counterpoint)) (first (notes
    counterpoint)) (third (m-intervals counterpoint)) (third (m-intervals-brut
    counterpoint)))
46 ; array of IntVar representing the melodic intervals between a thesis and an
    arsis note of the same measure the counterpoint
47 (setf (first (m-succ-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 0 12))
48 (setf (first (m-succ-intervals-brut counterpoint)) (gil::add-int-var-array *sp*
    *cf-last-index -12 12))
49 (create-m-intervals-in-meas (first (notes counterpoint)) (third (notes
    counterpoint)) (first (m-succ-intervals counterpoint)) (first (
    m-succ-intervals-brut counterpoint)))
50
51
52 ; creating melodic intervals array between the note n and n+2 for the whole
    counterpoint

```

```

53 (setf (m2-len counterpoint) (- (* *cf-last-index 2) 1)) ; number of melodic
    intervals between n and n+2 for thesis and arsis notes combined
54 (setf (m2-intervals counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) 0 12))
55 (setf (m2-intervals-brut counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) -12 12))
56 (create-m2-intervals (solution-array counterpoint) (m2-intervals counterpoint) (
    m2-intervals-brut counterpoint))
57
58 ; creating melodic intervals array between the note n and n+1 for the whole
    counterpoint
59 (setf (total-m-len counterpoint) (* *cf-last-index 2)) ; number of melodic
    intervals between n and n+1 for thesis and arsis notes combined
60 (setf (m-all-intervals counterpoint) (gil::add-int-var-array *sp* (total-m-len
    counterpoint) 0 12))
61 (setf (m-all-intervals-brut counterpoint) (gil::add-int-var-array *sp* (
    total-m-len counterpoint) -12 12))
62 (create-m-intervals-self (solution-array counterpoint) (m-all-intervals
    counterpoint) (m-all-intervals-brut counterpoint))
63
64 ; creating motion array
65 ; 0 = contrary, 1 = oblique, 2 = direct/parallel
66 (print "Creating motion array...")
67 (setf (third (motions counterpoint)) (gil::add-int-var-array *sp* *cf-last-index
    -1 2))
68 (setf (third (motions-cost counterpoint)) (gil::add-int-var-array-dom *sp* *
    cf-last-index *motions-domain*))
69 (setf (real-motions counterpoint) (gil::add-int-var-array *sp* *cf-last-index -1
    2))
70 (setf (real-motions-cost counterpoint) (gil::add-int-var-array-dom *sp* *
    cf-last-index *motions-domain*))
71 (create-motions (third (m-intervals-brut counterpoint)) (first (m-intervals-brut
    *lowest)) (third (motions counterpoint)) (third (motions-cost counterpoint)
    ) (is-not-lowest counterpoint))
72 (create-real-motions (first (m-succ-intervals counterpoint)) (first (motions
    counterpoint)) (third (motions counterpoint)) (real-motions counterpoint) (
    first (motions-cost counterpoint)) (third (motions-cost counterpoint)) (
    real-motions-cost counterpoint))
73
74 ; creating boolean diminution array
75 (print "Creating diminution array...")
76 ; Note: a diminution is the intermediate note that exists between two notes
    separated by a jump of a third
77 ; i.e. E -> D (dim) -> C
78 (setf (is-ta-dim-arr counterpoint) (gil::add-bool-var-array *sp* *cf-last-index
    0 1))
79 (create-is-ta-dim-arr (first (m-succ-intervals counterpoint)) (first (
    m-intervals counterpoint)) (third (m-intervals counterpoint)) (is-ta-dim-arr
    counterpoint))
80
81 ; creating boolean is cantus firmus bass array
82 (print "Creating is cantus firmus bass array...")
83 ; array of BoolVar representing if the cantus firmus is lower than the arsis
    counterpoint
84 (setf (third (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
    cf-last-index 0 1))
85 (create-is-cf-lower-arr (third (notes counterpoint)) (butlast *cf) (third (
    is-cf-lower-arr counterpoint)))
86
87 ; creating boolean is cantus firmus neighboring the counterpoint array
88 (print "Creating is cantus firmus neighboring array...")
89 (setf (is-nbour-arr counterpoint) (gil::add-bool-var-array *sp* *cf-last-index 0
    1))

```

```

90 (create-is-nbour-arr (h-intervals-abs counterpoint) (is-not-lowest counterpoint)
91   (first (m-intervals-brut *lowest)) (is-nbour-arr counterpoint))
92
93 ; creating boolean is counterpoint off key array
94 (print "Creating is counterpoint off key array...")
95 (setf (is-cp-off-key-arr counterpoint) (gil::add-bool-var-array *sp* (
96   solution-len counterpoint) 0 1))
97
98 (create-is-member-arr (solution-array counterpoint) (is-cp-off-key-arr
99   counterpoint) (off-domain counterpoint))
100
101 ;===== HARMONIC CONSTRAINTS
102 =====
103 (print "Posting constraints...")
104
105 (print "Harmonic consonances...")
106
107 ; for all harmonic intervals between the cantus firmus and the arsis notes, the
108   interval must be a consonance
109 ; unless the arsis note is a diminution
110 (print "No dissonance unless diminution for arsis notes...")
111 (add-h-cons-arsis-cst *cf-len *cf-penult-index (third (h-intervals counterpoint)
112   ) (is-ta-dim-arr counterpoint))
113
114 ; Fux does not follow this rule so deactivate ?
115 ; no unison between the cantus firmus and the arsis counterpoint
116 ; (print "No unison at all...")
117 ; (add-no-unison-at-all-cst (third (notes counterpoint)) (butlast (cf
118   counterpoint)))
119
120 (if (eq *N-PARTS 2) (progn
121   ; if penultimate measure, a major sixth or a minor third must be used
122   ; depending if the cantus firmus is at the bass or on the top part
123   (print "Penultimate measure...")
124   (add-penult-cons-cst (lastone (third (is-cf-lower-arr counterpoint))) (
125     lastone (third (h-intervals counterpoint))))))
126
127 (if (eq *N-PARTS 3) (progn
128   (print "Penultimate measure...")
129   (gil::g-member *sp* PENULT_CONS_3P_VAR (lastone (third (h-intervals
130     counterpoint))))))
131
132 ;===== MELODIC CONSTRAINTS
133 =====
134 (print "Melodic constraints...")
135
136 ; no more than minor sixth melodic interval between thesis and arsis notes
137   UNLESS:
138   ; - the interval between the cantus firmus and the thesis note <= major third
139   ; - the cantus firmus is getting closer to the thesis note
140 (print "No more than minor sixth melodic interval between thesis and arsis notes
141   unless...")
142 (add-m-inter-arsis-cst (first (m-succ-intervals counterpoint)) (is-nbour-arr
143   counterpoint))
144
145 ; Fux does not follow this rule, deactivate ?
146 ; (print "No more than minor sixth melodic interval between arsis and thesis
147   notes...")
148 ; (add-no-m-jump-cst (third (m-intervals counterpoint)))

```

```

139 ; no *chromatic motion between three consecutive notes
140 (print "No chromatic motion...")
141 (add-no-chromatic-m-cst (m-all-intervals-brut counterpoint) (m2-intervals-brut
    counterpoint))
142
143 ; no unison between two consecutive notes
144 (print "No unison between two consecutive notes...")
145 (case species
146   (2 (add-no-unison-at-all-cst (solution-array counterpoint) (rest (
    solution-array counterpoint))))
147   ; @completely new or reworked
148   ; ===== 2 counterpoints specific
149   (3v-2st (progn
150     ; when there is more than one counterpoint, unison can occur between the
    fourth-to-last and third-to-last note
151     (if (member 3 *species-list) (progn
152       ; when used in combination with a third species counterpoint, unison
    can also occur between the third-to-last and the
    second-to-last
153       (add-no-unison-at-all-cst (butlast (solution-array counterpoint) 3)
    (rest (butlast (solution-array counterpoint) 3))) ; no unison
    until fourth-to-last
154       (add-no-unison-at-all-cst (last (solution-array counterpoint) 2) (
    rest (last (solution-array counterpoint) 2))) ; no unison in the
    two last ones
155       (gil::g-rel *sp* (first (last (solution-array counterpoint) 4)) gil
    ::IRT_NQ (first (last (solution-array counterpoint) 2))) ; but
    the three of them cannot be a unison
156     ) (progn
157       ; when used in combination with another counterpoint (that is not of
    third species)
158       (add-no-unison-at-all-cst (butlast (solution-array counterpoint) 3)
    (rest (butlast (solution-array counterpoint) 3))) ; no unison
    until fourth-to-last
159       (add-no-unison-at-all-cst (last (solution-array counterpoint) 3) (
    rest (last (solution-array counterpoint) 3))) ; no unison in the
    three last ones
160     ))
161   ))
162   ; =====
163 )
164
165
166 ;===== MOTION CONSTRAINTS
    =====
167 (print "Motion constraints...")
168 ; no direct motion to reach a perfect consonance
169 (print "No direct motion to reach a perfect consonance...")
170 (if (eq species 2) (add-no-direct-move-to-p-cons-cst (real-motions counterpoint)
    (is-p-cons-arr counterpoint) (is-not-lowest counterpoint)))
171 ; no battuta kind of motion
172 ; i.e. contrary motion to an *octave, lower voice up, higher voice down,
    counterpoint melodic interval < -4
173 (print "No battuta kind of motion...")
174 (add-no-battuta-cst (third (motions counterpoint)) (first (h-intervals
    counterpoint)) (third (m-intervals-brut counterpoint)) (third (
    is-cf-lower-arr counterpoint)))
175
176
177
178 ;===== COST FACTORS
    =====
179 ; 1, 2) imperfect consonances are preferred to perfect consonances

```

```

180 (print "Imperfect consonances are preferred to perfect consonances...")
181 (add-p-cons-cost-cst (h-intervals counterpoint) (is-not-lowest counterpoint))
182
183 ; 3, 4) add off-key cost, m-degrees cost
184 (set-general-costs-cst counterpoint (solution-len counterpoint))
185
186 ; 5) contrary motion is preferred
187 (add-cost-to-factors (real-motions-cost counterpoint) 'motions-cost)
188
189
190 ; 6) the penultimate thesis note is not a fifth
191 (print "Penultimate thesis note is not a fifth...")
192 ; *penult-thesis-cost = *cf-len (big cost) if penultimate *h-interval /= 7
193 (setf (penult-thesis-cost counterpoint) (gil::add-int-var-dom *sp* (getparam-dom
194   'penult-sixth-cost)))
195 (add-single-cost-cst (penult (first (h-intervals counterpoint))) gil::IRT_NQ 7 (
196   penult-thesis-cost counterpoint) *penult-sixth-cost*)
197 (add-cost-to-factors (penult-thesis-cost counterpoint) 'penult-thesis-cost nil)
198
199 ;===== COST FUNCTION
200 ;=====
201 (print "Cost function...")
202
203 (case species
204   (2 (append (fux-search-engine (solution-array counterpoint) '(2)) (list (
205     list 2))))
206   (3v-2sp nil) ; if 3v don't return a search engine, just apply the
207     constraints
208 )
209 )

```

D.9 3sp-ctp.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard, adapted by Anton Lamotte
4 ; Date: June 3, 2023, adapted January 2024
5 ; This file contains the function that adds all the necessary constraints to the
6   third species.
7
8 ;;=====#
9 ;; THIRD SPECIES #
10 ;;=====#
11 ;; Note: fux-cp-3rd execute the first species algorithm without some constraints.
12 ;; In this function, 4 quarter notes by measure are assumed.
13 (defun fux-cp-3rd (counterpoint &optional (species 3))
14   "Create the CSP for the 3rd species of Fux's counterpoint, with the cantus
15    firmus as input"
16   (print "##### THIRD SPECIES #####")
17   (print "Creating the CSP for the 3rd species of Fux's counterpoint...")
18
19   ;; ADD FIRST SPECIES CONSTRAINTS
20   (fux-cp-1st counterpoint species)
21   ;===== CREATION OF GIL ARRAYS
22   ;=====
23   (print "Initializing variables...")
24
25   (loop for i from 1 to 3 do
26     (setf i-1 (- i 1))
27     ; creating harmonic intervals array

```

```

26 ; array of IntVar representing the absolute intervals % 12 between the
    cantus firmus and the counterpoint
27 (setf (nth i (h-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 0 11))
28 (create-h-intervals (nth i (notes counterpoint)) (butlast (first (notes *
    lowest))) (nth i (h-intervals counterpoint)))
29
30 (setf (nth i (h-intervals-to-cf counterpoint)) (gil::add-int-var-array *sp*
    *cf-last-index 0 11))
31 (create-h-intervals (nth i (notes counterpoint)) (butlast *cf) (nth i (
    h-intervals-to-cf counterpoint)))
32
33 ; array of IntVar representing the absolute intervals between a thesis and
    an arsis note of the same measure the counterpoint
34 (setf (nth i-1 (m-succ-intervals counterpoint)) (gil::add-int-var-array *sp*
    *cf-last-index 1 12))
35 (setf (nth i-1 (m-succ-intervals-brut counterpoint)) (gil::add-int-var-array
    *sp* *cf-last-index -12 12))
36 (create-intervals (nth i-1 (notes counterpoint)) (nth i (notes counterpoint)
    ) (nth i-1 (m-succ-intervals counterpoint)) (nth i-1 (
    m-succ-intervals-brut counterpoint)))
37 )
38
39 ; merging cp and cp-arsis into one array
40 (print "Mergin cps...")
41 (setf (solution-len counterpoint) (+ *cf-len (* *cf-last-index 3))) ; total
    length of the counterpoint array
42 (setf (solution-array counterpoint) (gil::add-int-var-array *sp* (solution-len
    counterpoint) 0 127)) ; array of IntVar representing thesis and arsis notes
    combined
43 (merge-cp (notes counterpoint) (solution-array counterpoint)) ; merge the four
    counterpoint arrays into one
44 ; creating melodic intervals array
45 (print "Creating melodic intervals array...")
46 ; array of IntVar representing the absolute intervals
47 ; between the last note of measure m and the first note of measure m+1 of the
    counterpoint
48 (setf (fourth (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 1 12))
49 #| next line defined in init-counterpoint |#
50 ; (setf (fourth (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index -12 12)) ; same without absolute reduction
51 (create-m-intervals-next-meas (fourth (notes counterpoint)) (first (notes
    counterpoint)) (fourth (m-intervals counterpoint)) (fourth (m-intervals-brut
    counterpoint)))
52
53 ; creating melodic intervals array between the note n and n+2 for the whole
    counterpoint
54 (setf (m2-len counterpoint) (- (* *cf-last-index 4) 1)) ; number of melodic
    intervals between n and n+2 for the total counterpoint
55 (setf (m2-intervals counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) 0 12))
56 (setf (m2-intervals-brut counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) -12 12))
57 (create-m2-intervals (solution-array counterpoint) (m2-intervals counterpoint) (
    m2-intervals-brut counterpoint))
58
59 ; creating melodic intervals array between the note n and n+1 for the whole
    counterpoint
60 (setf (total-m-len counterpoint) (* *cf-last-index 4)) ; number of melodic
    intervals between n and n+1 for the total counterpoint
61 (setf (m-all-intervals counterpoint) (gil::add-int-var-array *sp* (total-m-len
    counterpoint) 0 12))

```

```

62 (setf (m-all-intervals-brut counterpoint) (gil::add-int-var-array *sp* (
    total-m-len counterpoint) -12 12))
63 (create-m-intervals-self (solution-array counterpoint) (m-all-intervals
    counterpoint) (m-all-intervals-brut counterpoint))
64 ; creating motion array
65 ; 0 = contrary, 1 = oblique, 2 = direct/parallel
66 (print "Creating motion array...")
67 (setf (fourth (motions counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index -1 2))
68 (setf (fourth (motions-cost counterpoint)) (gil::add-int-var-array-dom *sp* *
    cf-last-index *motions-domain*))
69 (create-motions (fourth (m-intervals-brut counterpoint)) (first (
    m-intervals-brut *lowest)) (fourth (motions counterpoint)) (fourth (
    motions-cost counterpoint)) (is-not-lowest counterpoint))
70
71 ; creating boolean is cantus firmus bass array
72 (print "Creating is cantus firmus bass array...")
73 ; array of BoolVar representing if the cantus firmus is lower than the arsis
    counterpoint
74 (setf (fourth (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
    cf-last-index 0 1))
75 (create-is-cf-lower-arr (fourth (notes counterpoint)) (butlast *cf) (fourth (
    is-cf-lower-arr counterpoint)))
76
77 ; creating boolean are five consecutive notes by joint degree array
78 (print "Creating are five consecutive notes by joint degree array...")
79 ; array of BoolVar representing if the five consecutive notes are by joint
    degree
80 (setf (is-5qn-linked-arr counterpoint) (gil::add-bool-var-array *sp* *
    cf-last-index 0 1))
81 (create-is-5qn-linked-arr (m-all-intervals counterpoint) (m-all-intervals-brut
    counterpoint) (is-5qn-linked-arr counterpoint))
82
83 ; creating boolean diminution array
84 (print "Creating diminution array...")
85 ; Note: a diminution is the intermediate note that exists between two notes
    separated by a jump of a third
86 ; i.e. E -> D (dim) -> C
87 (setf (is-ta-dim-arr counterpoint) (gil::add-bool-var-array *sp* *cf-last-index
    0 1))
88 (create-is-ta-dim-arr (second (m-succ-intervals counterpoint)) (collect-by-4 (
    m2-intervals counterpoint) 1 T) (third (m-succ-intervals counterpoint)) (
    is-ta-dim-arr counterpoint))
89
90 ; creating boolean is consonant array
91 (print "Creating is consonant array...")
92 (loop for i from 0 to 3 do
93     ; array of BoolVar representing if the interval is consonant
94     (if (eq i 0)
95         (setf (nth i (is-cons-arr counterpoint)) (gil::add-bool-var-array *sp* *
            cf-len 0 1))
96         (setf (nth i (is-cons-arr counterpoint)) (gil::add-bool-var-array *sp* *
            cf-last-index 0 1))
97     )
98     (create-is-member-arr (nth i (h-intervals counterpoint)) (nth i (is-cons-arr
        counterpoint)))
99 )
100
101
102 ; creating boolean is not cambiata array
103 (print "Creating is not cambiata array...")
104 (setf (is-not-cambiata-arr counterpoint) (gil::add-bool-var-array *sp* *
    cf-last-index 0 1))

```

```

105 (create-is-not-cambiata-arr (second (is-cons-arr counterpoint)) (third (
    is-cons-arr counterpoint)) (second (m-succ-intervals counterpoint)) (
    is-not-cambiata-arr counterpoint))
106
107 ; creating boolean is counterpoint off key array
108 (print "Creating is counterpoint off key array...")
109 (setf (is-cp-off-key-arr counterpoint) (gil::add-bool-var-array *sp* (
    solution-len counterpoint) 0 1))
110 (create-is-member-arr (solution-array counterpoint) (is-cp-off-key-arr
    counterpoint) (off-domain counterpoint))
111
112
113 ;===== HARMONIC CONSTRAINTS
    =====
114 (print "Posting constraints...")
115 (if (eq *N-PARTS 2) (progn
116     ; must start with a perfect consonance
117     (print "Perfect consonance at the beginning...")
118     (add-p-cons-start-cst (first (h-intervals counterpoint)))
119
120     ; must end with a perfect consonance
121     (print "Perfect consonance at the end...")
122     (add-p-cons-end-cst (first (h-intervals counterpoint)))
123
124     ; if penultimate measure, a major sixth or a minor third must be used
125     ; depending if the cantus firmus is at the bass or on the top part
126     (print "Penultimate measure...")
127     (add-penult-cons-cst (lastone (fourth (is-cf-lower-arr counterpoint))) (
        lastone (fourth (h-intervals-to-cf counterpoint))))
128     ))
129
130 (if (eq *N-PARTS 3) (progn
131     (print "Penultimate measure...")
132     (gil::g-member *sp* PENULT_CONS_3P_VAR (lastone (fourth (h-intervals
        counterpoint)))))
133     ))
134
135 ; the third note of the penultimate measure must be below the fourth one.
136 (gil::g-rel *sp* (lastone (third (m-succ-intervals-brut counterpoint))) gil::
    IRT_GR 1)
137
138 ; the second note and the third note of the penultimate measure must be distant
    by greater than 1 semi-tone from the fourth note
139 (gil::g-rel *sp* (penult (m2-intervals counterpoint)) gil::IRT_NQ 1)
140
141 ; five consecutive notes by joint degree implies that the first and the third
    note are consonants
142 (print "Five consecutive notes by joint degree...")
143 (add-linked-5qn-cst (third (is-cons-arr counterpoint)) (is-5qn-linked-arr
    counterpoint))
144
145 ; any dissonant note implies that it is surrounded by consonant notes
146 (print "Any dissonant note...")
147 (add-h-dis-or-cons-3rd-cst (second (is-cons-arr counterpoint)) (third (
    is-cons-arr counterpoint)) (fourth (is-cons-arr counterpoint)) (
    is-ta-dim-arr counterpoint))
148
149 ;===== MELODIC CONSTRAINTS
    =====
150 (print "Melodic constraints...")
151
152 ; no melodic interval between 9 and 11
153 (loop for m in (m-succ-intervals counterpoint) do

```

```

154      (add-no-m-jump-extend-cst m)
155    )
156    (add-no-m-jump-extend-cst (fourth (m-intervals counterpoint)))
157
158    ; no *chromatic motion between three consecutive notes
159    (print "No chromatic motion...")
160    (add-no-chromatic-m-cst (m-all-intervals-brut counterpoint) (m2-intervals-brut
      counterpoint))
161
162    ; Marcel's rule: contrary melodic step after skip
163    (print "Marcel's rule...")
164    (add-contrary-step-after-skip-cst (m-all-intervals counterpoint) (
      m-all-intervals-brut counterpoint))
165
166    ;===== MOTION CONSTRAINTS
167    =====
168    (print "Motion constraints...")
169
170    ; no direct motion to reach a perfect consonance
171    (print "No direct motion to reach a perfect consonance...")
172    (if (eq *N-PARTS 2) (add-no-direct-move-to-p-cons-cst (fourth (motions
      counterpoint)) (is-p-cons-arr counterpoint) (is-not-lowest counterpoint)))
173
174    ; no battuta kind of motion
175    ; i.e. contrary motion to an *octave, lower voice up, higher voice down,
      counterpoint melodic interval < -4
176    (print "No battuta kind of motion...")
177    (add-no-battuta-cst (fourth (motions counterpoint)) (first (h-intervals-to-cf
      counterpoint)) (fourth (m-intervals-brut counterpoint)) (fourth (
      is-cf-lower-arr counterpoint)))
178
179    ;===== COST FACTORS
180    =====
181    ; 1, 2) imperfect consonances are preferred to perfect consonances
182    (print "Imperfect consonances are preferred to perfect consonances...")
183    (add-p-cons-cost-cst (h-intervals counterpoint) (is-not-lowest counterpoint))
184
185    ; 3, 4) add off-key cost, m-degrees cost and tritons cost
186    (set-general-costs-cst counterpoint (solution-len counterpoint))
187
188    ; 5) contrary motion is preferred
189    (add-cost-to-factors (fourth (motions-cost counterpoint)) 'motions-cost)
190
191    ; 6) cambiata notes are preferred (cons - dis - cons > cons - cons - cons)
192    (print "Cambiata notes are preferred...")
193    ; IntVar array representing the cost to have cambiata notes
194    (setf (not-cambiata-cost counterpoint) (gil::add-int-var-array-dom *sp* *
      cf-last-index (getparam-dom 'non-cambiata-cost)))
195    (add-cost-bool-cst (is-not-cambiata-arr counterpoint) (not-cambiata-cost
      counterpoint) *non-cambiata-cost*)
196    (add-cost-to-factors (not-cambiata-cost counterpoint) 'not-cambiata-cost)
197
198    ; 7) intervals between notes n and n+2 are preferred greater than zero
199    (print "Intervals between notes n and n+2 are preferred different than zero...")
200    ; IntVar array representing the cost to have intervals between notes n and n+2
      equal to zero
201    (setf (m2-eq-zero-cost counterpoint) (gil::add-int-var-array-dom *sp* (m2-len
      counterpoint) (getparam-dom 'm2-eq-zero-cost)))
202    (add-cost-cst (m2-intervals counterpoint) gil::IRT-EQ 0 (m2-eq-zero-cost
      counterpoint) *m2-eq-zero-cost*)
203    (add-cost-to-factors (m2-eq-zero-cost counterpoint) 'm2-eq-zero-cost)
204
205    ;===== COST FUNCTION
206    =====

```

```

203 (print "Cost function...")
204
205 (case species
206   (3 (append (fux-search-engine (solution-array counterpoint) '(3)) (list (
207     list 3))))
208   (3v-3sp nil) ; if 3v don't return a search engine, just apply the
209     constraints
210 )
211 )

```

D.10 4sp-ctp.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard, adapted by Anton Lamotte
4 ; Date: June 3, 2023, adapted January 2024
5 ; This file contains the function that adds all the necessary constraints to the
6   fourth species.
7
8 ;;=====#
9 ;; FOURTH SPECIES      #
10 ;;=====#
11 ;; Note: fux-cp-4th execute the first species algorithm without some constraints.
12 ;; In this function, the first notes are in Arsīs because of the syncopation.
13 (defun fux-cp-4th (counterpoint &optional (species 4))
14   "Create the CSP for the 2nd species of Fux's counterpoint, with the cantus
15    firmus as input"
16
17   (print "##### FOURTH SPECIES #####")
18
19   ;===== CREATION OF GIL ARRAYS
20   ;=====
21   (print "Initializing variables...")
22
23   ; merging cp and cp-arsis into one array
24   (setf (solution-len counterpoint) (* *cf-last-index 2))
25   (setf (solution-array counterpoint) (gil::add-int-var-array *sp* (solution-len
26     counterpoint) 0 127)) ; array of IntVar representing thesis and arsis notes
27     combined
28   (merge-cp-same-len (list (third (notes counterpoint)) (first (notes counterpoint
29     ))) (solution-array counterpoint)) ; merge the two counterpoint arrays into
30     one
31
32   ; creating harmonic intervals array
33   (print "Creating harmonic intervals array...")
34   ; array of IntVar representing the absolute intervals % 12 between the cantus
35     firmus and the counterpoint (arsis notes)
36   (setf (third (h-intervals counterpoint)) (gil::add-int-var-array *sp* *
37     cf-last-index 0 11))
38   (setf (first (h-intervals counterpoint)) (gil::add-int-var-array *sp* *
39     cf-last-index 0 11))
40   (create-h-intervals (third (notes counterpoint)) (butlast (first (notes *lowest)
41     ))) (third (h-intervals counterpoint)))
42   (create-h-intervals (first (notes counterpoint)) (rest (first (notes *lowest))))
43     (first (h-intervals counterpoint)))
44
45   (setf (third (h-intervals-to-cf counterpoint)) (gil::add-int-var-array *sp* *
46     cf-last-index 0 11))
47   (create-h-intervals (third (notes counterpoint)) (butlast *cf) (third (
48     h-intervals-to-cf counterpoint)))

```

```

36
37
38 ; creating melodic intervals array
39 (print "Creating melodic intervals array...")
40 ; array of IntVar representing the melodic intervals between arsis and next
    thesis note of the counterpoint
41 (setf (third (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index 0 8))
42
43 #| next line defined in init-counterpoint |#
44 ; (setf (third (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
    cf-last-index -12 12)) ; same without absolute reduction
45 (create-intervals (third (notes counterpoint)) (first (notes counterpoint)) (
    third (m-intervals counterpoint)) (third (m-intervals-brut counterpoint)))
46 ; array of IntVar representing the melodic intervals between a thesis and an
    arsis note of the same measure the counterpoint
47 (setf (first (m-succ-intervals counterpoint)) (gil::add-int-var-array *sp* *
    cf-penult-index 1 12))
48 (setf (first (m-succ-intervals-brut counterpoint)) (gil::add-int-var-array *sp*
    *cf-penult-index -12 12))
49 (create-m-intervals-in-meas (first (notes counterpoint)) (rest (third (notes
    counterpoint))) (first (m-succ-intervals counterpoint)) (first (
    m-succ-intervals-brut counterpoint)))
50
51
52 ; creating melodic intervals array between the note n and n+2 for the whole
    counterpoint
53 (setf (m2-len counterpoint) (- (* *cf-last-index 2) 2)) ; number of melodic
    intervals between n and n+2 for thesis and arsis notes combined
54 (setf (m2-intervals counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) 0 12))
55 (setf (m2-intervals-brut counterpoint) (gil::add-int-var-array *sp* (m2-len
    counterpoint) -12 12))
56 (create-m2-intervals (solution-array counterpoint) (m2-intervals counterpoint) (
    m2-intervals-brut counterpoint))
57
58 ; creating melodic intervals array between the note n and n+1 for the whole
    counterpoint
59 (setf (total-m-len counterpoint) (- (* *cf-last-index 2) 1)) ; number of melodic
    intervals between n and n+1 for thesis and arsis notes combined
60 (setf (m-all-intervals counterpoint) (gil::add-int-var-array *sp* (total-m-len
    counterpoint) 0 12))
61 (setf (m-all-intervals-brut counterpoint) (gil::add-int-var-array *sp* (
    total-m-len counterpoint) -12 12))
62 (create-m-intervals-self (solution-array counterpoint) (m-all-intervals
    counterpoint) (m-all-intervals-brut counterpoint))
63
64 ; creating perfect consonances boolean array
65 (print "Creating perfect consonances boolean array...")
66 ; array of BoolVar representing if the interval between the cantus firmus and
    the counterpoint is a perfect consonance
67 (setf (is-p-cons-arr counterpoint) (gil::add-bool-var-array *sp* *cf-len 0 1))
68 (create-is-p-cons-arr (first (h-intervals counterpoint)) (is-p-cons-arr
    counterpoint))
69
70 ; creating boolean is cantus firmus bass array
71 (print "Creating is cantus firmus bass array...")
72 ; array of BoolVar representing if the cantus firmus is lower than the arsis
    counterpoint
73 (setf (third (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
    cf-last-index 0 1))
74 (create-is-cf-lower-arr (third (notes counterpoint)) (butlast *cf) (third (
    is-cf-lower-arr counterpoint)))

```

```

75
76 ; creating boolean is counterpoint off key array
77 (print "Creating is counterpoint off key array...")
78 (setf (is-cp-off-key-arr counterpoint) (gil::add-bool-var-array *sp* (
79   solution-len counterpoint) 0 1))
80
81 ; creating boolean is consonant array
82 (print "Creating is consonant array...")
83 ; array of BoolVar representing if the interval is consonant
84 (setf (first (is-cons-arr counterpoint)) (gil::add-bool-var-array *sp* *
85   cf-last-index 0 1))
86 (create-is-member-arr (first (h-intervals counterpoint)) (first (is-cons-arr
87   counterpoint)))
88
89 ; creation boolean is no syncope array
90 (print "Creating is no syncope array...")
91 ; array of BoolVar representing if the thesis note is note related to the
92   previous one
93 (setf (is-no-syncope-arr counterpoint) (gil::add-bool-var-array *sp* *
94   cf-penult-index 0 1))
95 (create-is-no-syncope-arr (third (m-intervals counterpoint)) (is-no-syncope-arr
96   counterpoint))
97
98 ;===== HARMONIC CONSTRAINTS
99   =====
100 (print "Posting constraints...")
101
102 ; for all harmonic intervals between the cantus firmus and the thesis notes, the
103   interval must be a consonance
104 (print "Harmonic consonances...")
105 ; here the penultimate thesis note must be a seventh or a second and the arsis
106   note must be a major sixth or a minor third
107 (add-penult-dom-cst (penult (first (h-intervals counterpoint)))
108   PENULT_SYNCOPE_VAR)
109
110 (add-h-cons-cst *cf-last-index *cf-penult-index (third (h-intervals counterpoint)
111   )) PENULT_CONS_VAR 4 (is-not-lowest counterpoint))
112 (add-no-sync-h-cons (first (h-intervals counterpoint)) (is-no-syncope-arr
113   counterpoint))
114
115 ; no seventh dissonance if the cantus firmus is at the top
116 (print "No seventh dissonance if the cantus firmus is at the top...")
117 (add-no-seventh-cst (first (h-intervals counterpoint)) (is-not-lowest
118   counterpoint))
119
120 (if (eq *N-PARTS 2) (progn
121   ; must start with a perfect consonance
122   (print "Perfect consonance at the beginning...")
123   (add-p-cons-start-cst (third (h-intervals counterpoint)))
124
125   ; must end with a perfect consonance
126   (print "Perfect consonance at the end...")
127   (add-p-cons-end-cst (first (h-intervals counterpoint)))
128
129   ; if penultimate measure, a major sixth or a minor third must be used
130   ; depending if the cantus firmus is at the bass or on the top part
131   (print "Penultimate measure...")
132   (add-penult-cons-cst (lastone (third (is-cf-lower-arr counterpoint))) (
133     lastone (third (h-intervals-to-cf counterpoint))))))
134 ))

```

```

123
124 (if (eq *N-PARTS 3) (progn
125   (print "Penultimate measure...")
126   (gil::g-member *sp* PENULT_SYNCOPES_VAR (lastone (third (h-intervals
127     counterpoint))))
128 ))
129
130 ;===== MELODIC CONSTRAINTS
131 ;=====
132 (print "Melodic constraints...")
133
134 ; melodic intervals cannot be greater than a minor sixth expect the octave
135 (print "No more than minor sixth melodic interval between arsis and thesis notes
136   ...")
137 (add-no-m-jump-extend-cst (first (m-succ-intervals counterpoint)))
138
139 ; no *chromatic motion between three consecutive notes
140 (print "No chromatic motion...")
141 (add-no-chromatic-m-cst (m-all-intervals-brut counterpoint) (m2-intervals-brut
142   counterpoint))
143
144 ;===== MOTION CONSTRAINTS
145 ;=====
146 (print "Motion constraints...")
147
148 ; dissonant notes must be followed by the consonant note below
149 (print "Dissonant notes must be followed by the consonant note below...")
150
151 (add-h-dis-imp-cons-below-cst (first (m-succ-intervals-brut counterpoint)) (
152   first (is-cons-arr counterpoint)))
153
154 ; no second dissonance if the cantus firmus is at the bass and a octave/unison
155 precedes it
156 (print "No second dissonance if the cantus firmus is at the bass...")
157 (add-no-second-cst (third (h-intervals counterpoint)) (first (h-intervals
158   counterpoint)) (is-not-lowest counterpoint))
159
160 ;===== COST FACTORS
161 ;=====
162 (print "Cost factors...")
163
164 ; 1, 2) imperfect consonances are preferred to perfect consonances
165 (add-p-cons-cost-cst (h-intervals counterpoint) (is-not-lowest counterpoint) t)
166
167 ; 3, 4) add off-key cost, m-degrees cost and tritons cost
168 (set-general-costs-cst counterpoint (solution-len counterpoint))
169
170 ; 5) add no syncopation cost
171 (print "No syncopation cost...")
172 (setf (no-syncopation-cost counterpoint) (gil::add-int-var-array-dom *sp* *
173   cf-penult-index (getparam-dom 'no-syncopation-cost)))
174 (add-cost-cst (butlast (third (m-intervals counterpoint))) gil::IRT_NQ 0 (
175   no-syncopation-cost counterpoint) *no-syncopation-cost*)
176 (add-cost-to-factors (no-syncopation-cost counterpoint) 'no-syncopation-cost)
177
178 ; 6) add m2-intervals equal to 0 cost
179 (print "Monotonia...")
180 (setf (m2-eq-zero-cost counterpoint) (gil::add-int-var-array-dom *sp* (- *cf-len
181   3) (getparam-dom 'm2-eq-zero-cost)))
182 (add-cost-multi-cst (third (notes counterpoint)) gil::IRT_EQ (caddr (third (notes
183   counterpoint))) (m2-eq-zero-cost counterpoint) *m2-eq-zero-cost*)
184 (add-cost-to-factors (m2-eq-zero-cost counterpoint) 'm2-eq-zero-cost)

```

```

173
174 ;===== COST FUNCTION
175 ;=====
176 (print "Cost function...")
177
178 ; RETURN
179 (if (eq species 4)
180     ; then create the search engine
181     (append (fux-search-engine (solution-array counterpoint) '(4)) (list (list
182         4)))
183     ; else if 3v
184     nil
185 )
186 )

```

D.11 5sp-ctp.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard, adapted by Anton Lamotte
4 ; Date: June 3, 2023, adapted January 2024
5 ; This file contains the function that adds all the necessary constraints to the
6 ; fifth species.
7
8 ;;=====#
9 ;; FIFTH SPECIES #
10 ;;=====#
11 ;; Note: fux-cp-5th execute the first species algorithm without some constraints.
12 ;; In this function, 4 notes by measure are assumed.
13 (defun fux-cp-5th (counterpoint &optional (species 5))
14   "Create the CSP for the 3rd species of Fux's counterpoint, with the cantus
15   firmus as input"
16   (print "Creating the CSP for the 3rd species of Fux's counterpoint...")
17
18   ;; CLEANING PREVIOUS SOLUTIONS
19   (setf *prev-sol-check nil)
20   (setf rhythmic+pitch nil)
21   (setf rhythmic-om nil)
22   (setf pitches-om nil)
23
24   (print "##### FIFTH SPECIES #####")
25
26   ;===== CREATION OF BOOLEAN SPECIES ARRAYS
27   ;=====
28   (print "Creation of boolean species arrays...")
29   ; total length of the counterpoint array
30   (setf (solution-len counterpoint) (+ *cf-len (* *cf-last-index 3)))
31   ; array representing the species type [0: no constraint, 1: 1st species, 2: 2nd
32   ; species, 3: 3rd species, 4: 4th species]
33   (setf (species-arr counterpoint) (gil::add-int-var-array *sp* (solution-len
34       counterpoint) 0 4))
35   (create-species-arr (species-arr counterpoint) (solution-len counterpoint))
36   ; arrays representing if a note is constraint by a species
37   (setf (nth 0 (is-nth-species-arr counterpoint)) (gil::add-bool-var-array *sp* (
38       solution-len counterpoint) 0 1))
39   (create-simple-boolean-arr (species-arr counterpoint) gil::IRT_EQ 0 (nth 0 (
40       is-nth-species-arr counterpoint)))
41   (setf (nth 1 (is-nth-species-arr counterpoint)) (gil::add-bool-var-array *sp* (
42       solution-len counterpoint) 0 1))
43   (create-simple-boolean-arr (species-arr counterpoint) gil::IRT_EQ 1 (nth 1 (
44       is-nth-species-arr counterpoint)))

```

```

36 (setf (nth 2 (is-nth-species-arr counterpoint)) (gil::add-bool-var-array *sp* (
    solution-len counterpoint) 0 1))
37 (create-simple-boolean-arr (species-arr counterpoint) gil::IRT_EQ 2 (nth 2 (
    is-nth-species-arr counterpoint)))
38 (setf (nth 3 (is-nth-species-arr counterpoint)) (gil::add-bool-var-array *sp* (
    solution-len counterpoint) 0 1))
39 (create-simple-boolean-arr (species-arr counterpoint) gil::IRT_EQ 3 (nth 3 (
    is-nth-species-arr counterpoint)))
40 (setf (nth 4 (is-nth-species-arr counterpoint)) (gil::add-bool-var-array *sp* (
    solution-len counterpoint) 0 1))
41 (create-simple-boolean-arr (species-arr counterpoint) gil::IRT_EQ 4 (nth 4 (
    is-nth-species-arr counterpoint)))
42
43 ; creating boolean is constrained array
44 (print "Creating is constrained array...")
45 ; array of BoolVar representing if the interval is constrained
46 (setf (is-constrained-arr counterpoint) (collect-not-array (nth 0 (
    is-nth-species-arr counterpoint)))))
47
48
49 ;===== CREATION OF GIL ARRAYS
    =====
50 (print "Initializing variables...")
51
52 (loop for i from 0 to 3 do
53   (if (eq i 0)
54     (progn
55       ; creating harmonic intervals array
56       (print "Creating harmonic intervals array...")
57       ; array of IntVar representing the absolute intervals % 12 between
58       ; the cantus firmus and the counterpoint
59       (setf (nth i (h-intervals counterpoint)) (gil::add-int-var-array *sp*
60         *cf-len 0 11))
61       (create-h-intervals (nth i (notes counterpoint)) (first (notes *
62         lowest)) (nth i (h-intervals counterpoint)))
63
64       (setf (nth i (h-intervals-to-cf counterpoint)) (gil::
65         add-int-var-array *sp* *cf-len 0 11))
66       (create-h-intervals (nth i (notes counterpoint)) *cf (nth i (
67         h-intervals-to-cf counterpoint)))
68
69       )
70     (progn
71       ; same as above but 1 note shorter
72       (setf (nth i (h-intervals counterpoint)) (gil::add-int-var-array *sp*
73         *cf-last-index 0 11))
74       (create-h-intervals (nth i (notes counterpoint)) (butlast (first (
75         notes *lowest))) (nth i (h-intervals counterpoint)))
76
77       (setf (nth i (h-intervals-to-cf counterpoint)) (gil::
78         add-int-var-array *sp* *cf-last-index 0 11))
79       (create-h-intervals (nth i (notes counterpoint)) (butlast *cf) (nth
80         i (h-intervals-to-cf counterpoint)))
81
82       )
83     )
84   )
85 )
86
87 (loop for i from 0 to 2 do
88   (setq i+1 (+ i 1))
89   (setf (nth i (m-succ-intervals-brut counterpoint)) (gil::add-int-var-array *
90     sp* *cf-last-index -12 12))
91   (if (eq i 1)

```

```

80         ; then melodic interval could be 0 if there was a dissonant syncope
           before (see that later)
81         (setf (nth i (m-succ-intervals counterpoint)) (gil::add-int-var-array *
           sp* *cf-last-index 0 12))
82         ; else no melodic interval of 0
83         (setf (nth i (m-succ-intervals counterpoint)) (gil::add-int-var-array *
           sp* *cf-last-index 0 12))
84     )
85     (create-intervals (nth i (notes counterpoint)) (nth i+1 (notes counterpoint)
           ) (nth i (m-succ-intervals counterpoint)) (nth i (m-succ-intervals-brut
           counterpoint)))
86 )
87
88
89 ; merging all cp arrays into one
90 (print "Merging cps...")
91 (setf (solution-array counterpoint) (gil::add-int-var-array *sp* (solution-len
           counterpoint) 0 127)) ; array of IntVar representing thesis and arsis notes
           combined
92 (merge-cp (notes counterpoint) (solution-array counterpoint)) ; merge the four
           counterpoint arrays into one
93
94 ; creating melodic intervals array
95 (print "Creating melodic intervals array...")
96 ; array of IntVar representing the melodic intervals between arsis and next
           thesis note of the counterpoint
97 (setf (third (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
           cf-last-index 0 16))
98 ;(setf (third (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
           cf-last-index -16 16)) ; same without absolute reduction
99 (create-m-intervals-next-meas (third (notes counterpoint)) (first (notes
           counterpoint)) (third (m-intervals counterpoint)) (third (m-intervals-brut
           counterpoint)))
100 ; array of IntVar representing the absolute intervals
101 ; between the last note of measure m and the first note of measure m+1 of the
           counterpoint
102 (setf (fourth (m-intervals counterpoint)) (gil::add-int-var-array *sp* *
           cf-last-index 0 12)) ; can be 0 if this is replace by 2 eight note
103
104 #| next line defined in init-counterpoint |#
105 ; (setf (fourth (m-intervals-brut counterpoint)) (gil::add-int-var-array *sp* *
           cf-last-index -12 12)) ; same without absolute reduction
106 (create-m-intervals-next-meas (fourth (notes counterpoint)) (first (notes
           counterpoint)) (fourth (m-intervals counterpoint)) (fourth (m-intervals-brut
           counterpoint)))
107
108 ; array of IntVar representing the melodic intervals between the thesis note and
           the arsis note of the same measure
109 (setf (m-ta-intervals counterpoint) (gil::add-int-var-array *sp* *cf-last-index
           0 16))
110 (setf (m-ta-intervals-brut counterpoint) (gil::add-int-var-array *sp* *
           cf-last-index -16 16)) ; same without absolute reduction
111 (create-intervals (first (notes counterpoint)) (third (notes counterpoint)) (
           m-ta-intervals counterpoint) (m-ta-intervals-brut counterpoint))
112
113 ; creating melodic intervals array between the note n and n+2 for the whole
           counterpoint
114 (setf (m2-len counterpoint) (- (* *cf-last-index 4) 1)) ; number of melodic
           intervals between n and n+2 for the total counterpoint
115 (setf (m2-intervals counterpoint) (gil::add-int-var-array *sp* (m2-len
           counterpoint) 0 16))
116 (setf (m2-intervals-brut counterpoint) (gil::add-int-var-array *sp* (m2-len
           counterpoint) -16 16))

```

```

117 (create-m2-intervals (solution-array counterpoint) (m2-intervals counterpoint) (
      m2-intervals-brut counterpoint))
118
119 ; creating melodic intervals array between the note n and n+1 for the whole
      counterpoint
120 (setf (total-m-len counterpoint) (* *cf-last-index 4)) ; number of melodic
      intervals between n and n+1 for the total counterpoint
121 (setf (m-all-intervals counterpoint) (gil::add-int-var-array *sp* (total-m-len
      counterpoint) 0 12))
122 (setf (m-all-intervals-brut counterpoint) (gil::add-int-var-array *sp* (
      total-m-len counterpoint) -12 12))
123 (create-m-intervals-self (solution-array counterpoint) (m-all-intervals
      counterpoint) (m-all-intervals-brut counterpoint) (is-constrained-arr
      counterpoint))
124
125 ; creating motion array
126 ; 0 = contrary, 1 = oblique, 2 = direct/parallel
127 (print "Creating motion array...")
128 (setf (fourth (motions counterpoint)) (gil::add-int-var-array *sp* *
      cf-last-index -1 2))
129 (setf (fourth (motions-cost counterpoint)) (gil::add-int-var-array-dom *sp* *
      cf-last-index *motions-domain*))
130 (create-motions (fourth (m-intervals-brut counterpoint)) (first (
      m-intervals-brut *lowest)) (fourth (motions counterpoint)) (fourth (
      motions-cost counterpoint)) (is-not-lowest counterpoint))
131
132 ; creating boolean is cantus firmus bass array
133 (print "Creating is cantus firmus bass array...")
134 ; array of BoolVar representing if the cantus firmus is lower than the arsis
      counterpoint
135 (setf (first (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
      cf-len 0 1))
136 (create-is-cf-lower-arr (first (notes counterpoint)) (rest *cf) (first (
      is-cf-lower-arr counterpoint))) ; 5th
137 (setf (third (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
      cf-last-index 0 1))
138 (create-is-cf-lower-arr (third (notes counterpoint)) (butlast *cf) (third (
      is-cf-lower-arr counterpoint))) ; 5th
139 (setf (fourth (is-cf-lower-arr counterpoint)) (gil::add-bool-var-array *sp* *
      cf-last-index 0 1))
140 (create-is-cf-lower-arr (fourth (notes counterpoint)) (butlast *cf) (fourth (
      is-cf-lower-arr counterpoint)))
141
142 ; creating boolean are five consecutive notes by joint degree array
143 (print "Creating are five consecutive notes by joint degree array...")
144 ; array of BoolVar representing if the five consecutive notes are by joint
      degree
145 (setf (is-5qn-linked-arr counterpoint) (gil::add-bool-var-array *sp* *
      cf-last-index 0 1))
146 (create-is-5qn-linked-arr (m-all-intervals counterpoint) (m-all-intervals-brut
      counterpoint) (is-5qn-linked-arr counterpoint))
147 (setf (is-mostly-3rd-arr counterpoint) (gil::add-bool-var-array *sp* *
      cf-last-index 0 1)) ; 5th
148 (create-is-mostly-3rd-arr (nth 3 (is-nth-species-arr counterpoint)) (
      is-mostly-3rd-arr counterpoint))
149
150 ; creating boolean is consonant array + species array
151 (print "Creating is consonant array and species array...")
152 (loop for i from 0 to 3 do
153   ; array of BoolVar representing if the interval is consonant
154   (if (eq i 0)
155     (progn

```

```

156         (setf (nth i (is-cons-arr counterpoint)) (gil::add-bool-var-array *
157             sp* *cf-len 0 1))
158         (setf (nth i (is-3rd-species-arr counterpoint)) (gil::
159             add-bool-var-array *sp* *cf-len 0 1))
160         (setf (nth i (is-4th-species-arr counterpoint)) (gil::
161             add-bool-var-array *sp* *cf-len 0 1))
162         (setf (nth i (is-cst-arr counterpoint)) (gil::add-bool-var-array *sp
163             * *cf-len 0 1))
164     )
165     (progn
166         (setf (nth i (is-cons-arr counterpoint)) (gil::add-bool-var-array *
167             sp* *cf-last-index 0 1))
168         (setf (nth i (is-3rd-species-arr counterpoint)) (gil::
169             add-bool-var-array *sp* *cf-last-index 0 1))
170         (setf (nth i (is-4th-species-arr counterpoint)) (gil::
171             add-bool-var-array *sp* *cf-last-index 0 1))
172         (setf (nth i (is-cst-arr counterpoint)) (gil::add-bool-var-array *sp
173             * *cf-last-index 0 1))
174     )
175     (create-is-member-arr (nth i (h-intervals counterpoint)) (nth i (is-cons-arr
176         counterpoint)))
177     (create-by-4 (nth 3 (is-nth-species-arr counterpoint)) (nth i (
178         is-3rd-species-arr counterpoint)) i)
179     (create-by-4 (nth 4 (is-nth-species-arr counterpoint)) (nth i (
180         is-4th-species-arr counterpoint)) i)
181     (create-by-4 (is-constrained-arr counterpoint) (nth i (is-cst-arr
182         counterpoint)) i)
183 )
184
185 ; creating boolean diminution array
186 (print "Creating diminution array...")
187 ; Note: a diminution is the intermediate note that exists between two notes
188 ; separated by a jump of a third
189 ; i.e. E -> D (dim) -> C
190 (setf (is-ta-dim-arr counterpoint) (gil::add-bool-var-array *sp* *cf-last-index
191     0 1))
192 (create-is-ta-dim-arr (second (m-succ-intervals counterpoint)) (collect-by-4 (
193     m2-intervals counterpoint) 1 T) (third (m-succ-intervals counterpoint)) (
194     is-ta-dim-arr counterpoint))
195
196 ; creating boolean is not cambiata array
197 (print "Creating is not cambiata array...")
198 (setf (is-not-cambiata-arr counterpoint) (gil::add-bool-var-array *sp* *
199     cf-last-index 0 1))
200 (create-is-not-cambiata-arr (second (is-cons-arr counterpoint)) (third (
201     is-cons-arr counterpoint)) (second (m-succ-intervals counterpoint)) (
202     is-not-cambiata-arr counterpoint))
203
204 ; creating boolean is counterpoint off key array
205 (print "Creating is counterpoint off key array...")
206 (setf (is-cp-off-key-arr counterpoint) (gil::add-bool-var-array *sp* (
207     solution-len counterpoint) 0 1))
208 (create-is-member-arr (solution-array counterpoint) (is-cp-off-key-arr
209     counterpoint) (off-domain counterpoint))
210
211 ; creating perfect consonances boolean array
212 (print "Creating perfect consonances boolean array...")
213 ; array of BoolVar representing if the interval between the cantus firmus and
214 ; the counterpoint is a perfect consonance
215 (setf (is-p-cons-arr counterpoint) (gil::add-bool-var-array *sp* *cf-len 0 1))
216 (create-is-p-cons-arr (first (h-intervals counterpoint)) (is-p-cons-arr
217     counterpoint))

```

```

196 ; creation boolean is no syncope array
197 (print "Creating is no syncope array...")
198 ; array of BoolVar representing if the thesis note is note related to the
199 previous one
200 (setf (is-no-syncope-arr counterpoint) (gil::add-bool-var-array *sp* *
201 cf-penult-index 0 1))
202 (create-is-no-syncope-arr (third (m-intervals counterpoint)) (is-no-syncope-arr
203 counterpoint))
204
205 ;===== HARMONIC CONSTRAINTS
206 =====
207 (print "Posting constraints...")
208
209 ; one possible value for non-constrained notes
210 (print "One possible value for non-constrained notes...")
211 (add-one-possible-value-cst (solution-array counterpoint) (nth 0 (
212 is-nth-species-arr counterpoint)))
213
214 (if (eq *N-PARTS 2) (progn
215 ; perfect consonances should be used at the start and at the end of the
216 piece
217 (print "Perfect consonances at the start and at the end...")
218 ; if first note is constrained then it must be a perfect consonance
219 (add-p-cons-cst-if (first (first (h-intervals counterpoint))) (first (
220 is-constrained-arr counterpoint)))
221 ; if first note is not constrained then the third note must be a perfect
222 consonance
223 (add-p-cons-cst-if (first (third (h-intervals counterpoint))) (first (nth 0
224 (is-nth-species-arr counterpoint))))
225 ; no matter what species it is, the last harmonic interval must be a perfect
226 consonance
227 (add-p-cons-end-cst (first (h-intervals counterpoint)))
228
229 ; if penultimate measure, a major sixth or a minor third must be used
230 ; depending if the cantus firmus is at the bass or on the top part
231 (print "Penultimate measure...")
232 (add-penult-cons-cst (lastone (fourth (is-cf-lower-arr counterpoint))) (
233 lastone (fourth (h-intervals-to-cf counterpoint)))
234 (penult (nth 3 (is-nth-species-arr counterpoint)))
235 ) ; 3rd species
236 ))
237 ; the third note of the penultimate measure must be below the fourth one. (3rd
238 species)
239 (gil::g-rel-reify *sp* (lastone (third (m-succ-intervals-brut counterpoint)))
240 gil::IRT_GR 1
241 (penult (nth 3 (is-nth-species-arr counterpoint))) gil::RM_IMP
242 ) ; 3rd species
243 ; the second note and the third note of the penultimate measure must be
244 ; distant by greater than 1 semi-tone from the fourth note (3rd species)
245 (gil::g-rel-reify *sp* (penult (m2-intervals counterpoint)) gil::IRT_NQ 1
246 (nth (total-index *cf-penult-index 1) (nth 3 (is-nth-species-arr
247 counterpoint))) gil::RM_IMP
248 ) ; 3rd species
249 (if (eq *N-PARTS 2) (progn
250 ; for the 4th species, the thesis note must be a seventh or a second and the
251 arsis note must be a major sixth or a minor third
252 ; major sixth or minor third
253 (add-penult-cons-cst (lastone (third (is-cf-lower-arr counterpoint))) (
254 lastone (third (h-intervals-to-cf counterpoint)))
255 (penult (butlast (nth 4 (is-nth-species-arr counterpoint)))))
256 ) ; 4th species

```

```

243         ; seventh or second
244         ; (note: a => !b <=> !(a ^ b)), so here we use the negation of the
           conjunction
245     ))
246
247     (if (eq *N-PARTS 3) (progn
248         (print "Penultimate measure...")
249         (gil::g-member *sp* PENULT_CONS_3P_VAR (lastone (third (h-intervals
           counterpoint)))))
250     ))
251
252     (setf is-penult-cons-to-cf (gil::add-bool-var *sp* 0 1))
253     (add-is-member-cst (penult (first (h-intervals-to-cf counterpoint)))
           ALL_CONS_VAR is-penult-cons-to-cf)
254     (gil::g-op *sp* (penult (first (is-4th-species-arr counterpoint))) gil::BOT_AND
           is-penult-cons-to-cf 0) ; 4th species
255
256     ; every thesis note should be consonant if it does not belong to the fourth
           species (or not constrained at all)
257     (print "Every thesis note should be consonant...")
258     (add-h-cons-cst-if (first (is-cons-arr counterpoint)) (collect-by-4 (nth 1 (
           is-nth-species-arr counterpoint)))) ; 1st species
259     (add-h-cons-cst-if (first (is-cons-arr counterpoint)) (collect-by-4 (nth 2 (
           is-nth-species-arr counterpoint)))) ; 2nd species
260     (add-h-cons-cst-if (first (is-cons-arr counterpoint)) (first (is-3rd-species-arr
           counterpoint))) ; 3rd species
261     (add-h-cons-cst-if (third (is-cons-arr counterpoint)) (third (is-4th-species-arr
           counterpoint))) ; 4th species
262     (add-h-cons-cst-if (first (is-cons-arr counterpoint)) (collect-bot-array (rest (
           first (is-4th-species-arr counterpoint))) (is-no-syncope-arr counterpoint)))
           ; 4th species
263
264     ; five consecutive notes by joint degree implies that the first and the third
           note are consonants
265     (print "Five consecutive notes by joint degree...") ; 3rd species
266     (add-linked-5qn-cst (third (is-cons-arr counterpoint)) (collect-bot-array (
           is-5qn-linked-arr counterpoint) (is-mostly-3rd-arr counterpoint)))
267
268     ; any dissonant note implies that it is surrounded by consonant notes
269     (print "Any dissonant note...") ; 3rd species
270     (add-h-dis-or-cons-3rd-cst
271         (second (is-cons-arr counterpoint))
272         (collect-t-or-f-array (third (is-cons-arr counterpoint)) (third (
           is-3rd-species-arr counterpoint)))
273         (fourth (is-cons-arr counterpoint))
274         (is-ta-dim-arr counterpoint)
275     )
276
277     ; no seventh dissonance if the cantus firmus is at the top
278     (print "No seventh dissonance if the cantus firmus is at the top...")
279     (add-no-seventh-cst (first (h-intervals counterpoint)) (is-not-lowest
           counterpoint) (first (is-4th-species-arr counterpoint))) ; 4th species
280
281
282     ;===== MELODIC CONSTRAINTS
           =====
283     (print "Melodic constraints...")
284
285     ; no melodic interval between 9 and 11
286     (add-no-m-jump-extend-cst (m-all-intervals counterpoint) (collect-bot-array (
           butlast (is-constrained-arr counterpoint)) (rest (is-constrained-arr
           counterpoint)))))
287

```

```

288 ; no unison between two consecutive notes
289 ; except for in the second part or the fourth part of the measure
290 (print "No unison between two consecutive notes...")
291 ; if 1st note and 2nd note exists (it means it belongs to a species)
292 (add-no-unison-at-all-cst
293   (first (notes counterpoint)) (second (notes counterpoint))
294   (collect-bot-array (first (is-cst-arr counterpoint)) (second (is-cst-arr
    counterpoint))))
295 ) ; 5th
296 (add-no-unison-at-all-cst
297   (third (notes counterpoint)) (fourth (notes counterpoint))
298   (collect-bot-array (third (is-cst-arr counterpoint)) (fourth (is-cst-arr
    counterpoint))))
299 ) ; 5th
300
301 ; melodic intervals between thesis and arsis note from the same measure
302 ; can't be greater than a minor sixth expect the octave (just for the fourth
    species)
303 (print "No more than minor sixth melodic interval between arsis and thesis notes
    ...")
304 ; only applied if the the second note is not constrained
305 (add-no-m-jump-extend-cst (m-ta-intervals counterpoint) (collect-by-4 (nth 0 (
    is-nth-species-arr counterpoint)) 1)) ; 4th species
306
307 ; no same syncopation if 4th species
308 (add-no-same-syncopation-cst (first (notes counterpoint)) (third (notes
    counterpoint)) (collect-bot-array (first (is-4th-species-arr counterpoint))
    (third (is-cst-arr counterpoint))))
309
310
311 ;===== MOTION CONSTRAINTS
    =====
312 (print "Motion constraints...")
313
314 ; no direct motion to reach a perfect consonance
315 (print "No direct motion to reach a perfect consonance...")
316 (if (eq species 5) (add-no-direct-move-to-p-cons-cst (fourth (motions
    counterpoint)) (collect-bot-array (is-p-cons-arr counterpoint) (fourth (
    is-3rd-species-arr counterpoint))) (is-not-lowest counterpoint) nil)) ; 3rd
    species
317
318 ; no battuta kind of motion
319 ; i.e. contrary motion to an *octave, lower voice up, higher voice down,
    counterpoint melodic interval < -4
320 (print "No battuta kind of motion...")
321 (add-no-battuta-cst
322   (fourth (motions counterpoint)) (first (h-intervals counterpoint)) (fourth (
    m-intervals-brut counterpoint)) (fourth (is-cf-lower-arr counterpoint))
    (fourth (is-3rd-species-arr counterpoint)))
323 ) ; 3rd species
324
325 ; dissonant notes must be followed by the consonant note below
326 (print "Dissonant notes must be followed by the consonant note below...")
327 (add-h-dis-imp-cons-below-cst (m-ta-intervals-brut counterpoint) (first (
    is-cons-arr counterpoint)) (first (is-4th-species-arr counterpoint))) ; TODO
    4th species
328
329 ; no second dissonance if the cantus firmus is at the bass and a octave/unison
    precedes it
330 (print "No second dissonance if the cantus firmus is at the bass...")
331 (add-no-second-cst
332   (third (h-intervals counterpoint)) (rest (first (h-intervals counterpoint)))
    (rest (is-not-lowest counterpoint)))

```

```

333     (rest (first (is-4th-species-arr counterpoint)))
334 ) ; TODO 4th species
335
336 ; Marcel's rule
337 (add-contrary-step-after-skip-cst (m-all-intervals counterpoint) (
    m-all-intervals-brut counterpoint))
338
339
340 ;===== COST FACTORS
    =====
341 (print "Imperfect consonances are preferred to perfect consonances...")
342 (setf (fifth-cost counterpoint) (gil::add-int-var-array-dom *sp* *cf-len (
    getparam-dom 'h-fifth-cost))) ; IntVar array representing the cost to have
    fifths
343 (setf (octave-cost counterpoint) (gil::add-int-var-array-dom *sp* *cf-len (
    getparam-dom 'h-octave-cost))) ; IntVar array representing the cost to have
    octaves
344 (add-cost-cst-if (first (h-intervals counterpoint)) gil::IRT_EQ 7 (first (
    is-cst-arr counterpoint)) (fifth-cost counterpoint) *h-fifth-cost*) ; (
    fifth-cost counterpoint) = 1 if *h-interval == 7
345 (let ((is-cst-and-not-bass-arr (gil::add-bool-var-array *sp* *cf-len 0 1)))
346     (dotimes (i *cf-len)
347         (gil::g-op *sp* (nth i (first (is-cst-arr counterpoint))) gil::BOT_AND (
            nth i (is-not-lowest counterpoint)) (nth i is-cst-and-not-bass-arr))
348     )
349     (add-cost-cst-if (first (h-intervals counterpoint)) gil::IRT_EQ 0
        is-cst-and-not-bass-arr (octave-cost counterpoint) *h-octave-cost*) ; (
        octave-cost counterpoint) = 1 if *h-interval == 0
350 )
351 (add-cost-to-factors (fifth-cost counterpoint) 'fifth-cost)
352 (add-cost-to-factors (octave-cost counterpoint) 'octave-cost)
353
354 ; 3, 4) add off-key cost, m-degrees cost and tritons cost
355 (set-general-costs-cst counterpoint (solution-len counterpoint) (
    is-constrained-arr counterpoint) (collect-bot-array (butlast (
    is-constrained-arr counterpoint)) (rest (is-constrained-arr counterpoint))))
356
357 ; 5) contrary motion is preferred
358 (add-cost-to-factors (fourth (motions-cost counterpoint)) 'motions-cost)
359
360 ; 6) cambiata notes are preferred (cons - dis - cons > cons - cons - cons)
361 (print "Cambiata notes are preferred...")
362 ; IntVar array representing the cost to have cambiata notes
363 (setf (not-cambiata-cost counterpoint) (gil::add-int-var-array-dom *sp* *
    cf-last-index (getparam-dom 'non-cambiata-cost)))
364 (add-cost-bool-cst-if (is-not-cambiata-arr counterpoint) (is-mostly-3rd-arr
    counterpoint) (not-cambiata-cost counterpoint) *non-cambiata-cost*)
365 (add-cost-to-factors (not-cambiata-cost counterpoint) 'non-cambiata-cost)
366
367 ; 7) intervals between notes n and n+2 are preferred greater than zero
368 (print "Intervals between notes n and n+2 are preferred different than zero...")
369 ; IntVar array representing the cost to have intervals between notes n and n+2
    equal to zero
370 (setf (m2-eq-zero-cost counterpoint) (gil::add-int-var-array-dom *sp* (m2-len
    counterpoint) (getparam-dom 'm2-eq-zero-cost)))
371 (add-cost-cst-if
    (m2-intervals counterpoint) gil::IRT_EQ 0
    (collect-bot-array (butlast (butlast (is-constrained-arr counterpoint))) (
        rest (rest (is-constrained-arr counterpoint))))
    (m2-eq-zero-cost counterpoint) *m2-eq-zero-cost*
372 )
373 (add-cost-to-factors (m2-eq-zero-cost counterpoint) 'm2-eq-zero-cost)
374
375
376
377

```

```

378 ; 8) add no syncopation cost
379 (setf (no-syncopation-cost counterpoint) (gil::add-int-var-array-dom *sp* *
      cf-penult-index (getparam-dom 'no-syncopation-cost)))
380 (add-cost-cst-if
381   (butlast (third (m-intervals counterpoint))) gil::IRT_NQ 0
382   (third (is-4th-species-arr counterpoint))
383   (no-syncopation-cost counterpoint)
384   *no-syncopation-cost*
385 )
386 (add-cost-to-factors (no-syncopation-cost counterpoint) 'no-syncopation-cost)
387
388
389 ;===== COST FUNCTION
      =====
390 (print "Cost function...")
391
392 (loop for i from 0 to 3 do
393   (setf (nth i (cons-cost counterpoint)) (gil::add-int-var-array *sp* *
      cf-last-index 0 1)) ; IntVar representing the cost to have a consonance
394   (add-cost-bool-cst (nth i (is-cons-arr counterpoint)) (nth i (cons-cost
      counterpoint))) ; (cons-cost counterpoint) = 1 if (is-cons-arr
      counterpoint) == 1
395 )
396
397 ; RETURN
398 (if (eq species 5)
399   ; then create the search engine
400   (append (fux-search-engine (solution-array counterpoint) '(5)) (list (list
      5)) (voice-type counterpoint))
401   ; else if 3v
402   nil
403 )
404 )

```

D.12 constraints.lisp

```

1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard, adapted by Anton Lamotte
4 ; Date: June 3, 2023, adapted January 2024
5 ; This file contains all the functions adding constraints to the CSP.
6 ; They are all called from the different species.
7
8
9 ;===== CP CONSTRAINTS UTILS
      =====
10
11
12 ; add a single cost regarding if the relation rel-type(tested, cst-val) is true
13 (defun add-single-cost-cst (tested rel-type cst-val cost &optional (cost-value ONE))
14   (let (
15     (b (gil::add-bool-var *sp* 0 1)) ; to store the result of the test
16   )
17     (gil::g-rel-reify *sp* tested rel-type cst-val b) ; test the relation
18     (gil::g-ite *sp* b cost-value ZERO cost) ; add the cost if the test is true
19   )
20 )
21
22 ; add a cost regarding if the relation rel-type(tested-var, cst-val) is true
23 (defun add-cost-cst (tested-var-arr rel-type cst-val costs &optional (cost-value ONE)
  )

```

```

24   (loop
25     for cost in costs
26     for tested in tested-var-arr
27     do
28       (add-single-cost-cst tested rel-type cst-val cost cost-value)
29   )
30 )
31
32 ; add a cost regarding if the relation rel-type(tested-var, cst-val) is true
33 ; NOTE: the difference with add-cost-cst is that the cst-val is an array
34 (defun add-cost-multi-cst (tested-var-arr rel-type cst-val-arr costs &optional (
35   cost-value ONE))
36   (loop
37     for cost in costs
38     for tested in tested-var-arr
39     for cst-val in cst-val-arr
40     do
41       (add-single-cost-cst tested rel-type cst-val cost cost-value)
42   )
43 )
44 ; add a cost regarding if the relation rel-type(tested-var, cst-val) is true AND
45   is-cst is true
46 (defun add-cost-cst-if (tested-var-arr rel-type cst-val is-cst-arr costs &optional (
47   cost-value ONE))
48   (loop
49     for cost in costs
50     for tested in tested-var-arr
51     for is-cst in is-cst-arr
52     do
53       (add-single-cost-cst-if tested rel-type cst-val is-cst cost cost-value)
54   )
55 )
56 (defun add-single-cost-cst-if (tested rel-type cst-val is-cst cost cost-value)
57   (let (
58     (b (gil::add-bool-var *sp* 0 1)) ; to store the result of the test
59     (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
60   )
61     (gil::g-rel-reify *sp* tested rel-type cst-val b)
62     (gil::g-op *sp* b gil::BOT_AND is-cst b-and) ; b-and = b and cst
63     (gil::g-ite *sp* b-and cost-value ZERO cost) ; add the cost if the test is
64       true
65   )
66 )
67 ; add a cost regarding if the booleans are true in bool-arr
68 (defun add-cost-bool-cst (bool-arr costs &optional (cost-value ONE))
69   (loop
70     for b in bool-arr
71     for cost in costs
72     do
73       (gil::g-ite *sp* b cost-value ZERO cost)
74   )
75 )
76 ; add a cost regarding if the booleans are true in bool-arr AND if is-cst is true in
77   is-cst-arr
78 (defun add-cost-bool-cst-if (bool-arr is-cst-arr costs &optional (cost-value ONE))
79   (loop
80     for b in bool-arr
81     for cst in is-cst-arr
82     for cost in costs

```

```

82         do
83             (add-single-cost-bool-cst-if b cst cost cost-value)
84         )
85     )
86
87 ; add a cost regarding if b is true AND if cst is true
88 (defun add-single-cost-bool-cst-if (b cst cost cost-value)
89     (let (
90         (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
91         )
92         (gil::g-op *sp* b gil::BOT_AND cst b-and) ; b-and = b and cst
93         (gil::g-ite *sp* b-and cost-value ZERO cost) ; add the cost if the test is
          true
94     )
95 )
96
97 ; add a cost regarding only if b AND cst are true (do not force ZERO if false)
98 (defun add-single-cost-bool-cst-eqv (b cst cost cost-value)
99     (let (
100         (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
101         )
102         (gil::g-op *sp* b gil::BOT_AND cst b-and) ; b-and = b and cst
103         (gil::g-rel-reify *sp* cost gil::IRT_EQ cost-value b-and gil::RM_IMP) ; add
          the cost if the test is true
104     )
105 )
106
107 ; add constraints such that costs =
108 ; - 0 if m-degree in [0, 1, 2]
109 ; - 1 if m-degree in [3, 4, 12]
110 ; - 2 otherwise
111 ; @m-all-intervals: all the melodic intervals of cp in a row
112 ; @m-degrees-cost: the cost of each melodic interval
113 (defun add-m-degrees-cost-cst (m-all-intervals m-degrees-cost m-degrees-type &
    optional (is-cst-arr nil))
114     (loop
115         for m in m-all-intervals
116         for c in m-degrees-cost
117         for d in m-degrees-type
118         do
119             (let (
120                 (b-l3 (gil::add-bool-var *sp* 0 1)) ; true if m < 3
121                 (b-3 (gil::add-bool-var *sp* 0 1)) ; true if m == 3
122                 (b-4 (gil::add-bool-var *sp* 0 1)) ; true if m == 4
123                 (b-34 (gil::add-bool-var *sp* 0 1)) ; true if m in [3, 4]
124                 (b-5 (gil::add-bool-var *sp* 0 1)) ; true if m == 5
125                 (b-6 (gil::add-bool-var *sp* 0 1)) ; true if m == 6
126                 (b-7 (gil::add-bool-var *sp* 0 1)) ; true if m == 7
127                 (b-8 (gil::add-bool-var *sp* 0 1)) ; true if m == 8
128                 (b-9 (gil::add-bool-var *sp* 0 1)) ; true if m == 9
129                 (b-89 (gil::add-bool-var *sp* 0 1)) ; true if m in [8, 9]
130                 (b-10 (gil::add-bool-var *sp* 0 1)) ; true if m == 10
131                 (b-11 (gil::add-bool-var *sp* 0 1)) ; true if m == 11
132                 (b-1011 (gil::add-bool-var *sp* 0 1)) ; true if m in [10, 11]
133                 (b-12 (gil::add-bool-var *sp* 0 1)) ; true if m == 12
134             )
135             (gil::g-rel-reify *sp* m gil::IRT_LE 3 b-l3) ; m < 3
136             (gil::g-rel-reify *sp* m gil::IRT_EQ 3 b-3) ; m = 3
137             (gil::g-rel-reify *sp* m gil::IRT_EQ 4 b-4) ; m = 4
138             (gil::g-op *sp* b-3 gil::BOT_OR b-4 b-34) ; m in [3, 4]
139             (gil::g-rel-reify *sp* m gil::IRT_EQ 5 b-5) ; m = 5
140             (gil::g-rel-reify *sp* m gil::IRT_EQ 6 b-6) ; m = 6
141             (gil::g-rel-reify *sp* m gil::IRT_EQ 7 b-7) ; m = 7

```

```

142 (gil::g-rel-reify *sp* m gil::IRT_EQ 8 b-8) ; m = 8
143 (gil::g-rel-reify *sp* m gil::IRT_EQ 9 b-9) ; m = 9
144 (gil::g-op *sp* b-8 gil::BOT_OR b-9 b-89) ; m in [8, 9]
145 (gil::g-rel-reify *sp* m gil::IRT_EQ 10 b-10) ; m = 10
146 (gil::g-rel-reify *sp* m gil::IRT_EQ 11 b-11) ; m = 11
147 (gil::g-op *sp* b-10 gil::BOT_OR b-11 b-1011) ; m in [10, 11]
148 (gil::g-rel-reify *sp* m gil::IRT_EQ 12 b-12) ; m = 12
149 ; set costs
150 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-step-cost* b-l3 gil::RM_IMP)
151 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-third-cost* b-34 gil::RM_IMP)
152 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-fourth-cost* b-5 gil::RM_IMP)
153 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-tritone-cost* b-6 gil::RM_IMP)
154 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-fifth-cost* b-7 gil::RM_IMP)
155 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-sixth-cost* b-89 gil::RM_IMP)
156 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-seventh-cost* b-1011 gil::RM_IMP
    )
157 (gil::g-rel-reify *sp* c gil::IRT_EQ *m-octave-cost* b-12 gil::RM_IMP)
158 ; set types
159 (gil::g-rel-reify *sp* d gil::IRT_EQ 2 b-l3 gil::RM_IMP)
160 (gil::g-rel-reify *sp* d gil::IRT_EQ 3 b-34 gil::RM_IMP)
161 (gil::g-rel-reify *sp* d gil::IRT_EQ 4 b-5 gil::RM_IMP)
162 (gil::g-rel-reify *sp* d gil::IRT_EQ 1 b-6 gil::RM_IMP)
163 (gil::g-rel-reify *sp* d gil::IRT_EQ 5 b-7 gil::RM_IMP)
164 (gil::g-rel-reify *sp* d gil::IRT_EQ 6 b-89 gil::RM_IMP)
165 (gil::g-rel-reify *sp* d gil::IRT_EQ 7 b-1011 gil::RM_IMP)
166 (gil::g-rel-reify *sp* d gil::IRT_EQ 8 b-12 gil::RM_IMP)
167 )
168 )
169 )
170
171 ; add cost constraints such that a cost is added when a fifth or an octave is
    present in the 1st beat
172 ; except for the 4th species where it is the 3rd beat
173 ; @is-sync: true means it is the 4th species
174 (defun add-p-cons-cost-cst (h-intervals is-not-lowest &optional (is-sync nil))
175   (setq fifth-cost (gil::add-int-var-array-dom *sp* *cf-penult-index (
    getparam-dom 'h-fifth-cost))) ; IntVar array representing the cost to have
    fifths
176   (setq octave-cost (gil::add-int-var-array-dom *sp* *cf-penult-index (
    getparam-dom 'h-octave-cost))) ; IntVar array representing the cost to have
    octaves
177   (if is-sync
178     ; then 4th species
179     (add-h-inter-cost-cst (rest (third h-intervals)) fifth-cost octave-cost (
    rest is-not-lowest))
180     ; else
181     (add-h-inter-cost-cst (restbutlast (first h-intervals)) fifth-cost
    octave-cost (restbutlast is-not-lowest))
182   )
183   (add-cost-to-factors fifth-cost 'h-fifth-cost)
184   (add-cost-to-factors octave-cost 'h-octave-cost)
185 )
186
187 ; add cost constraints such that a cost is added when a fifth or an octave is
    present in @h-intervals
188 (defun add-h-inter-cost-cst (h-intervals fifth-cost octave-cost is-not-lowest)
189   (add-cost-cst h-intervals gil::IRT_EQ 7 fifth-cost *h-fifth-cost*) ; *
    fifth-cost = 1 if *h-interval == 7
190   (add-cost-cst-if h-intervals gil::IRT_EQ 0 is-not-lowest octave-cost *
    h-octave-cost*) ; *octave-cost = 1 if *h-interval == 0
191 )
192

```

```

193 ; Get the minimum cost possible for a counterpoint depending on the costs of the
    melodic intervals
194 ; @m-len: number of melodic intervals
195 (defun get-min-m-cost (m-len)
196   ; get the minimum cost for skips
197   (setq min-skip-cost (min
198     (getparam 'm-third-cost)
199     (getparam 'm-fourth-cost)
200     (getparam 'm-tritone-cost)
201     (getparam 'm-fifth-cost)
202     (getparam 'm-sixth-cost)
203     (getparam 'm-seventh-cost)
204     (getparam 'm-octave-cost)
205   ))
206   ; get the minimum number of skips
207   (setq int-min-skip (ceiling (* (getparam 'min-skips-slider) m-len)))
208   ; return the minimum cost
209   (+
210     (* int-min-skip min-skip-cost)
211     (* (- m-len int-min-skip) (min (getparam 'm-step-cost) min-skip-cost))
212   )
213 )
214
215 ; Initializes the cost factors, accordingly to the species and the number of voices
216 (defun set-cost-factors ()
217   (setq *N-COST-FACTORS 3)
218   (case *N-PARTS
219     (2 (case (first *species-list)
220       (1 (incf *N-COST-FACTORS 5))
221       (2 (incf *N-COST-FACTORS 6))
222       (3 (incf *N-COST-FACTORS 7))
223       (4 (incf *N-COST-FACTORS 6))
224       (5 (incf *N-COST-FACTORS 8))
225     ))
226     (3 (progn
227       (incf *N-COST-FACTORS 7)
228       (dolist (species *species-list)
229         (case species
230           (1 (incf *N-COST-FACTORS 5))
231           (2 (incf *N-COST-FACTORS 6))
232           (3 (incf *N-COST-FACTORS 8)) ; + 7 from fux-cp-3rd and + 1 from
              fux-cp-3v
233           (4 (incf *N-COST-FACTORS 5)) ; + 6 from fux-cp-4th and -1 not
              used in fux-cp-3v
234           (5 (incf *N-COST-FACTORS 9)) ; + 7 from fux-cp-5th and + 1 from
              fux-cp-3v
235           (otherwise (error "Unexpected value in the species list (~A), when
              setting the costs." species))
236         )
237       )
238     ))
239   )
240   (gil::add-int-var-array *sp* *N-COST-FACTORS 0 100)
241 )
242
243 ; add general costs for most of the species
244 (defun set-general-costs-cst (counterpoint cp-len &optional (is-cst-arr1 nil) (
    is-cst-arr2 nil))
245   (let (
246     (m-len (- cp-len 1))
247   )
248     ; 2) sharps and flats should be used sparingly
249     (print "Sharps and flats should be used sparingly...")

```

```

250 (setf (off-key-cost counterpoint) (gil::add-int-var-array-dom *sp* cp-len (
    getparam-dom 'borrow-cost))) ; IntVar array representing the cost to
    have off-key notes
251 (if (null is-cst-arr1)
252     ; then
253     (add-cost-bool-cst (is-cp-off-key-arr counterpoint) (off-key-cost
        counterpoint) *borrow-cost*)
254     ; else
255     (add-cost-bool-cst-if (is-cp-off-key-arr counterpoint) is-cst-arr1 (
        off-key-cost counterpoint) *borrow-cost*))
256 )
257 ; sum of the cost of the off-key notes
258 (add-cost-to-factors (off-key-cost counterpoint) 'borrow-cost)
259
260 ; 3) melodic intervals should be as small as possible
261 (print "Melodic intervals should be as small as possible...")
262 ; IntVar array representing the cost to have melodic large intervals
263 (setq degrees-cost-domain
264     (remove-duplicates (mapcar (lambda (x) (getparam x))
265         (list 'm-step-cost 'm-third-cost 'm-fourth-cost 'm-tritone-cost '
            m-fifth-cost 'm-sixth-cost 'm-seventh-cost 'm-octave-cost)
266     ))
267 )
268 (setf (m-degrees-cost counterpoint) (gil::add-int-var-array-dom *sp* m-len
    degrees-cost-domain))
269 (setf (m-degrees-type counterpoint) (gil::add-int-var-array *sp* m-len 1 8))
270 (add-m-degrees-cost-cst (m-all-intervals counterpoint) (m-degrees-cost
    counterpoint) (m-degrees-type counterpoint) is-cst-arr2)
271 (add-cost-to-factors (m-degrees-cost counterpoint) 'm-degrees-cost)
272 (gil::g-count *sp* (m-degrees-type counterpoint) 2 gil::IRT_LQ (floor (* (-
    1 (getparam 'min-skips-slider)) m-len)))
273 )
274 )
275
276 ; merge lists intermittently such that the first element of the first list is
    followed by the first element of the second list, etc.
277 ; attention: cp-len is lenght of the first list in cp-list and it should be 1 more
    than the lenght of the other lists
278 (defun merge-cp (cp-list total-cp)
279     (let (
280         (cp-len-1 (- (length (first cp-list)) 1))
281         (n-list (length cp-list))
282     )
283         (loop
284             for i from 0 below cp-len-1
285             do
286                 (loop for j from 0 below n-list do
287                     (setf (nth (+ (* i n-list) j) total-cp) (nth i (nth j cp-list)))
288                 )
289             )
290             (gil::g-rel *sp* (lastone total-cp) gil::IRT_EQ (lastone (first cp-list)))
291         )
292     )
293
294 ; merge lists intermittently such that the first element of the first list is
    followed by the first element of the second list, etc.
295 ; attention: lengths should be the same
296 (defun merge-cp-same-len (cp-list total-cp)
297     (let (
298         (cp-len (length (first cp-list)))
299         (n-list (length cp-list))
300     )
301         (loop

```

```

302     for i from 0 below cp-len
303     do
304         (loop for j from 0 below n-list do
305             (setf (nth (+ (* i n-list) j) total-cp) (nth i (nth j cp-list)))
306         )
307     )
308 )
309 )
310
311 ; create the harmonic intervals between @cp and @cf in @h-intervals
312 (defun create-h-intervals (cp cf h-intervals)
313     (loop
314         for p in cp
315         for q in cf
316         for i in h-intervals do
317             (inter-eq-cst *sp* p q i) ; add a constraint to *sp* such that  $i = |p - q| \% 12$ 
318     )
319 )
320
321 ; create the intervals between @line1 and @line2 in @intervals and @brut-intervals
322 (defun create-intervals (line1 line2 intervals brut-intervals)
323     (loop
324         for p in line1
325         for q in line2
326         for i in intervals
327         for ib in brut-intervals
328         do
329             (inter-eq-cst-brut *sp* q p ib i) ; add a constraint to *sp* such that
330                  $ib = p - q$  and  $i = |ib|$ 
331     )
332 )
333 ; create the intervals between @line1 and @line2 in @intervals and @brut-intervals
334 ; where @is-cst-arr is true
335 (defun create-intervals-for-cst (line1 line2 intervals brut-intervals is-cst-arr)
336     (loop
337         for p in line1
338         for q in line2
339         for i in intervals
340         for ib in brut-intervals
341         for is-cst in is-cst-arr
342         do
343             (inter-eq-cst-brut-for-cst *sp* q p ib i is-cst) ; add a constraint to *
344                 sp* such that  $ib = p - q$  and  $i = |ib|$ 
345     )
346 )
347
348 ; create the melodic intervals of @cp in @m-intervals and @m-intervals-brut
349 ; @is-cst-arr is a list of booleans indicating whether the melodic interval is
350 ; constrained or not
351 (defun create-m-intervals-self (cp m-intervals m-intervals-brut &optional (
352     is-cst-arr nil))
353     (if is-cst-arr
354         ; then
355         (create-intervals-for-cst (butlast cp) (rest cp) m-intervals
356             m-intervals-brut is-cst-arr)
357         ; else
358         (create-intervals (butlast cp) (rest cp) m-intervals m-intervals-brut)
359     )
360 )
361 )

```

```

357 ; create an array of IntVar with the melodic interval between each arsis and its
      following thesis
358 (defun create-m-intervals-next-meas (cp-arsis cp m-intervals-arsis
      m-intervals-arsis-brut)
359   (create-intervals cp-arsis (rest cp) m-intervals-arsis m-intervals-arsis-brut)
360 )
361
362 ; create the melodic intervals two positions apart of @cp in @m2-intervals and
      @m2-intervals-brut
363 (defun create-m2-intervals (cp m2-intervals m2-intervals-brut)
364   (create-intervals (butlast (butlast cp)) (rest (rest cp)) m2-intervals
      m2-intervals-brut)
365 )
366
367 ; create the melodic intervals between the thesis of @cp and the arsis of @cp-arsis
      in @m-intervals and @m-intervals-brut
368 (defun create-m-intervals-in-meas (cp cp-arsis ta-intervals ta-intervals-brut)
369   (create-intervals (butlast cp) cp-arsis ta-intervals ta-intervals-brut)
370 )
371
372 ; create the brut melodic intervals of @cf in @cf-brut-m-intervals
373 (defun create-cf-brut-m-intervals (cf cf-brut-m-intervals)
374   (loop
375     for p in (butlast cf)
376     for q in (rest cf)
377     for i in cf-brut-m-intervals do
378       (let (
379         (ib (inter q p t))
380         )
381         (gil::g-rel *sp* i gil::IRT_EQ ib)
382       )
383   )
384 )
385
386 ; create the boolean array @is-p-cons-arr indicating if the interval is a perfect
      consonance or not
387 (defun create-is-p-cons-arr (h-intervals is-p-cons-arr)
388   (loop
389     for i in h-intervals
390     for p in is-p-cons-arr
391     do
392       (let (
393         (b-7 (gil::add-bool-var *sp* 0 1))
394         (b-0 (gil::add-bool-var *sp* 0 1))
395         )
396         (gil::g-rel-reify *sp* i gil::IRT_EQ 7 b-7) ; b-7 = (i == 7) -> the
            interval is a fifth
397         (gil::g-rel-reify *sp* i gil::IRT_EQ 0 b-0) ; b-0 = (i == 0) -> the
            interval is an octave
398         (gil::g-op *sp* b-0 gil::BOT_OR b-7 p) ; p = b-7 || b-0
399       )
400   )
401 )
402
403 ;; Initialises the strata arrays, so that there is a bijection between each part (cf
      , cp1 and cp2) and each strata (lowest, middle, highest)
404 (defun create-strata-arrays (parts)
405   (setf cantus-firmus (first parts))
406   (setq sorted-voices (make-list *cf-len :initial-element nil))
407   (dotimes (i *N-PARTS) (setf (is-not-lowest (nth i parts)) (gil::
      add-bool-var-array *sp* *cf-len 0 1))) ; is-not-lowest represents if the
      part is not the lowest stratum
408   (dotimes (i *cf-len) ; the ith measure

```

```

409 (setf voices (gil::add-int-var-array *sp* *N-PARTS 0 120)) ; the notes to
    sort
410 (dotimes (j *N-PARTS) ; the jth counterpoint
411   (if (eq (species (nth j parts)) 4)
412     (if (< i *cf-last-index)
413       (gil::g-rel *sp* (nth j voices) gil::IRT_EQ (nth i (third (notes
        (nth j parts)))))) ; if fourth species consider the third
        beat
414       (gil::g-rel *sp* (nth j voices) gil::IRT_EQ (nth *
        cf-penult-index (first (notes (nth j parts)))))) ; else
        consider the first
415     )
416     (gil::g-rel *sp* (nth j voices) gil::IRT_EQ (nth i (first (notes (
        nth j parts)))))) ; for the last index consider the first index
        no matter what: the last note is always on the first beat
417   )
418 )
419 (setf order (gil::add-int-var-array *sp* *N-PARTS 0 (- *N-PARTS 1))) ;
    contains the order of the parts, from the lowest note to the highest
420
421 (setf (nth i sorted-voices) (gil::add-int-var-array *sp* *N-PARTS 0 120)) ;
    the sorted notes
422 (gil::g-sorted *sp* voices (nth i sorted-voices) order) ; sort the notes and
    register their order
423
424 (gil::g-rel *sp* (nth i (first (notes *lowest))) gil::IRT_EQ (first (nth i
    sorted-voices))) ; the lowest stratum is the first in the sorted
425 (dotimes (j *N-COUNTERPOINTS) ; the jth voice
426   (gil::g-rel *sp* (nth i (first (notes (nth j *upper)))) gil::IRT_EQ (nth
    (+ j 1) (nth i sorted-voices))) ; the upper strata are the
    following
427 )
428
429 (let (
430   (cf-is-lowest (gil::add-bool-var *sp* 0 1)) ; boolean representing
    whether the cf is the lowest stratum or not
431   (cp1-is-lowest (gil::add-bool-var *sp* 0 1)) ; boolean representing
    whether the cp1 is the lowest stratum or not
432   (cp2-is-lowest (gil::add-bool-var *sp* 0 1)) ; boolean representing
    whether the cp2 is the lowest stratum or not
433   (cp1-equals-bass (gil::add-bool-var *sp* 0 1)) ; boolean representing
    whether the cp1 EQUALS the lowest stratum or not (not the same, it
    can be the same value but be the middle stratum, since there is a
    bijection between the two concepts)
434 )
435
436 ; the following lines compute the bijection and set the is-not-lowest
    variable for each part
437 ; if two parts compete for being the lowest stratum there is a priority:
    first the cf, then the cp1, then the cp2
438 ; e.g. if both cf and cp1 equal the value of the lowest stratum, cf will
    BE the lowest stratum and cp1 will not
439 ; e.g. if both cp1 and cp2 equal the value of the lowest stratum, cp1
    will BE the lowest stratum and cp2 will not
440
441 ; if cf==lowest -> cf is lowest <-> if cf!=lowest -> cf is not lowest
442 (gil::g-rel-reify *sp* (nth i (first (notes *lowest))) gil::IRT_NQ (nth
    i (first (notes cantus-firmus))) (nth i (is-not-lowest cantus-firmus
    )))
443
444
445 ; if cp1==lowest AND cf!=lowest -> cp1 is lowest <-> to know if cp1 is
    not lowest we take the following truth table

```

```

446 ; cpl==lowest      cf is lowest      cp is notlow
447 ;   1              1              1
448 ;   1              0              0
449 ;   0              1              1
450 ;   0              0              1
451 ; which is an implication, so cpl==lowest -> cf-is-lowest =
      cpl-is-not-lowest
452 (if (eq (species (second parts)) 4)
453     (if (< i *cf-last-index)
454         (gil::g-rel-reify *sp* (nth i (first (notes *lowest))) gil::
            IRT_EQ (nth i (third (notes (second parts))))
            cpl-equals-bass)
455         (gil::g-rel-reify *sp* (nth i (first (notes *lowest))) gil::
            IRT_EQ (nth *cf-penult-index (first (notes (second parts))))
            cpl-equals-bass)
456     )
457     (gil::g-rel-reify *sp* (nth i (first (notes *lowest))) gil::IRT_EQ (
        nth i (first (notes (second parts)))) cpl-equals-bass)
458 )
459 (gil::g-op *sp* cpl-equals-bass gil::BOT_IMP cf-is-lowest (nth i (
    is-not-lowest (second parts))))
460
461 ; if both cf and cpl are not the lowest then cp2 is the lowest
462 (if (eq *N-COUNTERPOINTS 2) (gil::g-op *sp* (nth i (is-not-lowest
    cantus-firmus)) gil::BOT_XOR (nth i (is-not-lowest (second parts)))
    (nth i (is-not-lowest (third parts)))))
463
464 ; computing the "is-lowest" for each part
465 (gil::g-op *sp* cf-is-lowest gil::BOT_XOR (nth i (is-not-lowest
    cantus-firmus)) 1)
466 (gil::g-op *sp* cpl-is-lowest gil::BOT_XOR (nth i (is-not-lowest (second
    parts))) 1)
467 (if (eq *N-COUNTERPOINTS 2) (gil::g-op *sp* cp2-is-lowest gil::BOT_XOR (
    nth i (is-not-lowest (third parts))) 1))
468
469 (if (> i 0) (let
470     (
471         (corresponding-m-intervals (make-list *N-PARTS :initial-element
            nil)) ; the last melodic interval of each measure for each
            part
472     )
473     (dotimes (j *N-PARTS)
474         (case (species (nth j parts))
475             (0 (setf (nth j corresponding-m-intervals) (first (
                m-intervals-brut (nth j parts))))) ; last melodic
                interval is between the first beat of the measure and
                the next measure
476             (1 (setf (nth j corresponding-m-intervals) (first (
                m-intervals-brut (nth j parts))))) ; last melodic
                interval is between the first beat of the measure and
                the next measure
477             (2 (setf (nth j corresponding-m-intervals) (third (
                m-intervals-brut (nth j parts))))) ; last melodic
                interval is between the third beat of the measure and
                the next measure
478             (3 (setf (nth j corresponding-m-intervals) (fourth (
                m-intervals-brut (nth j parts))))) ; last melodic
                interval is between the fourth beat of the measure and
                the next measure
479             (4 (setf (nth j corresponding-m-intervals) (third (
                m-intervals-brut (nth j parts))))) ; last melodic
                interval is between the third beat of the measure and
                the next measure

```

```

480         (5 (setf (nth j corresponding-m-intervals) (third (
           m-intervals-brut (nth j parts)))))) ; last melodic
           interval is between the third beat of the measure and
           the next measure
481     )
482 )
483
484 ; setting the melodic interval of the corresponding part to be the
           melodic interval of the lowest stratum
485 (gil::g-rel-reify *sp* (nth (- i 1) (nth 0 corresponding-m-intervals
           )) gil::IRT_EQ (nth (- i 1) (first (m-intervals-brut *lowest)))
           cf-is-lowest)
486 (gil::g-rel-reify *sp* (nth (- i 1) (nth 1 corresponding-m-intervals
           )) gil::IRT_EQ (nth (- i 1) (first (m-intervals-brut *lowest)))
           cp1-is-lowest)
487 (if (eq *N-COUNTERPOINTS 2) (gil::g-rel-reify *sp* (nth (- i 1) (nth
           2 corresponding-m-intervals)) gil::IRT_EQ (nth (- i 1) (first (
           m-intervals-brut *lowest))) cp2-is-lowest))
488 ))
489 )
490 )
491 )
492
493
494 ; create the boolean array @is-cf-lower-arr indicating if the cantus firmus is the
           bass or not
495 (defun create-is-cf-lower-arr (cp cf is-cf-lower-arr)
496   (loop
497     for p in cp
498     for q in cf
499     for b in is-cf-lower-arr
500     do
501       (gil::g-rel-reify *sp* p gil::IRT_GQ q b) ; b = (p >= q)
502   )
503 )
504
505 ; create an array of BoolVar such that is-ta-dim-arr is true if the note is a
           diminution:
506 ; 1 -> inter(thesis, arsis) == 1 or 2 && inter(thesis, thesis + 1) == 3 or 4 &&
           inter(arsis, thesis + 1) == 1 or 2
507 ; @m-intervals-ta: the melodic interval between each thesis and its following arsis
508 ; @m-intervals: the melodic interval between each thesis and its following thesis
509 ; @m-intervals-arsis: the melodic interval between each arsis and its following
           thesis
510 ; @is-ta-dim-arr: the array of BoolVar to fill
511 (defun create-is-ta-dim-arr (m-intervals-ta m-intervals m-intervals-arsis
           is-ta-dim-arr)
512   (loop
513     for mta in m-intervals-ta ; inter(thesis, arsis)
514     for mtt in m-intervals ; inter(thesis, thesis + 1)
515     for mat in m-intervals-arsis ; inter(arsis, thesis + 1)
516     for b in is-ta-dim-arr ; the BoolVar to create
517     do
518       (let (
519         (btt3 (gil::add-bool-var *sp* 0 1)) ; for mtt == 3
520         (btt4 (gil::add-bool-var *sp* 0 1)) ; for mtt == 4
521         (bta-second (gil::add-bool-var *sp* 0 1)) ; for mat <= 2
522         (btt-third (gil::add-bool-var *sp* 0 1)) ; for mtt == 3 or 4
523         (bat-second (gil::add-bool-var *sp* 0 1)) ; for mta <= 2
524         (b-and (gil::add-bool-var *sp* 0 1)) ; temporary BoolVar
525       )
526         (gil::g-rel-reify *sp* mtt gil::IRT_EQ 3 btt3) ; btt3 = (mtt == 3)
527         (gil::g-rel-reify *sp* mtt gil::IRT_EQ 4 btt4) ; btt4 = (mtt == 4)

```

```

528         (gil::g-rel-reify *sp* mta gil::IRT_LQ 2 bta-second) ; bta2 = (mta
          <= 2)
529         (gil::g-rel-reify *sp* mat gil::IRT_LQ 2 bat-second) ; bat1 = (mat
          <= 2)
530         (gil::g-op *sp* btt3 gil::BOT_OR btt4 btt-third) ; btt-third = btt3
          || btt4
531         (gil::g-op *sp* bta-second gil::BOT_AND btt-third b-and) ; temporay
          operation
532         (gil::g-op *sp* b-and gil::BOT_AND bat-second b) ; b = bta-second &&
          btt-third && bat-second
533     )
534 )
535 )
536
537 ; create an array of BoolVar
538 ; 1 -> inter(cp, cf) <= 4 && cf getting closer to cp
539 (defun create-is-nbour-arr (h-intervals-abs is-cf-lower-arr cf-brut-m-intervals
    is-nbour-arr)
540     (loop
541         for hi in (butlast h-intervals-abs)
542         for bass in (butlast is-cf-lower-arr)
543         for mi in cf-brut-m-intervals
544         for n in is-nbour-arr
545         do
546             (let (
547                 (b-hi (gil::add-bool-var *sp* 0 1)) ; for (hi <= 4)
548                 (b-cfu (gil::add-bool-var *sp* 0 1)) ; for cf going up
549                 (b-cfgc (gil::add-bool-var *sp* 0 1)) ; for cf getting closer to cp
550             )
551                 (gil::g-rel-reify *sp* hi gil::IRT_LQ 4 b-hi) ; b-hi = (hi <= 4)
552                 (gil::g-rel-reify *sp* mi gil::IRT_GQ 0 b-cfu) ; b-cfu = (mi >= 0)
553                 (gil::g-op *sp* bass gil::BOT_EQV b-cfu b-cfgc) ; b-cfgc = (bass ==
                    b-cfu)
554                 (gil::g-op *sp* b-hi gil::BOT_AND b-cfgc n) ; n = b-hi && b-cfgc
555             )
556         )
557 )
558
559 ; TODO: new version below should be used instead of this one
560 ; create an array of BoolVar
561 ; 1 -> 5 quarter notes strictly ups or downs and are linked by joint degrees
562 ; Note: the rule is applied measure by measure
563 (defun create-is-5qn-linked-arr (m-all-intervals m-all-intervals-brut
    is-5qn-linked-arr)
564     (loop
565         for i from 0 to (- (length m-all-intervals) 3)
566         for m1 in m-all-intervals
567         for m2 in (rest m-all-intervals)
568         for m3 in (rest (rest m-all-intervals))
569         for m4 in (rest (rest (rest m-all-intervals)))
570         for mb1 in m-all-intervals-brut
571         for mb2 in (rest m-all-intervals-brut)
572         for mb3 in (rest (rest m-all-intervals-brut))
573         for mb4 in (rest (rest (rest m-all-intervals-brut)))
574         for b in is-5qn-linked-arr
575         do
576             (if (eq (mod i 4) 0)
577                 ; then
578                 (let (
579                     (b1 (gil::add-bool-var *sp* 0 1)) ; (m1 <= 2)
580                     (b2 (gil::add-bool-var *sp* 0 1)) ; (m2 <= 2)
581                     (b3 (gil::add-bool-var *sp* 0 1)) ; (m3 <= 2)
582                     (b4 (gil::add-bool-var *sp* 0 1)) ; (m4 <= 2)

```

```

583      (bb1 (gil::add-bool-var *sp* 0 1)) ; (mb1 > 0)
584      (bb2 (gil::add-bool-var *sp* 0 1)) ; (mb2 > 0)
585      (bb3 (gil::add-bool-var *sp* 0 1)) ; (mb3 > 0)
586      (bb4 (gil::add-bool-var *sp* 0 1)) ; (mb4 > 0)
587      (b-and1 (gil::add-bool-var *sp* 0 1)) ; (b1 && b2)
588      (b-and2 (gil::add-bool-var *sp* 0 1)) ; (b3 && b4)
589      (b-and3 (gil::add-bool-var *sp* 0 1)) ; (b-and1 && b-and2)
590      (b-eq1 (gil::add-bool-var *sp* 0 1)) ; (mb1 == mb2)
591      (b-eq2 (gil::add-bool-var *sp* 0 1)) ; (mb3 == mb3)
592      (b-eq3 (gil::add-bool-var *sp* 0 1)) ; (b-eq1 == b-eq2)
593    )
594    (gil::g-rel-reify *sp* m1 gil::IRT_LQ 2 b1) ; b1 = (m1 <= 2)
595    (gil::g-rel-reify *sp* m2 gil::IRT_LQ 2 b2) ; b2 = (m2 <= 2)
596    (gil::g-rel-reify *sp* m3 gil::IRT_LQ 2 b3) ; b3 = (m3 <= 2)
597    (gil::g-rel-reify *sp* m4 gil::IRT_LQ 2 b4) ; b4 = (m4 <= 2)
598    (gil::g-rel-reify *sp* mb1 gil::IRT_GQ 0 bb1) ; bb1 = (mb1 > 0)
599    (gil::g-rel-reify *sp* mb2 gil::IRT_GQ 0 bb2) ; bb2 = (mb2 > 0)
600    (gil::g-rel-reify *sp* mb3 gil::IRT_GQ 0 bb3) ; bb3 = (mb3 > 0)
601    (gil::g-rel-reify *sp* mb4 gil::IRT_GQ 0 bb4) ; bb4 = (mb4 > 0)
602    (gil::g-op *sp* b1 gil::BOT_AND b2 b-and1) ; b-and1 = b1 && b2
603    (gil::g-op *sp* b3 gil::BOT_AND b4 b-and2) ; b-and2 = b3 && b4
604    (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 b-and3) ; b-and3 = b-and1
        && b-and2
605    (gil::g-op *sp* bb1 gil::BOT_EQV bb2 b-eq1) ; b-eq1 = (bb1 == bb2)
606    (gil::g-op *sp* bb3 gil::BOT_EQV bb4 b-eq2) ; b-eq2 = (bb3 == bb4)
607    (gil::g-op *sp* b-eq1 gil::BOT_EQV b-eq2 b-eq3) ; b-eq3 = (b-eq1 ==
        b-eq2)
608    (gil::g-op *sp* b-and3 gil::BOT_AND b-eq3 b) ; b = b-and3 && b-eq3
609  )
610 )
611 )
612 )
613
614 ; create an array of BoolVar representing if the second note is not cambiata
615 (defun create-is-not-cambiata-arr (is-cons-arr2 is-cons-arr3 m-intervals
    is-not-cambiata-arr)
616   (loop
617     for b2 in is-cons-arr2
618     for b3 in is-cons-arr3
619     for m in m-intervals
620     for b in is-not-cambiata-arr
621     do
622       (let (
623         (b-m (gil::add-bool-var *sp* 0 1)) ; (m <= 2)
624         (b-and (gil::add-bool-var *sp* 0 1)) ; (b2 && b3)
625       )
626         (gil::g-op *sp* b2 gil::BOT_AND b3 b-and) ; b-and = b2 && b3
627         (gil::g-rel-reify *sp* m gil::IRT_LQ 2 b-m) ; b-m = (m <= 2)
628         (gil::g-op *sp* b-and gil::BOT_AND b-m b) ; b = b-and && b-m
629       )
630     )
631 )
632
633 ; create an array of BoolVar representing if there is no syncopation
634 (defun create-is-no-syncopation-arr (m-intervals is-no-syncopation-arr)
635   (loop
636     for m in (butlast m-intervals)
637     for b in is-no-syncopation-arr
638     do
639       (gil::g-rel-reify *sp* m gil::IRT_NQ 0 b)
640     )
641 )
642

```

```

643 ; add constraints such that @b-member is true iff @candidate is a member of
    @member-list
644 (defun add-is-member-cst (candidate member-list b-member)
645   (let (
646     (results (gil::add-int-var-array *sp* (length member-list) 0 1)) ; where
        candidate == m
647     (sum (gil::add-int-var *sp* 0 (length member-list))) ; sum(results)
648   )
649     (loop
650       for m in member-list
651       for r in results
652       do
653         (let (
654           (b1 (gil::add-bool-var *sp* 0 1)) ; b1 = (candidate == m)
655         )
656           (gil::g-rel-reify *sp* candidate gil::IRT_EQ m b1) ; b1 = (candidate
            == m)
657           (gil::g-ite *sp* b1 ONE ZERO r) ; r = (b1 ? 1 : 0)
658         )
659       )
660     (gil::g-sum *sp* sum results) ; sum = sum(results)
661     (gil::g-rel-reify *sp* sum gil::IRT_GR 0 b-member) ; b-member = (sum >= 1)
662   )
663 )
664
665 ; create an array of BoolVar
666 ; 1 -> the harmonic interval is member of the set (consonances set by default)
667 (defun create-is-member-arr (h-intervals cons-arr &optional (cons-set ALL_CONS))
668   (loop
669     for h in h-intervals
670     for b in cons-arr
671     do
672       (add-is-member-cst h cons-set b)
673   )
674 )
675
676 ; add the constraint such that the harmonies in @h-intervals are consonances expect
    the penultimate note (specific rule). the fourth species also follows specific
    rules
677 ; @len: the length of the counterpoint
678 ; @cf-penult-index: the index of penultimate note in the counterpoint
679 ; @h-intervals: the array of harmonic intervals
680 ; @penult-dom-var: the domain of the penultimate note
681 ; @species: the species of the counterpoint
682 ; @is-not-lowest: boolean array to know whether the counterpoint is the lowest
    stratum
683 (defun add-h-cons-cst (len cf-penult-index h-intervals &optional (penult-dom-var
    PENULT_CONS_VAR) (species 0) (is-not-lowest nil))
684   (loop for i from 0 below len do
685     (if (/= species 4)
686       ; if not 4th species (normal case)
687       (if (eq i *cf-last-index) ; if it is the last note
688         ; then add only harmonic triad options
689         (gil::g-member *sp* MAJ_H_TRIAD_VAR (nth i h-intervals))
690         (if (eq i *cf-penult-index) ; if penult note
691           ; add penult options
692           (gil::g-member *sp* penult-dom-var (nth i h-intervals))
693           ; else add all consonances
694           (gil::g-member *sp* ALL_CONS_VAR (nth i h-intervals))
695         )
696       )
697     ; if 4th species (if the lowest stratum doesn't move then dissonance,
        else consonance)

```

```

698         (case i
699           (0 (gil::g-member *sp* ALL_CONS_VAR (nth i h-intervals))) ; first
              measure
700           (*cf-penult-index (gil::g-member *sp* penult-dom-var (nth i
              h-intervals))) ; penult measure
701           (*cf-last-index (gil::g-member *sp* MAJ_H_TRIAD_VAR (nth i
              h-intervals))) ; last measure
702           (otherwise (let
703             (
704               (lower-stays (gil::add-bool-var *sp* 0 1)) ; if the lowest
              stratum doesn't move
705               (is-not-lowest-and-lower-stays (gil::add-bool-var *sp* 0 1))
              ; if the ctp is not the lowest and the lowest doesn't
              move
706               (lower-not-stays (gil::add-bool-var *sp* 0 1)) ; if the
              lowest stratum moves
707               (is-not-lowest-and-lower-not-stays (gil::add-bool-var *sp* 0
              1)) ; if the ctp is not the lowest and the lowest moves
708               (h-dis (gil::add-int-var *sp* 0 1)) ; temp
709               (h-cons (gil::add-int-var *sp* 0 1)) ; temp
710             )
711             (gil::g-rel-reify *sp* (nth (- i 1) (first (m-intervals-brut *
              lowest))) gil::IRT_EQ 0 lower-stays) ; lower-stays := (
              m-intervals lowest = 0)
712             (gil::g-op *sp* lower-stays gil::BOT_AND (nth i is-not-lowest)
              is-not-lowest-and-lower-stays) ;
              is-not-lowest-and-lower-stays := is-not-lowest AND
              lowest-stays
713             (gil::g-rel-reify *sp* (nth (- i 1) (first (m-intervals-brut *
              lowest))) gil::IRT_NQ 0 lower-not-stays) ; lower-stays := (
              m-intervals lowest = 0)
714             (gil::g-op *sp* lower-not-stays gil::BOT_AND (nth i
              is-not-lowest) is-not-lowest-and-lower-not-stays) ;
              is-not-lowest-and-lower-stays := is-not-lowest AND
              lowest-not-stays
715
716             (gil::g-member *sp* DIS_VAR h-dis) ; temporary is member of DIS
717             (gil::g-member *sp* ALL_CONS_VAR h-cons) ; temporary is member
              of CONS
718             (gil::g-rel-reify *sp* h-dis gil::IRT_EQ (nth i h-intervals)
              is-not-lowest-and-lower-stays) ;
              is-not-lowest-and-lower-stays <=> h-interval is member of
              DIS
719             (gil::g-rel-reify *sp* h-cons gil::IRT_EQ (nth i h-intervals)
              is-not-lowest-and-lower-not-stays) ;
              is-not-lowest-and-lower-not-stays <=> h-interval is member
              of CONS
720           ))
721         )
722       )
723     )
724   )
725
726
727   ; add the constraint such that the penultimate note belongs to the domain
       @penult-dom-var
728   (defun add-penult-dom-cst (h-interval penult-dom-var)
729     (if (getparam 'penult-rule-check)
730       (gil::g-member *sp* penult-dom-var h-interval)
731     )
732   )
733
734

```

```

735 ; add the constraint such that is-cst-arr[i] => is-cons-arr[i] is true
736 ; -is-cons-arr: array of BoolVar, 1 -> the harmonic interval is a consonance
737 ; -is-cst-arr: array of BoolVar, 1 -> the note is constrained by a species
738 (defun add-h-cons-cst-if (is-cons-arr is-cst-arr)
739   (loop
740     for is-cons in is-cons-arr
741     for is-cst in is-cst-arr
742     do
743       (gil::g-op *sp* is-cst gil::BOT_IMP is-cons 1) ; (is-cst => is-cons) = 1
744   )
745 )
746
747 ; add the constraint such that h-intervals[i] belongs to ALL_CONS_VAR
748 ; is-no-syncop-arr[i] is true
749 ; in other words, if there is no syncopation the note cannot be dissonant
749 (defun add-no-sync-h-cons (h-intervals is-no-syncop-arr)
750   (loop
751     for h in h-intervals
752     for b in is-no-syncop-arr
753     do
754       (loop for d in DIS do
755         (gil::g-rel-reify *sp* h gil::IRT_NQ d b gil::RM_IMP) ; b => (h != d)
756       )
757     )
758   )
759
760 ; for future work: should use not(nth i is-cons-arr) instead of add a constraint for
761 ; each dissonance in DIS
762 ; -len: length of the harmonic array
763 ; -cf-penult-index: index of the penultimate note in the counterpoint
764 ; -h-intervals-arsis: harmonic intervals of the arsis of the counterpoint
765 ; -is-ta-dim-arr: array of BoolVar, 1 -> the note in arsis is a diminution
766 ; -penult-dom-var: domain of the penultimate note
766 (defun add-h-cons-arsis-cst (len cf-penult-index h-intervals-arsis is-ta-dim-arr &
767   optional (penult-dom-var PENULT_CONS_VAR))
768   (loop
769     for i from 0 below len
770     for b in is-ta-dim-arr
771     do
772       (if (eq i cf-penult-index) ; if it is the penultimate note
773         ; then add major sixth + minor third
774         (add-penult-dom-cst (nth i h-intervals-arsis) penult-dom-var)
775         ; else dissonance implies there is a diminution
776         (loop for d in DIS do
777           (gil::g-rel-reify *sp* (nth i h-intervals-arsis) gil::IRT_EQ d b gil::RM_PMI)
778         )
779       )
780     )
781   )
782
783 ; add the constraint such that (c3 OR (c2 AND c4)) AND (c3 OR dim) is true,
784 ; where : - cn represents if the nth note of the measure is consonant
785 ; - dim represents if the 3rd note is a diminution
785 (defun add-h-dis-or-cons-3rd-cst (is-cons-2nd is-cons-3rd is-cons-4th is-dim &
786   optional (is-cst-arr nil))
787   (loop
788     for b-c2nd in is-cons-2nd
789     for b-c3rd in is-cons-3rd
790     for b-c4th in is-cons-4th
791     for b-dim in is-dim
792     do
793       (let (

```

```

793      (b-and1 (gil::add-bool-var *sp* 0 1)) ; s.f. b-c2nd AND b-c4th
794    )
795    (gil::g-op *sp* b-c2nd gil::BOT_AND b-c4th b-and1) ; b-and1 = b-c2nd AND
      b-c4th
796    (gil::g-op *sp* b-c3rd gil::BOT_OR b-dim 1) ; b-and2 = b-c2nd AND b-c4th
      AND b-dim
797  )
798 )
799 )
800
801 ; add constraints such that
802 ; any dissonant note implies that it is followed by the next consonant note below
803 ; @m-succ-intervals-brut: list of IntVar, s.f. brut melodic intervals between thesis
      and arsis
804 ; @is-cons-arr: list of BoolVar, s.f. 1 -> the note is consonant
805 ; @is-cst-arr: list of BoolVar, s.f. 1 -> the note is constrained by a species
806 (defun add-h-dis-imp-cons-below-cst (m-succ-intervals-brut is-cons-arr &optional (
      is-cst-arr nil))
807   (loop
808     for m in m-succ-intervals-brut
809     for b in is-cons-arr
810     for i from 0 below (length m-succ-intervals-brut)
811     do
812       (let (
813         (b-not (gil::add-bool-var *sp* 0 1)) ; s.f. !b (dissonance)
814         (is-cst (true-if-null is-cst-arr i)) ; s.f. is-cst = 1 -> the note is
          constrained by a species
815         (b-and (gil::add-bool-var *sp* 0 1)) ; s.f. b-not && is-cst
816       )
817         (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b (dissonance)
818         (gil::g-op *sp* b-not gil::BOT_AND is-cst b-and) ; b-and = b-not &&
          is-cst
819         (gil::g-rel-reify *sp* m gil::IRT_LE 0 b-and gil::RM_IMP) ; b-and => m <
          0
820         (gil::g-rel-reify *sp* m gil::IRT_GQ -2 b-and gil::RM_IMP) ; b-and => m
          >= -2
821       )
822     )
823 )
824
825 ; add constraints such that if a melodic interval is greater than one step (2)
826 ; then the next melodic interval should be one step and in the opposite direction
827 (defun add-contrary-step-after-skip-cst (m-all-intervals m-all-intervals-brut)
828   (if (not (getparam 'con-m-after-skip-check))
829     (return-from add-contrary-step-after-skip-cst)
830   )
831   (loop
832     for m in m-all-intervals
833     for m+1 in (rest m-all-intervals)
834     for mb in m-all-intervals-brut
835     for mb+1 in (rest m-all-intervals-brut)
836     do
837       (let (
838         (b-skip (gil::add-bool-var *sp* 0 1)) ; m > 2
839         (b-mb-up (gil::add-bool-var *sp* 0 1)) ; mb > 0
840         (b-mb+1-down (gil::add-bool-var *sp* 0 1)) ; mb+1 < 0
841         (b-contrary (gil::add-bool-var *sp* 0 1)) ; b-mb-up <=> b-mb+1-down
842       )
843         (gil::g-rel-reify *sp* m gil::IRT_GR 2 b-skip) ; b-skip := m > 2
844         (gil::g-rel-reify *sp* mb gil::IRT_GR 0 b-mb-up) ; b-mb-up := mb > 0
845         (gil::g-rel-reify *sp* mb+1 gil::IRT_LE 0 b-mb+1-down) ; b-mb+1-down :=
          mb+1 < 0

```

```

846         (gil::g-op *sp* b-mb-up gil::BOT_EQV b-mb+1-down b-contrary) ;
            b-contrary := b-mb-up <=> b-mb+1-down
847         (gil::g-rel-reify *sp* m+1 gil::IRT_LQ 2 b-skip gil::RM_IMP) ; b-skip =>
            m+1 <= 2
848         (gil::g-op *sp* b-skip gil::BOT_IMP b-contrary 1) ; b-skip => b-contrary
849     )
850 )
851 )
852
853 ; is-5qn-linked-arr implies that is-cons-arr1 (supposed to always be true) and
    is-cons-arr3 are true
854 (defun add-linked-5qn-cst (is-cons-arr3 is-5qn-linked-arr)
855     (loop
856         ; for b1 in is-cons-arr1
857         for b3 in is-cons-arr3
858         for b in is-5qn-linked-arr
859         do
860             (gil::g-op *sp* b gil::BOT_IMP b3 1) ; b => b3
861         )
862     )
863
864 ; add the constraint such that there cp is never equal to cf
865 (defun add-no-unison-at-all-cst (cp cf &optional (is-cst-arr nil))
866     (loop
867         for p in cp
868         for q in cf
869         for i from 0 below (length cp)
870         do
871             (if (and p q)
872                 (rel-reify-if p gil::IRT_NQ q (nth i is-cst-arr))
873             )
874         )
875     )
876
877 ; add the constraint such that there is no unison unless it is the first or last
    note
878 (defun add-no-unison-cst (cp cf)
879     (add-no-unison-at-all-cst (restbutlast cp) (restbutlast cf))
880 )
881
882
883 ; add the constraint that the three voices go in different directions
884 ; i.e. that there are no two direct motions
885 ; i.e. that there can be only one part moving in direct motion (since one part has
    motion=-1 (bc it is the lowest stratum), and if the two other parts have motion=
    direct then all voices go in the same direction)
886 ; WARNING: this function needs to be scaled before implementing a fourth voice, it
    currently works by restricting the number of direct motions to max. 1
887 (defun add-no-together-move-cst (motions)
888     (loop
889         ; for each possible pair or motions
890         ; for example if we have (m1, m2 and m3), take (m1 and m2), (m1 and m3) and
            (m2 and m3)
891         for motions1 in motions
892         for i from 0
893         do (loop for motions2 in (nthcdr (1+ i) motions)
894             do
895                 (loop for m1 in motions1 for m2 in motions2 do
896                     (let (
897                         (m1-direct (gil::add-bool-var *sp* 0 1))
898                         (m2-direct (gil::add-bool-var *sp* 0 1))
899

```

```

900         (gil::g-rel-reify *sp* m1 gil::IRT_EQ 2 m1-direct) ; m1-direct := (
           motion1 == 2)
901         (gil::g-rel-reify *sp* m2 gil::IRT_EQ 2 m2-direct) ; m2-direct := (
           motion2 == 2)
902         (gil::g-op *sp* m1-direct gil::BOT_AND m2-direct 0) ; NOT (m1-direct AND
           m2-direct) (not both at the same time)
903     ))
904 ))
905 )
906 ; add the constraint such that the first harmonic interval is a perfect consonance
907 (defun add-p-cons-start-cst (h-intervals)
908     (gil::g-member *sp* P_CONS_VAR (first h-intervals))
909 )
910
911 ; add the constraint such that the last harmonic interval is a perfect consonance
912 (defun add-p-cons-end-cst (h-intervals)
913     (gil::g-member *sp* P_CONS_VAR (lastone h-intervals))
914 )
915
916 ; add the constraint that there cannot be a minor third in the last chord
917 (defun add-no-minor-third-cst (h-interval)
918     (gil::g-rel *sp* h-interval gil::IRT_NQ 3)
919 )
920
921 ; add the constraint that there cannot be a tenth in the last chord
922 (defun add-no-tenth-in-last-chord-cst (h-intervals h-intervals-brut)
923     (let (
924         (h (lastone h-intervals))
925         (hbrut (lastone h-intervals-brut))
926         (is-hbrut-not-third (gil::add-bool-var *sp* 0 1))
927     )
928         (gil::g-rel-reify *sp* hbrut gil::IRT_NQ 4
929         is-hbrut-not-third) ; if the hbrut is not a third
930         (gil::g-rel-reify *sp* h gil::IRT_NQ 4 is-hbrut-not-third) ; then there can
           be no third (as it would mean that the third would be a tenth)
931
932         ; There is no need to do the same for 3 (minor third) as minor thirds are
           prohibited altogether in the last chord
933     )
934 )
935
936 ; add the constraint that the chord shall be a harmonic triad ((1-3-5) or (1-5-8) or
   (1-3-8))
937 (defun add-last-chord-h-triad-cst (h-intervals-1 h-intervals-2)
938     (let (
939         (h-triad (gil::add-int-var-const-array *sp* (list 0 3 4 7)))
940     )
941         (gil::g-member *sp* h-triad (lastone h-intervals-1))
942         (gil::g-member *sp* h-triad (lastone h-intervals-2))
943     )
944 )
945
946 ; computes the harmonic triad cost
947 ; for each chord not being a harmonic triad, cost = *h-triad-cost*
948 (defun compute-h-triad-cost (h-intervals-1 h-intervals-2 costs)
949     (loop
950         for h1 in h-intervals-1
951         for h2 in h-intervals-2
952         for c in costs
953         do
954             (let (
955                 (is-h1-3 (gil::add-bool-var *sp* 0 1))
956                 (is-h1-4 (gil::add-bool-var *sp* 0 1))

```

```

957 (is-h1-third (gil::add-bool-var *sp* 0 1))
958 (is-h1-7 (gil::add-bool-var *sp* 0 1))
959 (is-h2-3 (gil::add-bool-var *sp* 0 1))
960 (is-h2-4 (gil::add-bool-var *sp* 0 1))
961 (is-h2-third (gil::add-bool-var *sp* 0 1))
962 (is-h2-7 (gil::add-bool-var *sp* 0 1))
963 (is-harmonic-triad-1st-possibility (gil::add-bool-var *sp* 0 1))
964 (is-harmonic-triad-2nd-possibility (gil::add-bool-var *sp* 0 1))
965 (is-harmonic-triad (gil::add-bool-var *sp* 0 1))
966 (is-not-h-triad (gil::add-bool-var *sp* 0 1))
967 )
968 (gil::g-rel-reify *sp* h1 gil::IRT_EQ 3 is-h1-3)
969 (gil::g-rel-reify *sp* h1 gil::IRT_EQ 4 is-h1-4)
970 (gil::g-rel-reify *sp* h2 gil::IRT_EQ 7 is-h2-7)
971 (gil::g-op *sp* is-h1-3 gil::BOT_OR is-h1-4 is-h1-third)
972 (gil::g-op *sp* is-h1-third gil::BOT_AND is-h2-7
    is-harmonic-triad-1st-possibility)
973
974 (gil::g-rel-reify *sp* h2 gil::IRT_EQ 3 is-h2-3)
975 (gil::g-rel-reify *sp* h2 gil::IRT_EQ 4 is-h2-4)
976 (gil::g-rel-reify *sp* h1 gil::IRT_EQ 7 is-h1-7) ;
977 (gil::g-op *sp* is-h2-3 gil::BOT_OR is-h2-4 is-h2-third)
978 (gil::g-op *sp* is-h2-third gil::BOT_AND is-h1-7
    is-harmonic-triad-2nd-possibility)
979
980 (gil::g-op *sp* is-harmonic-triad-1st-possibility gil::BOT_OR
    is-harmonic-triad-1st-possibility is-harmonic-triad)
981
982 (gil::g-op *sp* is-harmonic-triad gil::BOT_XOR is-not-h-triad 1) ;
    is-harmonic-triad = NOT is-not-h-triad
983 (gil::g-rel-reify *sp* c gil::IRT_EQ 0 is-harmonic-triad gil::RM_IMP) ;
    it costs 0 to be a harmonic triad
984 (gil::g-rel-reify *sp* c gil::IRT_EQ *h-triad-cost* is-not-h-triad gil::
    RM_IMP) ; it costs *h-triad-cost* not to be a harmonic triad
985 )
986 )
987 )
988
989 ; computes the harmonic triad cost for the 3rd species, i.e. 2nd and 3rd beat
990 ; for each chord not being a harmonic triad, cost = *h-triad-cost*
991 (defun compute-h-triad-3rd-species-cost (h-intervals costs)
992   (loop
993     for h-interval in h-intervals
994     for cost in costs
995     do
996       (let (
997         (not-minor-third (gil::add-bool-var *sp* 0 1))
998         (not-major-third (gil::add-bool-var *sp* 0 1))
999         (not-third (gil::add-bool-var *sp* 0 1))
1000         (not-major-fifth (gil::add-bool-var *sp* 0 1))
1001         (not-in-h-triad (gil::add-bool-var *sp* 0 1))
1002       )
1003         (gil::g-rel-reify *sp* h-interval gil::IRT_NQ 3 not-minor-third)
1004         (gil::g-rel-reify *sp* h-interval gil::IRT_NQ 4 not-major-third)
1005         (gil::g-rel-reify *sp* h-interval gil::IRT_NQ 7 not-major-fifth)
1006         (gil::g-op *sp* not-minor-third gil::BOT_AND not-major-third not-third)
1007         (gil::g-op *sp* not-third gil::BOT_AND not-major-fifth not-in-h-triad)
1008         (gil::g-rel-reify *sp* cost gil::IRT_EQ *h-triad-3rd-species-cost*
            not-in-h-triad)
1009       )
1010     )
1011 )
1012

```

```

1013 ; add the constraint such that the first and last harmonic interval are 0 if cp is
      at the bass
1014 ; not(is-cf-bass[0, 0]) => h-interval[0, 0] = 0
1015 ; not(is-cf-bass[-1, -1]) => h-interval[-1, -1] = 0
1016 ; @h-interval: the harmonic interval array
1017 ; @is-cf-lower-arr: boolean variables indicating if cf is lower than the given ctp
1018 (defun add-tonic-tuned-cst (h-interval is-cf-lower-arr)
1019   (let (
1020     (bf-not (gil::add-bool-var *sp* 0 1)) ; for !(first is-cf-lower-arr)
1021     (bl-not (gil::add-bool-var *sp* 0 1)) ; for !(lastone is-cf-lower-arr)
1022   )
1023     (gil::g-op *sp* (first is-cf-lower-arr) gil::BOT_EQV FALSE bf-not) ; bf-not
      = !(first is-cf-lower-arr)
1024     (gil::g-op *sp* (lastone is-cf-lower-arr) gil::BOT_EQV FALSE bl-not) ;
      bl-not = !(lastone is-cf-lower-arr)
1025     (gil::g-rel-reify *sp* (first h-interval) gil::IRT_EQ 0 bf-not gil::RM_IMP)
      ; bf-not => h-interval[0, 0] = 0
1026     (gil::g-rel-reify *sp* (lastone h-interval) gil::IRT_EQ 0 bl-not gil::RM_IMP)
      ; bl-not => h-interval[-1, -1] = 0
1027   )
1028 )
1029
1030 ; add the constraint such that the harmonic interval is a perfect consonance if it
      is constrained by a species
1031 (defun add-p-cons-cst-if (h-inter is-cst)
1032   (let (
1033     (b-fifth (gil::add-bool-var *sp* 0 1)) ; b-fifth = h-inter is a fifth
1034     (b-octave (gil::add-bool-var *sp* 0 1)) ; b-octave = h-inter is an octave
1035     (b-p-cons (gil::add-bool-var *sp* 0 1)) ; b-p-cons = h-inter is a perfect
      consonance
1036   )
1037     (gil::g-rel-reify *sp* h-inter gil::IRT_EQ 7 b-fifth) ; b-fifth = h-inter is
      a fifth
1038     (gil::g-rel-reify *sp* h-inter gil::IRT_EQ 0 b-octave) ; b-octave = h-inter
      is an octave
1039     (gil::g-op *sp* b-fifth gil::BOT_OR b-octave b-p-cons) ; b-p-cons = b-fifth
      or b-octave
1040     (gil::g-op *sp* is-cst gil::BOT_IMP b-p-cons 1) ; is-cst => b-p-cons
1041   )
1042 )
1043
1044 ; adds the constraint that if the cf is above the ctp, the interval must be a third,
      and if below then a sixth (to the cantus firmus)
1045 (defun add-penult-cons-lsp-and-cf-cst (is-not-lowest h-interval species)
1046   (case species
1047     (0 (if (getparam 'penult-rule-check) ; if the cantus firmus is not the
          lowest use a minor third
1048         (gil::g-rel-reify *sp* h-interval gil::IRT_EQ THREE is-not-lowest
          gil::RM_IMP)
1049       ))
1050     (1 (if (getparam 'penult-rule-check) ; if the cantus firmus is the lowest
          use a major sixth
1051         (gil::g-rel-reify *sp* h-interval gil::IRT_EQ NINE is-not-lowest gil
          ::RM_IMP)
1052       ))
1053   )
1054 )
1055
1056 ; adds the constraint that if the cf is above the ctp, the interval must be a third,
      and if below then a sixth (to the lowest stratum)
1057 (defun add-penult-cons-cst (b-bass h-interval &optional (and-cond nil))
1058   (if (getparam 'penult-rule-check)
1059     (gil::g-rel-reify *sp* h-interval gil::IRT_EQ THREE is-not-lowest gil
      ::RM_IMP)
1060     (gil::g-rel-reify *sp* h-interval gil::IRT_EQ SIX is-not-lowest gil
      ::RM_IMP)
1061   )
1062 )

```

```

1060     (if (null and-cond)
1061         (gil::g-ite *sp* b-bass NINE THREE h-interval)
1062         (and-ite b-bass NINE THREE h-interval and-cond)
1063     )
1064 )
1065 )
1066
1067 ; adds a constraint so that the last bass notes is the fundamental note of the key
1068 (defun last-lowest-note-same-as-root-note-cst ()
1069     (let (
1070         (TWELVE (gil::add-int-var-dom *sp* '(12))) ; the IntVar just used to store
1071             12
1072         (CF-MODULO (gil::add-int-var-dom *sp* (list (mod (first *cf) 12)))) ; the
1073             value of the first note of the cf modulo 12
1074     )
1075     (gil::g-mod *sp* (lastone (first (notes *lowest))) TWELVE CF-MODULO) ; the
1076         last note of the lowest stratum % 12 = CF-MODULO
1077 )
1078 )
1079
1080 ; add a constraint such that there is no seventh harmonic interval if cf is at the
1081 top
1082 (defun add-no-seventh-cst (h-intervals is-cf-lower-arr &optional (is-cst-arr nil))
1083     (loop
1084         for h in h-intervals
1085         for b in is-cf-lower-arr
1086         for i from 0 below (length h-intervals)
1087         do
1088             (let (
1089                 (b-not (gil::add-bool-var *sp* 0 1)) ; b-not = !b
1090                 (is-cst (nth i is-cst-arr)) ; is-cst = is-cst-arr[i]
1091                 (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b-not and is-cst
1092             )
1093             (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b
1094             (if (null is-cst)
1095                 (gil::g-op *sp* b-not gil::BOT_AND TRUE b-and) ; b-and = b-not
1096                 (gil::g-op *sp* b-not gil::BOT_AND is-cst b-and) ; b-and = b-not and
1097                     is-cst
1098             )
1099             (gil::g-rel-reify *sp* h gil::IRT_NQ 10 b-and gil::RM_IMP) ; b-and => h
1100                 != 10
1101             (gil::g-rel-reify *sp* h gil::IRT_NQ 11 b-and gil::RM_IMP) ; b-and => h
1102                 != 11
1103         )
1104     )
1105 )
1106
1107 ; add a constraint such that there is no second harmonic interval if:
1108 ; - cf is at the bass AND
1109 ; - octave/unison harmonic interval precedes it
1110 (defun add-no-second-cst (h-intervals-arsis h-intervals-thesis is-cf-lower-arr &
1111     optional (is-cst-arr nil))
1112     (loop
1113         for ia in h-intervals-arsis
1114         for it in h-intervals-thesis
1115         for b in is-cf-lower-arr
1116         for i from 0 below (length h-intervals-arsis)
1117         do
1118             (let (
1119                 (b-uni (gil::add-bool-var *sp* 0 1)) ; b-uni = (ia == 0)
1120                 (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b AND b-uni
1121                 (is-cst (true-if-null is-cst-arr i)) ; is-cst = is-cst-arr[i] or TRUE
1122                 (b-and-cst (gil::add-bool-var *sp* 0 1)) ; b-and-cst = b-and AND is-cst

```

```

1115 )
1116 (gil::g-rel-reify *sp* ia gil::IRT_EQ 0 b-uni) ; b-uni = (ia == 0)
1117 (gil::g-op *sp* b gil::BOT_AND b-uni b-and) ; b-and = b AND b-uni
1118 (gil::g-op *sp* b-and gil::BOT_AND is-cst b-and-cst) ; b-and-cst = b-and
      AND is-cst
1119 (gil::g-rel-reify *sp* it gil::IRT_NQ 1 b-and-cst gil::RM_IMP)
1120 (gil::g-rel-reify *sp* it gil::IRT_NQ 2 b-and-cst gil::RM_IMP)
1121 )
1122 )
1123 )
1124
1125 ; add a constraint such that there is no melodic interval greater than @jump (8,
      minor 6th by default)
1126 (defun add-no-m-jump-cst (m-intervals &optional (jump 8))
1127   (gil::g-rel *sp* m-intervals gil::IRT_LQ jump)
1128 )
1129
1130 ; add a constraint such that m-intervals does not belong to [9, 10, 11]
1131 (defun add-no-m-jump-extend-cst (m-intervals &optional (is-cst-arr nil))
1132   (if (null is-cst-arr)
1133     ; then
1134     (progn
1135       (gil::g-rel *sp* m-intervals gil::IRT_NQ 9)
1136       (gil::g-rel *sp* m-intervals gil::IRT_NQ 10)
1137       (gil::g-rel *sp* m-intervals gil::IRT_NQ 11)
1138     )
1139     ; else
1140     (progn
1141       (loop
1142         for m in m-intervals
1143         for b in is-cst-arr
1144         do
1145           (gil::g-rel-reify *sp* m gil::IRT_NQ 9 b gil::RM_IMP)
1146           (gil::g-rel-reify *sp* m gil::IRT_NQ 10 b gil::RM_IMP)
1147           (gil::g-rel-reify *sp* m gil::IRT_NQ 11 b gil::RM_IMP)
1148         )
1149       )
1150     )
1151 )
1152
1153 ; add melodic interval constraints such that:
1154 ; - minor sixth intervals and octave intervals implies that is-nbour is true
1155 ; - no seventh intervals
1156 (defun add-m-inter-arsis-cst (m-intervals-ta is-nbour-arr)
1157   (loop
1158     for m in m-intervals-ta
1159     for n in is-nbour-arr
1160     do
1161       (let (
1162         (b-maj-six (gil::add-bool-var *sp* 0 1)) ; for (m = 9)
1163         (b-min-sev (gil::add-bool-var *sp* 0 1)) ; for (m == 10)
1164         (b-maj-sev (gil::add-bool-var *sp* 0 1)) ; for (m == 11)
1165         (b-or (gil::add-bool-var *sp* 0 1)) ; temporary variable for (
          b-min-sev or b-maj-sev)
1166       )
1167       (gil::g-rel-reify *sp* m gil::IRT_EQ 12 n gil::RM_PMI) ; m == 12
          implies n is true
1168       (gil::g-rel-reify *sp* m gil::IRT_EQ 9 b-maj-six) ; b-maj-six = (m
          == 9)
1169       (gil::g-rel-reify *sp* m gil::IRT_EQ 10 b-min-sev) ; b-min-sev = (m
          == 10)
1170       (gil::g-rel-reify *sp* m gil::IRT_EQ 11 b-maj-sev) ; b-maj-sev = (m
          == 11)

```

```

1171         (gil::g-op *sp* b-min-sev gil::BOT_OR b-maj-sev b-or) ; b-or = (
1172             b-min-sev or b-maj-sev)
1173     )
1174 )
1175 )
1176
1177 ; add melodic interval constraints such that there is no chromatic interval:
1178 ; - no m1 == 1 and m2 == 2 OR
1179 ; - no m1 == -1 and m2 == -2
1180 (defun add-no-chromatic-m-cst (m-intervals-brut m2-intervals-brut)
1181     (loop
1182         for m1 in (rest m-intervals-brut)
1183         for m2 in m2-intervals-brut do
1184             (let (
1185                 (b1 (gil::add-bool-var *sp* 0 1)) ; for (m1 == 1)
1186                 (b2 (gil::add-bool-var *sp* 0 1)) ; for (m2 == 2)
1187                 (b3 (gil::add-bool-var *sp* 0 1)) ; for (m1 == -1)
1188                 (b4 (gil::add-bool-var *sp* 0 1)) ; for (m2 == -2)
1189             )
1190                 (gil::g-rel-reify *sp* m1 gil::IRT_EQ 1 b1) ; b1 = (m1 == 1)
1191                 (gil::g-rel-reify *sp* m2 gil::IRT_EQ 2 b2) ; b2 = (m2 == 2)
1192                 (gil::g-op *sp* b1 gil::BOT_AND b2 0) ; not(b1 and b2)
1193                 (gil::g-rel-reify *sp* m1 gil::IRT_EQ -1 b3) ; b3 = (m1 == -1)
1194                 (gil::g-rel-reify *sp* m2 gil::IRT_EQ -2 b4) ; b4 = (m2 == -2)
1195                 (gil::g-op *sp* b3 gil::BOT_AND b4 0) ; not(b3 and b4)
1196             )
1197         )
1198     )
1199
1200 ; add melodic interval constraints such that there is no chromatic interval:
1201 ; - no m1 == 1 and m2 == 1 OR
1202 ; - no m1 == -1 and m2 == -1
1203 ; @m-intervals-brut: list of all the melodic intervals
1204 (defun add-no-chromatic-allm-cst (m-intervals-brut)
1205     (loop
1206         for m1 in m-intervals-brut
1207         for m2 in (rest m-intervals-brut) do
1208             (let (
1209                 (b1 (gil::add-bool-var *sp* 0 1)) ; for (m1 == 1)
1210                 (b2 (gil::add-bool-var *sp* 0 1)) ; for (m2 == 1)
1211                 (b3 (gil::add-bool-var *sp* 0 1)) ; for (m1 == -1)
1212                 (b4 (gil::add-bool-var *sp* 0 1)) ; for (m2 == -1)
1213             )
1214                 (gil::g-rel-reify *sp* m1 gil::IRT_EQ 1 b1) ; b1 = (m1 == 1)
1215                 (gil::g-rel-reify *sp* m2 gil::IRT_EQ 1 b2) ; b2 = (m2 == 1)
1216                 (gil::g-op *sp* b1 gil::BOT_AND b2 0) ; not(b1 and b2)
1217                 (gil::g-rel-reify *sp* m1 gil::IRT_EQ -1 b3) ; b3 = (m1 == -1)
1218                 (gil::g-rel-reify *sp* m2 gil::IRT_EQ -1 b4) ; b4 = (m2 == -1)
1219                 (gil::g-op *sp* b3 gil::BOT_AND b4 0) ; not(b3 and b4)
1220             )
1221         )
1222     )
1223
1224 ; create the motions array based on the melodic intervals of the melodic intervals
1225 ; it is given
1226 (defun create-motions (m-intervals-brut cf-brut-m-intervals motions costs
1227     is-not-lowest-arr)
1228     (loop
1229         for p in m-intervals-brut
1230         for q in cf-brut-m-intervals
1231         for m in motions

```

```

1230   for c in costs
1231   for is-not-lowest in (rest is-not-lowest-arr)
1232   do
1233       (let (
1234           ; boolean variables
1235           (b-pu (gil::add-bool-var *sp* 0 1)) ; boolean p up
1236           (b-qu (gil::add-bool-var *sp* 0 1)) ; boolean q up
1237           (b-ps (gil::add-bool-var *sp* 0 1)) ; boolean p stays
1238           (b-qs (gil::add-bool-var *sp* 0 1)) ; boolean q stays
1239           (b-pd (gil::add-bool-var *sp* 0 1)) ; boolean p down
1240           (b-qd (gil::add-bool-var *sp* 0 1)) ; boolean q down
1241           ; direct motion
1242           (b-both-up (gil::add-bool-var *sp* 0 1)) ; boolean both up
1243           (b-both-stays (gil::add-bool-var *sp* 0 1)) ; boolean both stays
1244           (b-both-down (gil::add-bool-var *sp* 0 1)) ; boolean both down
1245           (dm-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1246           (dm-or2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1247           (is-direct (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1248           ; oblique motion
1249           (b-pu-qs (gil::add-bool-var *sp* 0 1)) ; boolean p up and q stays
1250           (b-pd-qs (gil::add-bool-var *sp* 0 1)) ; boolean p down and q stays
1251           (b-ps-qu (gil::add-bool-var *sp* 0 1)) ; boolean p stays and q up
1252           (b-ps-qd (gil::add-bool-var *sp* 0 1)) ; boolean p stays and q down
1253           (om-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1254           (om-or2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1255           (om-or3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1256           (is-oblique (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1257           ; contrary motion
1258           (b-pu-qd (gil::add-bool-var *sp* 0 1)) ; boolean p up and q down
1259           (b-pd-qu (gil::add-bool-var *sp* 0 1)) ; boolean p down and q up
1260           (cm-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1261           (is-contrary (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1262           ; is lowest
1263           (is-lowest (gil::add-bool-var *sp* 0 1))
1264       )
1265       (gil::g-rel-reify *sp* p gil::IRT_LE 0 b-pd) ; b-pd = (p < 0)
1266       (gil::g-rel-reify *sp* p gil::IRT_EQ 0 b-ps) ; b-ps = (p == 0)
1267       (gil::g-rel-reify *sp* p gil::IRT_GR 0 b-pu) ; b-pu = (p > 0)
1268       (gil::g-rel-reify *sp* q gil::IRT_LE 0 b-qd) ; b-qd = (q < 0)
1269       (gil::g-rel-reify *sp* q gil::IRT_EQ 0 b-qs) ; b-qs = (q == 0)
1270       (gil::g-rel-reify *sp* q gil::IRT_GR 0 b-qu) ; b-qu = (q > 0)
1271       ; direct motion
1272       (gil::g-op *sp* b-pu gil::BOT_AND b-qu b-both-up) ; b-both-up = (
1273           b-pu and b-qu)
1274       (gil::g-op *sp* b-ps gil::BOT_AND b-qs b-both-stays) ; b-both-stays =
1275           (b-ps and b-qs)
1276       (gil::g-op *sp* b-pd gil::BOT_AND b-qd b-both-down) ; b-both-down =
1277           (b-pd and b-qd)
1278       (gil::g-op *sp* b-both-up gil::BOT_OR b-both-stays dm-or1) ; dm-or1 =
1279           (b-both-up or b-both-stays)
1280       (gil::g-op *sp* dm-or1 gil::BOT_OR b-both-down dm-or2) ; dm-or2 = (
1281           dm-or1 or b-both-down)
1282       (gil::g-op *sp* dm-or2 gil::BOT_AND is-not-lowest is-direct)
1283       (gil::g-rel-reify *sp* m gil::IRT_EQ DIRECT is-direct) ; m = 1 if
1284           dm-or2
1285       (gil::g-rel-reify *sp* c gil::IRT_EQ *dir-motion-cost* is-direct gil
1286           ::RM_IMP) ; add the cost of direct motion
1287       ; oblique motion
1288       (gil::g-op *sp* b-pu gil::BOT_AND b-qs b-pu-qs) ; b-pu-qs = (b-pu
1289           and b-qs)
1290       (gil::g-op *sp* b-pd gil::BOT_AND b-qs b-pd-qs) ; b-pd-qs = (b-pd
1291           and b-qs)

```

```

1283 (gil::g-op *sp* b-ps gil::BOT_AND b-qu b-ps-qu) ; b-ps-qu = (b-ps
1284 and b-qu)
1285 (gil::g-op *sp* b-ps gil::BOT_AND b-qd b-ps-qd) ; b-ps-qd = (b-ps
1286 and b-qd)
1287 (gil::g-op *sp* b-pu-qs gil::BOT_OR b-pd-qs om-or1) ; om-or1 = (
1288 b-pu-qs or b-pd-qs)
1289 (gil::g-op *sp* om-or1 gil::BOT_OR b-ps-qu om-or2) ; om-or2 = (
1290 om-or1 or b-ps-qu)
1291 (gil::g-op *sp* om-or2 gil::BOT_OR b-ps-qd om-or3) ; om-or3 = (
1292 om-or2 or b-ps-qd)
1293 (gil::g-op *sp* om-or3 gil::BOT_AND is-not-lowest is-oblique)
1294 (gil::g-rel-reify *sp* m gil::IRT_EQ OBLIQUE is-oblique) ; m = 0 if
1295 om-or3
1296 (gil::g-rel-reify *sp* c gil::IRT_EQ *obl-motion-cost* is-oblique
1297 gil::RM_IMP) ; add the cost of oblique motion
1298 ; contrary motion
1299 (gil::g-op *sp* b-pu gil::BOT_AND b-qd b-pu-qd) ; b-pu-qd = (b-pu
1300 and b-qd)
1301 (gil::g-op *sp* b-pd gil::BOT_AND b-qu b-pd-qu) ; b-pd-qu = (b-pd
1302 and b-qu)
1303 (gil::g-op *sp* b-pu-qd gil::BOT_OR b-pd-qu cm-or1) ; cm-or1 = (
1304 b-pu-qd or b-pd-qu)
1305 (gil::g-op *sp* cm-or1 gil::BOT_AND is-contrary is-contrary)
1306 (gil::g-rel-reify *sp* m gil::IRT_EQ CONTRARY is-contrary) ; m = -1
1307 if cm-or1
1308 (gil::g-rel-reify *sp* c gil::IRT_EQ *con-motion-cost* is-contrary
1309 gil::RM_IMP) ; add the cost of contrary motion
1310 ; is bass (no motion)
1311 (gil::g-op *sp* is-not-lowest gil::BOT_XOR is-lowest 1)
1312 (gil::g-rel-reify *sp* m gil::IRT_EQ -1 is-lowest) ;
1313 (gil::g-rel-reify *sp* c gil::IRT_EQ 0 is-lowest gil::RM_IMP) ;
1314 )
1315 )
1316 )
1317 ; create the motion list variable as it is perceived by the human ear,
1318 ; i.e. if the interval between the thesis and the arsis note is greater than a third
1319 ,
1320 ; then the motion is perceived from the arsis note and not from the thesis note
1321 ; @m-intervals-ta: melodic intervals between the thesis and the arsis note
1322 ; @motions: motions perceived from the thesis note
1323 ; @motions-arsis: motions perceived from the arsis note
1324 ; @real-motions: motions perceived by the human ear
1325 (defun create-real-motions (m-intervals-ta motions motions-arsis real-motions
1326 motions-costs motions-arsis-costs real-motions-costs)
1327 (loop
1328   for tai in m-intervals-ta
1329   for t-move in motions
1330   for a-move in motions-arsis
1331   for r-move in real-motions
1332   for t-c in motions-costs
1333   for a-c in motions-arsis-costs
1334   for r-c in real-motions-costs
1335   do
1336     (let (
1337       (b (gil::add-bool-var *sp* 0 1)) ; for (tai > 4)
1338     )
1339       (gil::g-rel-reify *sp* tai gil::IRT_GR 4 b) ; b = (tai > 4)
1340       (gil::g-ite *sp* b a-move t-move r-move) ; r-move = (b ? a-move :
1341         t-move)
1342       (gil::g-ite *sp* b a-c t-c r-c) ; r-c = (b ? a-c : t-c)
1343     )
1344   )
1345 )

```

```

1331 )
1332
1333 ; add the constraint such that there is no perfect consonance in thesis that is
      reached by direct motion
1334 (defun add-no-direct-move-to-p-cons-cst (motions is-p-cons-arr is-not-lowest-arr &
      optional (r t))
1335   (loop
1336     for m in motions
1337     for b in (rest-if is-p-cons-arr r)
1338     for is-not-lowest in (rest-if is-not-lowest-arr r)
1339     do
1340       (let
1341         (
1342           (is-p-cons-and-is-not-lowest (gil::add-bool-var *sp* 0 1))
1343         )
1344         (gil::g-op *sp* is-not-lowest gil::BOT_AND b
1345           is-p-cons-and-is-not-lowest)
1346         (gil::g-rel-reify *sp* m gil::IRT_NQ DIRECT
1347           is-p-cons-and-is-not-lowest gil::RM_IMP) ; if it is a p-cons and
1348           ; of course nothing happens if it is the lowest stratum
1349       )
1350   )
1351
1352 ; add the costs such that there if a perfect consonance is reached by direct motion
      a cost is set
1353 (defun compute-no-direct-move-to-p-cons-costs-cst (motions cost-array is-p-cons-arr
      &optional (r t))
1354   (loop
1355     for m in motions
1356     for c in cost-array
1357     for is-p-cons in (rest-if is-p-cons-arr r)
1358     do (let (
1359       (is-direct-move (gil::add-bool-var *sp* 0 1))
1360       (is-direct-move-to-p-cons (gil::add-bool-var *sp* 0 1))
1361       (is-not-direct-move-to-p-cons (gil::add-bool-var *sp* 0 1))
1362     )
1363       (gil::g-rel-reify *sp* m gil::IRT_EQ DIRECT is-direct-move) ;
1364       is-direct-move = (m = direct)
1365       (gil::g-op *sp* is-direct-move gil::BOT_AND is-p-cons
1366         is-direct-move-to-p-cons) ; is-direct-move-to-p-cons = (
1367         is-direct-move AND is-p-cons)
1368       (gil::g-op *sp* is-direct-move-to-p-cons gil::BOT_XOR
1369         is-not-direct-move-to-p-cons 1)
1370       (gil::g-rel-reify *sp* c gil::IRT_EQ *direct-move-to-p-cons-cost*
1371         is-direct-move-to-p-cons gil::RM_IMP) ; if
1372         is-direct-move-to-p-cons then cost is set
1373       (gil::g-rel-reify *sp* c gil::IRT_EQ 0 is-not-direct-move-to-p-cons
1374         gil::RM_IMP) ; else it is equal to 0
1375     )
1376   )
1377
1378 ; add the constraint that there cannot be two ascending sixths
1379 (defun add-no-ascending-sixths-cst (h-intervals cp)
1380   (dotimes (i *cf-last-index)
1381     (let (
1382       (first-is-sixth (gil::add-bool-var *sp* 0 1))
1383       (first-h-equals-8 (gil::add-bool-var *sp* 0 1))
1384       (first-h-equals-9 (gil::add-bool-var *sp* 0 1))

```

```

1380         (second-is-sixth (gil::add-bool-var *sp* 0 1))
1381         (second-h-equals-8 (gil::add-bool-var *sp* 0 1))
1382         (second-h-equals-9 (gil::add-bool-var *sp* 0 1))
1383         (both-h-are-sixths (gil::add-bool-var *sp* 0 1))
1384         (is-ascending (gil::add-bool-var *sp* 0 1))
1385     )
1386     (gil::g-rel-reify *sp* (nth i h-intervals) gil::IRT_EQ 8 first-h-equals-8)
1387     (gil::g-rel-reify *sp* (nth i h-intervals) gil::IRT_EQ 9 first-h-equals-9)
1388     (gil::g-op *sp* first-h-equals-8 gil::BOT_OR first-h-equals-9 first-is-sixth
1389         )
1390     (gil::g-rel-reify *sp* (nth (+ i 1) h-intervals) gil::IRT_EQ 8
1391         second-h-equals-8)
1392     (gil::g-rel-reify *sp* (nth (+ i 1) h-intervals) gil::IRT_EQ 9
1393         second-h-equals-9)
1394     (gil::g-op *sp* second-h-equals-8 gil::BOT_OR second-h-equals-9
1395         second-is-sixth)
1396     (gil::g-op *sp* first-is-sixth gil::BOT_AND second-is-sixth
1397         both-h-are-sixths)
1398     (gil::g-rel-reify *sp* (nth i cp) gil::IRT_LE (nth (+ i 1) cp) is-ascending)
1399     (gil::g-op *sp* both-h-are-sixths gil::BOT_AND is-ascending 0) ; prohibit
1400         that we have ascending sixths
1401 )
1402 )
1403 )
1404 ; add the cost of having two successive perfect consonances between two voices
1405 (defun add-no-successive-p-cons-cst (is-p-cons-array successive-p-cons-cost)
1406     (loop
1407         for i from 0 to (- (length is-p-cons-array) 2)
1408         do (let ((successive-p-cons (gil::add-bool-var *sp* 0 1)))
1409             (gil::g-op *sp* (nth i is-p-cons-array) gil::BOT_AND (nth (+ i 1)
1410                 is-p-cons-array) successive-p-cons)
1411             (gil::g-rel-reify *sp* (nth i successive-p-cons-cost) gil::IRT_EQ *
1412                 succ-p-cons-cost* successive-p-cons)
1413         ))
1414 )
1415 ; add the cost of having two successive perfect consonances between two voices - 4th
1416     species -> successive FIFTHS are allowed
1417 (defun add-no-successive-p-cons-4th-species-cst (is-p-cons-array h-intervals
1418     successive-p-cons-cost)
1419     (dotimes (i (- (length h-intervals) 1))
1420         (let (
1421             (first-not-fifth (gil::add-bool-var *sp* 0 1))
1422             (second-not-fifth (gil::add-bool-var *sp* 0 1))
1423             (not-successive-fifths (gil::add-bool-var *sp* 0 1))
1424             (successive-p-cons (gil::add-bool-var *sp* 0 1))
1425             (successive-p-cons-and-not-successive-fifths (gil::add-bool-var *sp* 0
1426                 1))
1427         )
1428             (gil::g-rel-reify *sp* (nth i h-intervals) gil::IRT_NQ 7 first-not-fifth
1429                 )
1430             (gil::g-rel-reify *sp* (nth (+ 1 i) h-intervals) gil::IRT_NQ 7
1431                 second-not-fifth)
1432             (gil::g-op *sp* first-not-fifth gil::BOT_OR second-not-fifth
1433                 not-successive-fifths)
1434             (gil::g-op *sp* (nth i is-p-cons-array) gil::BOT_AND (nth (+ i 1)
1435                 is-p-cons-array) successive-p-cons)
1436             (gil::g-op *sp* successive-p-cons gil::BOT_AND not-successive-fifths
1437                 successive-p-cons-and-not-successive-fifths)

```

```

1427         (gil::g-rel-reify *sp* (nth i successive-p-cons-cost) gil::IRT_EQ *
           succ-p-cons-cost* successive-p-cons-and-not-successive-fifths) ;
           successive p cons and not successive fifths -> set the cost
1428     )
1429 )
1430 )
1431
1432 ; add the cost of having two successive perfect consonances between two voices - 2nd
       species -> successive FIFTHS are allowed IF there is a third in between
1433 (defun add-no-successive-p-cons-2nd-species-cst (is-p-cons-array h-intervals
           m-succ-intervals successive-p-cons-cost)
1434     (loop
1435       for i from 0 to (- (length is-p-cons-array) 2)
1436       do (let (
1437           ; 1st case
1438           (first-not-fifth (gil::add-bool-var *sp* 0 1))
1439           (second-not-fifth (gil::add-bool-var *sp* 0 1))
1440           (not-successive-fifths (gil::add-bool-var *sp* 0 1))
1441
1442           (successive-p-cons (gil::add-bool-var *sp* 0 1))
1443           (successive-p-cons-and-not-successive-fifths (gil::add-bool-var *sp* 0
1444               1))
1445
1446           ; 2nd case
1447           (m-is-not-third-1 (gil::add-bool-var *sp* 0 1))
1448           (m-is-not-third-2 (gil::add-bool-var *sp* 0 1))
1449           (m-is-not-third (gil::add-bool-var *sp* 0 1))
1450
1451           (first-is-fifth (gil::add-bool-var *sp* 0 1))
1452           (second-is-fifth (gil::add-bool-var *sp* 0 1))
1453           (successive-fifths (gil::add-bool-var *sp* 0 1))
1454           (successive-fifths-and-not-third (gil::add-bool-var *sp* 0 1))
1455
1456           ; finally
1457           (apply-the-cost (gil::add-bool-var *sp* 0 1)) ; true if the cost must be
           applied, else false
1458       )
1459       ; first case : the successive perfect consonances are not successive fifths
1460       (gil::g-rel-reify *sp* (nth i h-intervals) gil::IRT_NQ 7 first-not-fifth)
1461       (gil::g-rel-reify *sp* (nth (+ 1 i) h-intervals) gil::IRT_NQ 7
           second-not-fifth)
1462       (gil::g-op *sp* first-not-fifth gil::BOT_OR second-not-fifth
           not-successive-fifths)
1463
1464       (gil::g-op *sp* (nth i is-p-cons-array) gil::BOT_AND (nth (+ i 1)
           is-p-cons-array) successive-p-cons)
1465       (gil::g-op *sp* successive-p-cons gil::BOT_AND not-successive-fifths
           successive-p-cons-and-not-successive-fifths)
1466
1467       ; second case : the successive perfect consonants are fifths
1468       (gil::g-rel-reify *sp* (nth i m-succ-intervals) gil::IRT_NQ 3
           m-is-not-third-1)
1469       (gil::g-rel-reify *sp* (nth i m-succ-intervals) gil::IRT_NQ 4
           m-is-not-third-2)
1470       (gil::g-op *sp* m-is-not-third-1 gil::BOT_AND m-is-not-third-2
           m-is-not-third)
1471
1472       (gil::g-rel-reify *sp* (nth i h-intervals) gil::IRT_EQ 7 first-is-fifth)
1473       (gil::g-rel-reify *sp* (nth (+ 1 i) h-intervals) gil::IRT_EQ 7
           second-is-fifth)
1474       (gil::g-op *sp* first-is-fifth gil::BOT_AND second-is-fifth
           successive-fifths)

```

```

1474      (gil::g-op *sp* m-is-not-third gil::BOT_AND successive-fifths
1475          successive-fifths-and-not-third)
1476      ; finally
1477      (gil::g-op *sp* successive-p-cons-and-not-successive-fifths gil::BOT_OR
1478          successive-fifths-and-not-third apply-the-cost)
1479      (gil::g-rel-reify *sp* (nth i successive-p-cons-cost) gil::IRT_EQ *
1480          succ-p-cons-cost* apply-the-cost)
1481    ))
1482  )
1483  ; computes the variety cost, i.e. the number of times a note repeats itself in a
1484  ; frame of 7 measures
1485  (defun compute-variety-cost (cp variety-cost)
1486    (let (
1487      (k 0)
1488    )
1489      (loop
1490        for i from 0 below (length cp)
1491        do (loop
1492          ; for each note in the three following
1493          for j from (+ i 1) to (min (+ i 3) (- (length cp) 1))
1494          do(let (
1495            (is-equal (gil::add-bool-var *sp* 0 1))
1496            (is-not-equal (gil::add-bool-var *sp* 0 1))
1497          )
1498            (gil::g-rel-reify *sp* (nth i cp) gil::IRT_EQ (nth j cp) is-equal)
1499            (gil::g-rel-reify *sp* (nth i cp) gil::IRT_NQ (nth j cp)
1500              is-not-equal)
1501            (gil::g-rel-reify *sp* (nth k variety-cost) gil::IRT_EQ *
1502              variety-cost* is-equal gil::RM_IMP) ; if it is equal set the
1503              cost
1504            (gil::g-rel-reify *sp* (nth k variety-cost) gil::IRT_EQ 0
1505              is-not-equal gil::RM_IMP)
1506          )
1507          (setf k (+ 1 k))
1508        )
1509      )
1510    )
1511  )
1512  ; return the rest of the list if the boolean is true, else return the list
1513  (defun rest-if (l b)
1514    (if b
1515      (rest l)
1516      l
1517    )
1518  )
1519  ; TODO pass to new version function below
1520  ; add the constraint such that there is no battuta kind of motion, i.e.:
1521  ; - contrary motion
1522  ; - skip in the upper voice
1523  ; - lead to an octave
1524  (defun add-no-battuta-cst (motions h-intervals m-intervals-brut is-cf-lower-arr &
1525    optional (is-cst-arr nil))
1526    (loop
1527      for move in motions
1528      for hi in (rest h-intervals)
1529      for mi in m-intervals-brut
1530      for b in (butlast is-cf-lower-arr)

```

```

1528   for i from 0 below *cf-last-index
1529   do
1530     (let (
1531       (is-cm (gil::add-bool-var *sp* 0 1)) ; is contrary motion
1532       (is-oct (gil::add-bool-var *sp* 0 1)) ; is moving to octave
1533       (is-cp-down (gil::add-bool-var *sp* 0 1)) ; is counterpoint going down
1534       (b-and1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1535       (b-and2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1536       (b-and3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1537     )
1538       (gil::g-rel-reify *sp* move gil::IRT_EQ CONTRARY is-cm) ; is-cm = (m ==
1539         -1)
1540       (gil::g-rel-reify *sp* hi gil::IRT_EQ 0 is-oct) ; is-oct = (hi == 0)
1541       (gil::g-rel-reify *sp* mi gil::IRT_LE -4 is-cp-down) ; is-cp-down = (mi
1542         < -4)
1543       (gil::g-op *sp* is-cm gil::BOT_AND is-oct b-and1) ; b-and1 = (is-cm and
1544         is-oct)
1545       (gil::g-op *sp* b-and1 gil::BOT_AND is-cp-down b-and2) ; b-and2 = (
1546         b-and1 and is-cp-down)
1547       (if (null is-cst-arr)
1548         ; then constraint is always added
1549         (gil::g-op *sp* b-and2 gil::BOT_AND b 0) ; (is-cm and is-oct and
1550           is-cp-down and b) = FALSE
1551         ; else constraint is added only if the current note is constrained
1552         (progn
1553           (gil::g-op *sp* b-and2 gil::BOT_AND b b-and3) ; b-and3 = (b-and2
1554             and b)
1555           ; is-cst => (b-and3 == 0) can be written as not (is-cst and
1556             b-and3)
1557           (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and3 0)
1558         )
1559       )
1560     )
1561   )
1562 )
1563
1564 ; TEST new version
1565 ; add the constraint such that there is no battuta kind of motion, i.e.:
1566 ; - contrary motion
1567 ; - skip in the upper voice
1568 ; - lead to an octave
1569 (defun add-no-battuta-bis-cst (motions h-intervals m-intervals-brut
1570   cf-brut-m-intervals is-cf-lower-arr &optional (is-cst-arr nil))
1571   (loop
1572     for move in motions
1573     for hi in (rest h-intervals)
1574     for mi in m-intervals-brut
1575     for cf-mi in cf-brut-m-intervals
1576     for b in (butlast is-cf-lower-arr)
1577     for i from 0 below *cf-last-index
1578     do
1579       (let (
1580         (is-cm (gil::add-bool-var *sp* 0 1)) ; is contrary motion
1581         (is-oct (gil::add-bool-var *sp* 0 1)) ; is moving to octave
1582         (is-cp-down (gil::add-bool-var *sp* 0 1)) ; is counterpoint going down
1583           more than 4 semi-tones
1584         (is-cf-down (gil::add-bool-var *sp* 0 1)) ; is cantus firmus going down
1585           more than 4 semi-tones
1586         (b-not (gil::add-bool-var *sp* 0 1)) ; !b = cantus firmus is not the
1587           bass
1588         (b-and1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1589         (b-and2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
1590         (b-and3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean

```

```

1580 )
1581 (gil::g-rel-reify *sp* move gil::IRT_EQ CONTRARY is-cm) ; is-cm = (m ==
1582 0)
1583 (gil::g-rel-reify *sp* hi gil::IRT_EQ 0 is-oct) ; is-oct = (hi == 0)
1584 (gil::g-rel-reify *sp* mi gil::IRT_LE -4 is-cp-down) ; is-cp-down = (mi
1585 < -4)
1586 (gil::g-rel-reify *sp* cf-mi gil::IRT_LE -4 is-cf-down) ; is-cf-down = (
1587 cf-mi < -4)
1588 (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b
1589 (gil::g-op *sp* is-cm gil::BOT_AND is-oct b-and1) ; b-and1 = (is-cm and
1590 is-oct)
1591 (gil::g-op *sp* b gil::BOT_AND is-cp-down b-and2) ; b-and2 = (b-and1 and
1592 is-cp-down)
1593 (gil::g-op *sp* b-not gil::BOT_AND is-cf-down b-and3) ; b-and3 = (b-not
1594 and is-cf-down)
1595
1596 (if (null is-cst-arr)
1597 ; then constraint is always added
1598 (progn
1599 ; first case: (is-cm and is-oct and b and is-cp-down) = FALSE
1600 (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 0)
1601 ; second case: (is-cm and is-oct and b-not and is-cf-down) =
1602 FALSE
1603 (gil::g-op *sp* b-and1 gil::BOT_AND b-and3 0)
1604 )
1605 ; else constraint is added only if the current note is constrained
1606 (progn (let (
1607 (b-and4 (gil::add-bool-var *sp* 0 1)) ; first case
1608 (b-and5 (gil::add-bool-var *sp* 0 1)) ; second case
1609 )
1610 (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 b-and4) ; first case:
1611 b-and4 = (b-and1 and b-and2)
1612 (gil::g-op *sp* b-and1 gil::BOT_AND b-and3 b-and5) ; second case
1613 : b-and5 = (b-and1 and b-and3)
1614 ; is-cst => (b-and == 0) can be written as not (is-cst and b-and
1615 )
1616 (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and4 0) ;
1617 first case
1618 (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and5 0) ;
1619 second case
1620 ))
1621 )
1622 )
1623 )
1624 ;; 5th species methods
1625 ; add the constraint such that the selected notes are the same as the midi-selected
1626 notes
1627 (defun add-selected-notes-cst (selected midi-selected cp)
1628 (print "Adding selected notes constraint")
1629 (print selected)
1630 (print midi-selected)
1631 (loop
1632 for i in selected
1633 for ms in midi-selected
1634 do
1635 (setq i+1 (+ i 1))
1636 (gil::g-rel *sp* (nth i cp) gil::IRT_EQ (first ms))
1637 (gil::g-rel *sp* (nth i+1 cp) gil::IRT_EQ (second ms))
1638 )
1639 )
1640 )

```

```

1630 ; add constraints such that the boolean array is true if the simple constraint is
      respected
1631 (defun create-simple-boolean-arr (candidate-arr rel-type cst b-arr)
1632   (loop
1633     for c in candidate-arr
1634     for b in b-arr
1635     do
1636       (gil::g-rel-reify *sp* c rel-type cst b)
1637   )
1638 )
1639
1640 ; do the gil::g-ite constraint but only if and-cond is true
1641 (defun and-ite (test then else var and-cond)
1642   (let (
1643     (b-and-then (gil::add-bool-var *sp* 0 1)) ; b-and-then = test and and-cond
1644     (test-not (gil::add-bool-var *sp* 0 1)) ; test-not = !test
1645     (b-and-else (gil::add-bool-var *sp* 0 1)) ; b-and-else = !test and and-cond
1646   )
1647     (gil::g-op *sp* test gil::BOT_AND and-cond b-and-then) ; b-and-then = test
      and and-cond
1648     (gil::g-op *sp* test-not gil::BOT_EQV FALSE test-not) ; test-not = !test
1649     (gil::g-op *sp* test-not gil::BOT_AND and-cond b-and-else) ; b-and-else = !
      test and and-cond
1650     (gil::g-rel-reify *sp* var gil::IRT_EQ then b-and-then gil::RM_IMP) ;
      b-and-then => var = then
1651     (gil::g-rel-reify *sp* var gil::IRT_EQ else b-and-else gil::RM_IMP) ;
      b-and-else => var = else
1652   )
1653 )
1654
1655 ; merge the boolean arrays with the and operator
1656 (defun bot-merge-array (b-arr1 b-arr2 b-collect-arr &optional (bot gil::BOT_AND))
1657   (loop
1658     for b1 in b-arr1
1659     for b2 in b-arr2
1660     for b in b-collect-arr
1661     do
1662       (gil::g-op *sp* b1 bot b2 b)
1663   )
1664 )
1665
1666 ; merge the boolean arrays with the or operator and just return it
1667 (defun collect-bot-array (b-arr1 b-arr2 &optional (bot gil::BOT_AND))
1668   (let (
1669     (b-collect-arr (gil::add-bool-var-array *sp* (length b-arr1) 0 1))
1670   )
1671     (loop
1672       for b1 in b-arr1
1673       for b2 in b-arr2
1674       for b in b-collect-arr
1675       do
1676         (gil::g-op *sp* b1 bot b2 b)
1677     )
1678     b-collect-arr
1679   )
1680 )
1681
1682
1683 (defun collect-t-or-f-array (yes-arr no-arr)
1684   (collect-bot-array
1685     yes-arr
1686     (collect-not-array no-arr)
1687     gil::BOT_OR

```

```

1688 )
1689 )
1690
1691 (defun collect-not-array (arr)
1692   (collect-bot-array arr (gil::add-bool-var-array *sp* (length arr) 0 0) gil::
     BOT_EQV)
1693 )
1694
1695 ; do the gil::g-rel-reify constraint but use the condition that (b AND and-cond) is
     true
1696 (defun bot-reify (var rel-type cst b and-cond &optional (bot gil::BOT_AND) (mode gil
     ::RM_EQV))
1697   (let (
1698     (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b and and-cond
1699   )
1700     (gil::g-op *sp* b bot and-cond b-and) ; b-and = b and and-cond
1701     (gil::g-rel-reify *sp* var rel-type cst b-and mode) ; b-and == var rel-type
     cst
1702   )
1703 )
1704
1705 ; return the index of a note as all the notes are in a row,
1706 ; i.e. return the total index of the note at the given measure at the given beat
     assuming that we are in 4 4 time
1707 ; the index is 0-based, same for measure and beat
1708 (defun total-index (measure beat)
1709   (+ (* measure 4) beat)
1710 )
1711
1712 ; is-mostly-3rd is true if second, third and fourth notes are from 3rd species
1713 ; note that is-mostly-3rd-arr have a length 4 times shorter than is-3rd-species-arr
1714 (defun create-is-mostly-3rd-arr (is-3rd-species-arr is-mostly-3rd-arr)
1715   (loop
1716     for meas from 0 below (length is-mostly-3rd-arr)
1717     do
1718       (let (
1719         (b-23 (gil::add-bool-var *sp* 0 1)) ; b-23 = is-3rd-species-arr[meas][1]
           AND is-3rd-species-arr[meas][2]
1720       )
1721         ; b-23
1722         (gil::g-op *sp* (nth (total-index meas 1) is-3rd-species-arr) gil::
           BOT_AND (nth (total-index meas 2) is-3rd-species-arr) b-23)
1723         ; b-23 and "b-4" are stocked in is-mostly-3rd-arr[meas]
1724         (gil::g-op *sp* b-23 gil::BOT_AND (nth (total-index meas 3)
           is-3rd-species-arr) (nth meas is-mostly-3rd-arr))
1725       )
1726   )
1727 )
1728
1729 ; collect elements all the 4 elements of the array, i.e. n, n+4, n+8, n+12, etc.
1730 ; note: n is the offset
1731 (defun collect-by-4 (arr &optional (offset 0) (b nil) (up-bound 4))
1732   (setq len (if (eq offset 0) *cf-len *cf-last-index))
1733   (if (null b)
1734     ; then make a boolean array
1735     (setq ret (gil::add-bool-var-array *sp* len 0 1))
1736     ; else make a integer array
1737     (setq ret (gil::add-int-var-array *sp* len 0 up-bound))
1738   )
1739   (loop
1740     for i from offset below (length arr) by 4
1741     for j from 0 below len
1742     do

```

```

1743         (gil::g-rel *sp* (nth i arr) gil::IRT_EQ (nth j ret))
1744     )
1745     ret
1746 )
1747
1748 ; create an array for one beat from the entire array
1749 (defun create-by-4 (arr-from arr-to &optional (offset 0))
1750     (loop
1751         for i from offset below (length arr-from) by 4
1752         for j in arr-to
1753         do
1754             (gil::g-rel *sp* (nth i arr-from) gil::IRT_EQ j)
1755     )
1756 )
1757
1758 ; add a reify constraint if @b is not nil, else add a rel constraint
1759 (defun rel-reify-if (var rel-type cst &optional (b nil) (rm gil::RM_IMP))
1760     (if (null b)
1761         (gil::g-rel *sp* var rel-type cst)
1762         (gil::g-rel-reify *sp* var rel-type cst b rm)
1763     )
1764 )
1765
1766 ; return BoolVar true if nil element
1767 (defun true-if-null (arr i)
1768     (if (null arr)
1769         ; then
1770         TRUE
1771         ; else
1772         (nth i arr)
1773     )
1774 )
1775
1776 ; add the constraint such that if sp3 is 4th species, then sp4 is 0 and the next sp1
1777 ; is 4th species
1778 ; and vice versa (cannot have 4th species in first position without 4th species in
1779 ; third position)
1780 ; - sp-arr3: array of IntVar for species at the third position
1781 ; - sp-arr4: array of IntVar for species at the fourth position
1782 ; - sp-arr1: array of IntVar for species at the first position
1783 (defun add-4th-rythmic-cst (sp-arr3 sp-arr4 sp-arr1)
1784     (loop
1785         for sp3 in sp-arr3
1786         for sp4 in sp-arr4
1787         for sp1 in (rest sp-arr1)
1788         do
1789             (let (
1790                 (b-34 (gil::add-bool-var *sp* 0 1)) ; b-34 = sp3 == 4th species
1791                 (b-14 (gil::add-bool-var *sp* 0 1)) ; b-14 = sp1 == 4th species
1792             )
1793                 (gil::g-rel-reify *sp* sp3 gil::IRT_EQ 4 b-34) ; b-34 = sp3 == 4th
1794                     species
1795                 (gil::g-rel-reify *sp* sp1 gil::IRT_EQ 4 b-14) ; b-14 = sp1 == 4th
1796                     species
1797                 (gil::g-rel-reify *sp* sp4 gil::IRT_EQ 0 b-34 gil::RM_IMP) ; b-34 => sp4
1798                     == 0
1799                 (gil::g-op *sp* b-34 gil::BOT_EQV b-14 1) ; b-34 <=> b-14
1800             )
1801         )
1802     )
1803 )
1804
1805 ; add the constraint such that if n belongs to @species, then n+m have to exist (not
1806 0)

```

```

1800 ; by default, the constraint is added for the third species
1801 ; - species-arr: array of IntVar for species
1802 ; - spec: species to check
1803 ; - offset: offset to check
1804 (defun add-no-silence-cst (species-arr &key (spec 3) (offset 1))
1805   (loop
1806     for n in species-arr
1807     for n+m in (nthcdr offset species-arr)
1808     do
1809       (let (
1810         (b (gil::add-bool-var *sp* 0 1)) ; b = (n == species)
1811         )
1812         (gil::g-rel-reify *sp* n gil::IRT_EQ spec b) ; b = (n == spec)
1813         (gil::g-rel-reify *sp* n+m gil::IRT_NQ 0 b gil::RM_IMP) ; b => (n+m !=
1814           0)
1815         )
1816       )
1817 )
1818 ; add the constraint such that there is maximum 2 consecutive measures without 4th
1819 ; species
1820 (defun add-min-syncope-cst (third-sp-arr)
1821   (loop
1822     for sp1 in (nthcdr 1 third-sp-arr)
1823     for sp2 in (nthcdr 2 third-sp-arr)
1824     for sp3 in (nthcdr 3 third-sp-arr)
1825     do
1826       (let (
1827         (b1-not-4 (gil::add-bool-var *sp* 0 1)) ; b1-not-4 = sp1 != 4
1828         (b2-not-4 (gil::add-bool-var *sp* 0 1)) ; b2-not-4 = sp2 != 4
1829         (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b1-not-4 && b2-not-4
1830         )
1831         (gil::g-rel-reify *sp* sp1 gil::IRT_NQ 4 b1-not-4) ; b1-not-4 = sp1 != 4
1832         (gil::g-rel-reify *sp* sp2 gil::IRT_NQ 4 b2-not-4) ; b2-not-4 = sp2 != 4
1833         (gil::g-op *sp* b1-not-4 gil::BOT_AND b2-not-4 b-and) ; b-and = b1-not-4
1834           && b2-not-4
1835         (gil::g-rel-reify *sp* sp3 gil::IRT_EQ 4 b-and gil::RM_IMP) ; b-and =>
1836           sp3 == 4
1837         )
1838       )
1839     )
1840 )
1841 ; add all constraints to create a rhythmic and select what species to use
1842 ; mandatory rules are:
1843 ; - 4th species is only used in third and first position
1844 ; - 4th species in third position is followed by a 0 (no note/constraint) and then a
1845 ; 4th species
1846 ; - no 3rd species followed by 0
1847 ; classic rules are:
1848 ; - first and penultimate measure are 4th species
1849 ; - only 3rd and 4th species are used
1850 ; - 3rd species should represent at least 1/3 of the notes
1851 ; - 4th species should represent at least 1/4 of the notes
1852 (defun create-species-arr (species-arr solution-len &key (min-3rd-pc (* (- 1 (
1853   getparam 'pref-species-slider)) 0.66)) (min-4th-pc (* (getparam '
1854     pref-species-slider) 0.5)))
1855   (print "Create species array...")
1856   (let* (
1857     (count-3rd (gil::add-int-var-array *sp* solution-len 0 1))
1858     (count-4th (gil::add-int-var-array *sp* solution-len 0 1))
1859     (n-3rd-int (floor (* solution-len min-3rd-pc))) ; minimum number of 3rd
1860       species

```

```

1854 (n-4th-int (floor (* solution-len min-4th-pc))) ; minimum number of 4th
      species
1855 (sum-3rd (gil::add-int-var *sp* n-3rd-int solution-len)) ; set the bounds of
      sum-3rd
1856 (sum-4th (gil::add-int-var *sp* n-4th-int solution-len)) ; set the bounds of
      sum-4th
1857 )
1858 (setq *sp-arr (list
1859   (collect-by-4 species-arr 0 t)
1860   (collect-by-4 species-arr 1 t)
1861   (collect-by-4 species-arr 2 t)
1862   (collect-by-4 species-arr 3 t)
1863 ))
1864
1865 (print "Counting 3rd and 4th species...")
1866 ; count the number of 3rd and 4th species
1867 (add-cost-cst species-arr gil::IRT_EQ 3 count-3rd)
1868 (add-cost-cst species-arr gil::IRT_EQ 4 count-4th)
1869 ; sum the number of 3rd and 4th species
1870 (gil::g-sum *sp* sum-3rd count-3rd)
1871 (gil::g-sum *sp* sum-4th count-4th)
1872
1873 ; 4th species is only used in third and first position
1874 (gil::g-rel *sp* (second *sp-arr) gil::IRT_NQ 4) ; second position not 4th
      species
1875 (gil::g-rel *sp* (fourth *sp-arr) gil::IRT_NQ 4) ; fourth position not 4th
      species
1876
1877 ; 4th species in third position is followed by a 0 (no note/constraint) and
      then a 4th species
1878 (add-4th-rythmic-cst (third *sp-arr) (fourth *sp-arr) (first *sp-arr))
1879
1880 ; only 3rd and 4th species are used
1881 (gil::g-rel *sp* species-arr gil::IRT_NQ 1) ; not 1st species
1882 (gil::g-rel *sp* species-arr gil::IRT_NQ 2) ; not 2nd species
1883
1884 ; first and penultimate measure are 4th species
1885 ; first measure = [0 0 4 0]
1886 (gil::g-rel *sp* (first (first *sp-arr)) gil::IRT_EQ 0) ; first note is
      silent
1887 (gil::g-rel *sp* (first (second *sp-arr)) gil::IRT_EQ 0) ; second note is
      silent
1888 (gil::g-rel *sp* (first (third *sp-arr)) gil::IRT_EQ 4) ; third note is 4th
      species
1889 ; penultimate measure = [4 0 4 0]
1890 (gil::g-rel *sp* (penult (first *sp-arr)) gil::IRT_EQ 4) ; first note is 4th
      species
1891 (gil::g-rel *sp* (lastone (second *sp-arr)) gil::IRT_EQ 0) ; second note
      does not exist
1892 (gil::g-rel *sp* (lastone (third *sp-arr)) gil::IRT_EQ 4) ; third note is 4
      th species
1893
1894 ; no silence after 3rd species notes
1895 (add-no-silence-cst species-arr)
1896
1897 ; no silence after 4th species notes in n+4 position
1898 (add-no-silence-cst species-arr :spec 4 :offset 4)
1899
1900 ; maximum two consecutive measures without 4th species
1901 (add-min-syncope-cst (third *sp-arr))
1902 )
1903 )
1904

```

```

1905 ; add constraints such that the non-constrained notes have only one possible value
1906 (defun add-one-possible-value-cst (cp is-not-cst-arr)
1907   (loop
1908     for p in cp
1909     for p+1 in (nthcdr 1 cp)
1910     for b-not-cst in is-not-cst-arr
1911     do
1912       (gil::g-rel-reify *sp* p gil::IRT_EQ p+1 b-not-cst gil::RM_IMP) ; TODO the
          value of the note
1913   )
1914 )
1915
1916 ; add constraints such that consecutives syncopations cannot be the same
1917 ; depending on @is-syncope-arr which is true if the note is a syncopation
1918 (defun add-no-same-syncopation-cst (cp-thesis cp-arsis is-syncope-arr)
1919   (loop
1920     for th in (rest cp-thesis)
1921     for ar in (rest cp-arsis)
1922     for b in (rest is-syncope-arr)
1923     do
1924       (gil::g-rel-reify *sp* th gil::IRT_NQ ar b gil::RM_IMP)
1925   )
1926 )
1927
1928 ; add the constraint that the species arrays from two fifth-species parts must be at
          least 50% different
1929 (defun add-make-fifth-species-different-cst (parts)
1930   ; the fifth species attributes to each note a species between 1 and 4. when
          composing with two fifth-species, only half the notes can be of the same
          species at the same time -> if not, there is a lot of redundancy between the
          two fifth-species voices
1931   (let (
1932     (is-same-species (gil::add-bool-var-array *sp* (solution-len (second
          parts)) 0 1))
1933     (is-same-species-int (gil::add-int-var-array *sp* (solution-len (second
          parts)) 0 1))
1934     (percentage-same-species (gil::add-int-var *sp* 0 (solution-len (second
          parts)))))
1935   )
1936   (dotimes (i (solution-len (second parts)))
1937     (gil::g-rel-reify *sp* (nth i (species-arr (second parts))) gil::IRT_EQ
          (nth i (species-arr (third parts))) (nth i is-same-species))
1938     (gil::g-rel-reify *sp* (nth i is-same-species-int) gil::IRT_EQ 1 (nth i
          is-same-species))
1939   )
1940   (gil::g-sum *sp* percentage-same-species is-same-species-int)
1941   (gil::g-rel *sp* percentage-same-species gil::IRT_LE (floor (/ (solution-len
          (second parts)) 2)))
1942 )
1943 )
1944
1945 ; find the next @type note in the borrowed scale,
1946 ; if there is no note in the range then return the note of the other @type
1947 ; - note: integer for the current note
1948 ; - type: atom [lower | higher] for the type of note to find
1949 ; note: this function has nothing to do with GECODE
1950 (defun find-next-note (note type extended-cp-domain)
1951   (let (
1952     ; first sort the scale corresponding to the type
1953     (sorted-scale (if (eq type 'lower)
          (sort extended-cp-domain #>)
1954     (sort extended-cp-domain #<))
1955   ))

```

```

1957 )
1958 (if (eq type 'lower)
1959   ; then we search the first note in the sorted scale that is lower than
        the current note
1960   (progn
1961     (loop for n in sorted-scale do
1962       (if (< n note) (return-from find-next-note n))
1963     )
1964     ; no note so we return the penultimate element of the sorted scale
1965     (penult sorted-scale)
1966   )
1967   ; else we search the first note in the sorted scale that is higher than
        the current note
1968   (progn
1969     (loop for n in sorted-scale do
1970       (if (> n note) (return-from find-next-note n))
1971     )
1972     ; no note so we return the penultimate element of the sorted scale
1973     (penult sorted-scale)
1974   )
1975 )
1976 )
1977 )
1978
1979 ; parse the species array to get the corresponding rythmic pattern for open music
1980 ; - species-arr: array of integer for species (returned by the next-solution
        algorithm)
1981 ; - cp-arr: array of integer for counterpoint notes (returned by the next-solution
        algorithm)
1982 ; note: this function has noting to do with GECODE
1983 (defun parse-species-to-om-rythmic (species-arr cp-arr extended-cp-domain)
1984   ; replace the last element of the species array by 1
1985   (setf (first (last species-arr)) 1)
1986   (build-rythmic-pattern species-arr cp-arr nil nil extended-cp-domain)
1987 )
1988
1989 ; build the rythmic pattern for open music from the species array
1990 ; - species-arr: array of integer for species
1991 ; - cp-arr: array of integer for counterpoint notes
1992 ; - rythmic-arr: array of integer for the rythmic (supposed to be nil and then
        filled by the recursive function)
1993 ; - notes-arr: array of integer for notes (supposed to be nil and then filled by
        the recursive function)
1994 ; - b-debug: boolean to print debug info
1995 ; note: this function has noting to do with GECODE
1996 (defun build-rythmic-pattern (species-arr cp-arr &optional (rythmic-arr nil) (
        notes-arr nil) (extended-cp-domain nil) (b-debug nil))
1997   ; print debug info
1998   (if b-debug
1999     (progn
2000       (print "Current species and notes:")
2001       (print species-arr)
2002       (print cp-arr)
2003       (print "Current answer:")
2004       (print rythmic-arr)
2005       (print notes-arr)
2006     )
2007   )
2008   ; base case
2009   (if (null species-arr)
2010     ; then return the rythmic pattern
2011     (list rythmic-arr notes-arr)
2012   )

```

```

2013
2014 (let (
2015   (sn (first species-arr)) ; current species
2016   (sn+1 (second species-arr)) ; next species
2017   (sn+2 (third species-arr)) ; next next species
2018   (sn+3 (fourth species-arr)) ; next next next species
2019   (cn (first cp-arr)) ; current counterpoint note
2020   (cn+1 (second cp-arr)) ; next counterpoint note
2021   (cn+2 (third cp-arr)) ; next next counterpoint note
2022   (cn+3 (fourth cp-arr)) ; next next next counterpoint note
2023 )
2024 ; replace all nil by -1 for the species
2025 (if (null sn) (setf sn -1))
2026 (if (null sn+1) (setf sn+1 -1))
2027 (if (null sn+2) (setf sn+2 -1))
2028 (if (null sn+3) (setf sn+3 -1))
2029 ; replace all nil by -1 for the counterpoint
2030 (if (null cn) (setf cn -1))
2031 (if (null cn+1) (setf cn+1 -1))
2032 (if (null cn+2) (setf cn+2 -1))
2033 (if (null cn+3) (setf cn+3 -1))
2034
2035 (if b-debug
2036   (progn
2037     (print (format nil "sn: ~a, sn+1: ~a, sn+2: ~a, sn+3: ~a" sn sn+1 sn+2
2038                   sn+3))
2039     (print (format nil "cn: ~a, cn+1: ~a, cn+2: ~a, cn+3: ~a" cn cn+1 cn+2
2040                   cn+3))
2041   )
2042 )
2043 (cond
2044   ; 1 if it is the last note [1 -1 ...]
2045   ((and (eq sn 1) (eq sn+1 -1))
2046     (list (append rhythmic-arr (list 1)) (append notes-arr (list cn)))
2047   )
2048   ; if [4 0 4 ...] -> which syncope ?
2049   ((and (eq sn 4) (eq sn+1 0) (eq sn+2 4))
2050     (if (/= cn cn+2) ; syncopation but different notes ?
2051       ; then same as half note
2052       (if (eq sn+3 3)
2053         ; then 1/2 + 1/4 if [4 0 4 3] (syncopation catch up by a quarter
2054           note)
2055         (build-rhythmic-pattern
2056           (nthcdr 3 species-arr)
2057           (nthcdr 3 cp-arr)
2058           (append rhythmic-arr (list 1/2 1/4))
2059           (append notes-arr (list cn cn+2))
2060           extended-cp-domain
2061         )
2062       ; else 1/2 + 1/2 if [4 0 4 0] (basic syncopation)
2063       (build-rhythmic-pattern
2064         (nthcdr 4 species-arr)
2065         (nthcdr 4 cp-arr)
2066         (append rhythmic-arr (list 1/2 1/2))
2067         (append notes-arr (list cn cn+2))
2068         extended-cp-domain
2069       )
2070     )
2071   ; else same as full note syncopated
2072   (if (eq sn+3 3)
     ; then 3/4 if [4 0 4 3] (syncopation catch up by a quarter note)

```

```

2073         (build-rythmic-pattern
2074           (nthcdr 3 species-arr)
2075           (nthcdr 3 cp-arr)
2076           (append rythmic-arr (list 3/4))
2077           (append notes-arr (list cn))
2078           extended-cp-domain
2079         )
2080       ; else 1 if [4 0 4 0] (basic syncopation)
2081       (build-rythmic-pattern
2082         (nthcdr 4 species-arr)
2083         (nthcdr 4 cp-arr)
2084         (append rythmic-arr (list 1))
2085         (append notes-arr (list cn))
2086         extended-cp-domain
2087       )
2088     )
2089   ))
2090
2091   ; 1/8 note (croche) if cn == cn+1 AND [!0 (3 or 4) ...]
2092   ((and (eq cn cn+1) (/= sn 0) (or (eq sn+1 3) (eq sn+1 4))))
2093     (if (>= (lastone notes-arr) cn)
2094       ; then eighth note with the next lower note
2095       (build-rythmic-pattern
2096         (nthcdr 1 species-arr)
2097         (nthcdr 1 cp-arr)
2098         (append rythmic-arr (list 1/8 1/8))
2099         (append notes-arr (list cn (find-next-note cn 'lower
2100           extended-cp-domain))))
2100       extended-cp-domain
2101     )
2102     ; else eighth note with the next higher note
2103     (build-rythmic-pattern
2104       (nthcdr 1 species-arr)
2105       (nthcdr 1 cp-arr)
2106       (append rythmic-arr (list 1/8 1/8))
2107       (append notes-arr (list cn (find-next-note cn 'higher
2108         extended-cp-domain))))
2109       extended-cp-domain
2110     )
2111   )
2112
2113   ; silence if [0 0 ...]
2114   ((and (eq sn 0) (eq sn+1 0))
2115     (build-rythmic-pattern
2116       (nthcdr 2 species-arr)
2117       (nthcdr 2 cp-arr)
2118       (append rythmic-arr (list -1/2))
2119       notes-arr
2120       extended-cp-domain
2121     )
2122   )
2123
2124   ; 1 if [1 0 0 0] (full note)
2125   ((and (eq sn 1) (eq sn+1 0) (eq sn+2 0) (eq sn+3 0))
2126     (build-rythmic-pattern
2127       (nthcdr 4 species-arr)
2128       (nthcdr 4 cp-arr)
2129       (append rythmic-arr (list 1))
2130       (append notes-arr (list cn))
2131       extended-cp-domain
2132     )
2133   )

```

```

2134
2135 ; 1/2 if [2 0 ...] (half note)
2136 ((and (eq sn 2) (eq sn+1 0))
2137   (build-rythmic-pattern
2138     (nthcdr 2 species-arr)
2139     (nthcdr 2 cp-arr)
2140     (append rhythmic-arr (list 1/2))
2141     (append notes-arr (list cn))
2142     extended-cp-domain
2143   )
2144 )
2145
2146 ; 1/4 if [3 ...] (quarter note)
2147 ((eq sn 3)
2148   (build-rythmic-pattern
2149     (nthcdr 1 species-arr)
2150     (nthcdr 1 cp-arr)
2151     (append rhythmic-arr (list 1/4))
2152     (append notes-arr (list cn))
2153     extended-cp-domain
2154   )
2155 )
2156
2157 ; 1/2 if [4 0 1 ...] (penultimate note for the 4th species)
2158 ((and (eq sn 4) (eq sn+1 0) (eq sn+2 1))
2159   (build-rythmic-pattern
2160     (nthcdr 2 species-arr)
2161     (nthcdr 2 cp-arr)
2162     (append rhythmic-arr (list 1/2))
2163     (append notes-arr (list cn))
2164     extended-cp-domain
2165   )
2166 )
2167 )
2168 )
2169 )
2170
2171 ; get the basic rhythmic pattern and the corresponding notes the given species
2172 ; - species-list: the species [1 2 3 4]
2173 ; - len: the length of the cantus-firmus
2174 ; - sol-pitches: the whole solution array
2175 ; - counterpoints: the counterpoints we are working with (only useful for the 5th
2176 ;   species as it needs the extended-cp-domain)
2177 ; - sol: the solutino space (only useful for the 5th species)
2178 ; return format = '( (rhythmic-1 pitches-1) (rhythmic-2 pitches-2) ... (rhythmic-n
2179 ;   pitches-n) )
2180 ; examples:
2181 ; ((1) 5) -> (((1 1 1 1 1) (60 62 64 65 60)))
2182 ; ((1 2) 5) -> (((1 1 1 1 1) (60 62 64 65 60)) ((1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1)
2183 ;   (60 62 64 65 64 62 60 62 60)))
2184 ; ((2) 5) -> ((1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1 (pitches))
2185 ; ((3) 5) -> ((1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1 (
2186 ;   pitches))
2187 ; ((4) 5) -> ~((-1/2 1 1 1 1/2 1/2 1 (pitches)) depending on the counterpoint
2188
2189 (defun get-basic-rhythmics (species-list len sol-pitches counterpoints sol)
2190   (setq len-1 (- len 1))
2191   (setq len-2 (- len 2))
2192   (let (
2193     (rhythmic+pitches (make-list *N-COUNTERPOINTS :initial-element nil))
2194   )
2195     (loop for i from 0 below *N-COUNTERPOINTS do (progn
2196       (case (nth i species-list)

```

```

2193 (1 (progn
2194   (setf (nth i rhythmic+pitches) (list
2195     ; rhythm
2196     (make-list len :initial-element 1)
2197     ; pitches
2198     (subseq sol-pitches 0 len)
2199   ))
2200   (setf sol-pitches (subseq sol-pitches len))
2201 )
2202 (2 (let (
2203   (rhythmic (append (make-list (* 2 len-1) :initial-element
2204     1/2) '(1)))
2205   (pitches (subseq sol-pitches 0 (- (* 2 len) 1)))
2206 )
2207   (if (eq (car (last pitches 4)) (car (last pitches 3))) (
2208     progn ; if the first note in the penult bar is the same
2209     as the last in the 2nd-to last
2210     ; then ligature them
2211     (setf rhythmic (append (butlast rhythmic 4) '(1) (last
2212       rhythmic 2)))
2213     (loop
2214       for i from (- (length pitches) 4) below (- (length
2215         pitches) 1)
2216       do (setf (nth i pitches) (nth (+ i 1) pitches))
2217     )
2218   ))
2219   (if (eq (car (last pitches 3)) (car (last pitches 2))) (
2220     progn ; same but for 3rd-to-last and 2nd-to-last
2221     (setf rhythmic (append (butlast rhythmic 3) '(1) (last
2222       rhythmic 1)))
2223     (loop
2224       for i from (- (length pitches) 3) below (- (length
2225         pitches) 1)
2226       do (setf (nth i pitches) (nth (+ i 1) pitches))
2227     )
2228   ))
2229   (setf (nth i rhythmic+pitches) (list
2230     rhythmic
2231     pitches
2232   ))
2233   ; remove all the notes we've just considered from
2234   sol-pitches
2235   (setf sol-pitches (subseq sol-pitches (length pitches)))
2236 )
2237 )
2238 (3 (progn
2239   (setf (nth i rhythmic+pitches) (list
2240     ; rhythm
2241     (append (make-list (* 4 len-1) :initial-element 1/4) '(1))
2242     ; pitches
2243     (subseq sol-pitches 0 (- (* 4 len) 3))
2244   ))
2245   ; remove all the notes we've just considered from sol-pitches
2246   (setf sol-pitches (subseq sol-pitches (- (* 4 len) 3)))
2247 )
2248 (4 (progn
2249   (setf (nth i rhythmic+pitches) (build-rhythmic-pattern
2250     (get-4th-species-array len-2)
2251     (get-4th-notes-array (subseq sol-pitches 0 (* 2 len-1)) (+
2252       (* 4 len-1) 1))
2253   ))
2254   (setf j 0)
2255   (dotimes (k *cf-penult-index)

```

```

2246         (setf j (+ j 2))
2247         ; if we have a note that creates a hidden fifth (direct
           motion to a fifth), then remove the note
2248         (if (and
2249             (eq 7 (nth (+ 1 j) (gil::g-values sol (first (
               h-intervals (nth i counterpoints))))))
2250             (eq DIRECT (nth j (gil::g-values sol (first (motions (
               nth i counterpoints))))))
2251             )
2252             (setf (nth j (first (nth i rhythmic+pitchess))) -1/2)
2253         )
2254     )
2255     (setf sol-pitches (subseq sol-pitches (* 2 len-1)))
2256 ))
2257 (5 (let (
2258     (sol-species (gil::g-values sol (species-arr (nth i
       counterpoints)))) ; store the values of the solution
2259 )
2260     (setf (nth i rhythmic+pitchess)
2261         (parse-species-to-om-rhythmic sol-species sol-pitches (
           extended-cp-domain (nth i counterpoints)))
2262     )
2263     (setf sol-pitches (subseq sol-pitches (solution-len (nth i
       counterpoints))))
2264 ))
2265 )
2266 ))
2267 (assert (eql sol-pitches nil) (sol-pitches) "Assertion failed: sol-pitches
       should be nil at the end of function get-basic-rhythmics.")
2268 rhythmic+pitchess
2269 )
2270 )
2271
2272 ; return a species array for a 4th species counterpoint
2273 ; - len-2: the length of the counterpoint - 2
2274 (defun get-4th-species-array (len-2)
2275     (append (list 0 0) (get-n-4040 len-2) (list 4 0 1))
2276 )
2277
2278 ; return a note array for a 4th species counterpoint
2279 ; - len: the length of the cantus firmus
2280 (defun get-4th-notes-array (cp len)
2281     (let* (
2282         (notes (make-list len :initial-element 0)) ; notes that we don't care about
           can be 0
2283     )
2284         (loop
2285             for n from 2 below len by 2 ; we move from 4 to 4 (4 0 4 ...) after the
           silence (0 0) at the start
2286             for p in cp
2287             do
2288                 (setf (nth n notes) p)
2289         )
2290         notes
2291     )
2292 )
2293
2294 ; return a list with n * (4 0 4 0), used to build the rhythmic pattern for the 4th
       species
2295 ; - n: the number of times the pattern is repeated
2296 (defun get-n-4040 (n)
2297     (if (eq n 0)
2298         nil

```

```

2299         (append (list 4 0 4 0) (get-n-4040 (- n 1)))
2300     )
2301 )
2302
2303 ; return the tone offset of the voice
2304 ; => [0, ..., 11]
2305 ; 0 = C, 1 = C#, 2 = D, 3 = D#, 4 = E, 5 = F, 6 = F#, 7 = G, 8 = G#, 9 = A, 10 = A#,
      11 = B
2306 (defun get-tone-offset (voice)
2307     (let (
2308         (tone (om::tonalite voice))
2309     )
2310         (if (eq tone nil)
2311             ; then default to C major
2312             0
2313             ; else check if the mode is major or minor
2314             (let (
2315                 (mode (om::mode tone))
2316             )
2317                 (if (eq (third mode) 300)
2318                     (midicent-to-midi-offset (+ (om::tonmidi tone) 300))
2319                     (midicent-to-midi-offset (om::tonmidi tone))
2320                 )
2321             )
2322         )
2323     )
2324 )
2325
2326 ; converts a midicent value to the corresponding offset midi value
2327 ; note:[0, 12700] -> [0, 11]
2328 ; 0 corresponds to C, 11 to B
2329 (defun midicent-to-midi-offset (note)
2330     (print (list "midicent-to-midi-offset..." note))
2331     (mod (/ note 100) 12)
2332 )
2333
2334 ; return the absolute difference between two midi notes modulo 12
2335 ; or the brut interval if b is true
2336 (defun inter (n1 n2 &optional (b nil))
2337     (if b
2338         (- n1 n2)
2339         (mod (abs (- n1 n2)) 12)
2340     )
2341 )
2342
2343 ; add constraint in sp such that the interval between the two notes is a member of
      interval-set
2344 (defun inter-member-cst (sp n1-var n2-val interval-set)
2345     (let (
2346         (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2-val)) ; t1 = n1 - n2
2347         (t2 (gil::add-int-var sp 0 127)) ; used to store the absolute value of t1
2348         note-inter
2349     )
2350         (gil::g-abs sp t1 t2) ; t2 = |t1|
2351         (setq note-inter (gil::add-int-var-expr sp t1 gil::IOP_MOD 12)) ; note-inter
          = t1 % 12
2352         (gil::g-member sp interval-set note-inter) ; note-inter in interval-set
2353     )
2354 )
2355
2356 ; add constraint such that n3-var = |n1-var - n2-val| % 12
2357 (defun inter-eq-cst (sp n1-var n2-val n3-var)
2358     (let (

```

```

2359     (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2-val)) ; t1 = n1 - n2
2360     (t2 (gil::add-int-var sp 0 127)) ; used to store the absolute value of t1
2361     (modulo (gil::add-int-var-dom sp '(12))) ; the IntVar just used to store 12
2362   )
2363   (gil::g-abs sp t1 t2) ; t2 = |t1|
2364   (gil::g-mod sp t2 modulo n3-var) ; n3-var = t2 % 12
2365 )
2366 )
2367
2368 ; add constraint such that
2369 ; brut-var = n1-var - n2
2370 ; abs-var = |brut-var|
2371 (defun inter-eq-cst-brut (sp n1-var n2 brut-var abs-var)
2372   (let (
2373     (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2)) ; t1 = n1-var - n2
2374   )
2375     (gil::g-rel sp t1 gil::IRT_EQ brut-var) ; t1 = brut-var
2376     (gil::g-abs sp t1 abs-var) ; abs-var = |t1|
2377   )
2378 )
2379
2380 ; add constraint such that
2381 ; brut-var = n1-var - n2
2382 ; abs-var = |brut-var|
2383 (defun inter-eq-cst-brut-for-cst (sp n1-var n2 brut-var abs-var is-cst)
2384   (let (
2385     (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2)) ; t1 = n1-var - n2
2386     (t2 (gil::add-int-var sp 0 12)) ; store the absolute value of t1
2387   )
2388     (gil::g-abs sp t1 t2) ; t2 = |t1|
2389     (gil::g-ite sp is-cst t1 ZERO brut-var) ; brut-var = t1 if is-cst, else
2390       brut-var = 0
2391     (gil::g-ite sp is-cst t2 ZERO abs-var) ; abs-var = t2 if is-cst, else
2392       abs-var = 0
2393   )
2394 )
2395
2396 ; return the last element of a list
2397 (defun lastone (l)
2398   (first (last l))
2399 )
2400
2401 ; return the rest of a list without its last element
2402 (defun restbutlast (l)
2403   (butlast (rest l))
2404 )
2405
2406 ; return the penultimate element of a list
2407 (defun penult (l)
2408   (nth (- (length l) 2) l)
2409 )
2410
2411 ; return an approximative checksum of pitches associated to a rhythmic
2412 ; - p: the list of pitches
2413 ; - r: the list of rhythmic values (with the -1/2 at the beginning)
2414 (defun checksum-sol (p r)
2415   (let (
2416     (l (length p))
2417   )
2418     (mod (floor (reduce #'+
2419       (mapcar #'* (range (+ l 5) :min 5) (rest r) p)))
2420       (expt l 12))
2421   )

```

```

2420 )
2421
2422 ; add the sum of the @factor-arr as a cost to the *cost-factors array and increment
      *n-cost-added
2423 ; additionally, keeps track of the index it was added to
2424 ; @g-sum: t if we need to sum the factor-arr, false if not
2425 (defun add-cost-to-factors (factor-arr cost-name &optional (g-sum 1))
2426   (assert (< *n-cost-added *N-COST-FACTORS) (*n-cost-added) "Assertion failed:
      Trying to set more costs than what has been defined (~A). Please increase
      the value of *N-COST-FACTORS." *N-COST-FACTORS)
2427   (if g-sum
2428     (gil::g-sum *sp* (nth *n-cost-added *cost-factors) factor-arr)
2429     (setf (nth *n-cost-added *cost-factors) factor-arr)
2430   )
2431   ; store the index of the cost in the cost-index hashmap
2432   (setf (gethash cost-name *cost-indexes) (append (gethash cost-name *cost-indexes)
      (list *n-cost-added)))
2433   (incf *n-cost-added)
2434 )

```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl