GENERALIZED MULTIPLE ORDER-NUMBER FUNCTION ARRAYS

By

LUIS FELIPE VIEIRA DAMIANI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2022

To my family

ACKNOWLEDGMENTS

Acknowledgments go here.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS 4

LIST OF TABLES 7

LIST OF FIGURES 8

CHAPTER

ABSTRACT 9

1 INTRODUCTION 10

1.1 Basic Derivation Theory 11
1.1.1 Aggregate Realizations 15
1.1.2 Row Segments 23
1.2 A Survey of Derivation Techniques 27
1.2.1 Folded Derivation 30
1.2.2 Shifted Derivation 32
1.2.3 Self-Derivation 33
1.2.4 Derivation from Aggregate Realizations 38
1.3 The Mallalieu Property 45
1.4 Final Remarks 49

2 LITERATURE REVIEW 52

2.1 Discovering Derivation 52

3 THEORETICAL FRAMEWORK 56

3.1 Group Actions 56
3.2 Polya's Enumeration Formula 60

4 THEORETICAL FRAMEWORK 66

4.1 Defining an algorithm to compute a row class 66
4.1.1 The canonical form of a row 67
4.1.2 Storing a row class in memory 73
4.2 Defining an algorithm to compute semi-magical squares 78
4.2.1 Computing the partitions of an integer 79
4.2.2 Combining partition rows into squares 84
4.3 Computing all possible combination matrices for a row 89
4.3.1 Computing the side and top rows of a combination matrix 91
4.3.2 Writing a solution as text 95
4.3.3 Computing all possible solutions for a given combination matrix 102
4.4 Creating work data for use in concurrent threads 108
4.5 Defining a top-level procedure for computing all solutions for a given row class 114

4.5.1 Directions for the future 125

APPENDIX

A 126

A.1 ELEMENTARY TWELVE-TONE THEORY 126

B ELEMENTARY GROUP THEORY 127

BIOGRAPHICAL SKETCH 130

LIST OF TABLES

Table		page
1-1	Rahn's Common Tones Under Inversion	25
1-2	Derivation Involving the Retrograde and an Arbitrary Operation	28
1-3	Folded Derivation Involving the Retrograde	30
1-4	Shifted Derivation Involving the Retrograde	32
1-5	Self-Derivation Involving the Retrograde	34
2-1	Martino's Source Hexachords	52

LIST OF FIGURES

Figure page

1-1 Berg's *Lulu* 10

1-2 Derived Row in Berg's *Lulu* 11

1-3 Bars 1–7 in Webern's Op. 27 16

1-4 Aggregate Realization of Bars 1–7 in Webern's Op. 27 17

1-5 Another Aggregate Realization of Bars 1–7 in Webern's Op. 27 17

1-6 Bars 8–10 in Webern's Op. 27 18

1-7 An Aggregate Realization of Bars 8–10 in Webern's Op. 27 18

1-8 Bars 19–23 in Webern's Op. 27 19

1-9 An Aggregate Realization of Bars 19–23 in Webern's Op. 27 19

1-10 Intersecting Bars 1–7 with Bars 8–10 in Webern's Op. 27 20

1-11 Intersecting Bars 1–7 with Bars 19–23 in Webern's Op. 27 20

1-12 Intersecting Bars 8–10 with Bars 19–23 in Webern's Op. 27 20

1-13 The intersection of Bars 1–7, 8–10, and 19–23 in Webern's Op. 27 21

1-14 Self-derived row in Scotto's *Tetralogy* 35

1-15 Folded derivation in Scotto's *Tetralogy* 36

1-16 Schenkerian middle-ground structure in Scotto's *Tetralogy* 36

1-17 Musical realization of the middle-ground in Scotto's *Tetralogy* 37

1-18 Prolongation of the middle-ground structure in Scotto's *Tetralogy* 37

1-19 Musical realization of the prolonged middle-ground in Scotto's *Tetralogy* 37

1-20 Self-derivation in Damiani's *Stingray*. 45

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

GENERALIZED MULTIPLE ORDER-NUMBER FUNCTION ARRAYS

By

Luis Felipe Vieira Damiani

December 2022

Chair: Dr. James P. Sain
Major: Music Composition

The purpose of the present study is the application of self-derivation matrices to the algorithmic composition of acoustic and electroacoustic music. It departs from the seminal work of Daniel Starr in the field of twelve-tone theory, published by Perspectives of New Music in 1984, generalizing it to arbitrary equal-temperament systems. A solid mathematical foundation that describes the derivation of multiple order-number function arrays is a major contribution to the fields of music composition and music theory, as well as a requirement for the implementation of non-brute-force algorithms. We confront our findings with those of David Kowalski, and verify the latter for accuracy. The results of this research include a complete classification of rows that support self-derivation in matrices of all sizes, as well as a complete classification of the algebraic operations that afford such procedures. In addition, several general-purpose and non-brute-force algorithms are provided that implement our findings in the programming languages that pertain to musicians the most. A myriad of musical compositions are specially devised with the intent of illustrating the real-life use of such algorithms, and to provide criticism as to where such compositional practices situate within the realm of 21st-century music composition.

CHAPTER 1
INTRODUCTION

Music has become an almost arbitrary matter, and composers will no longer be bound by laws and rules, but avoid the names of School and Law as they would Death itself...

– Johann Joseph Fux

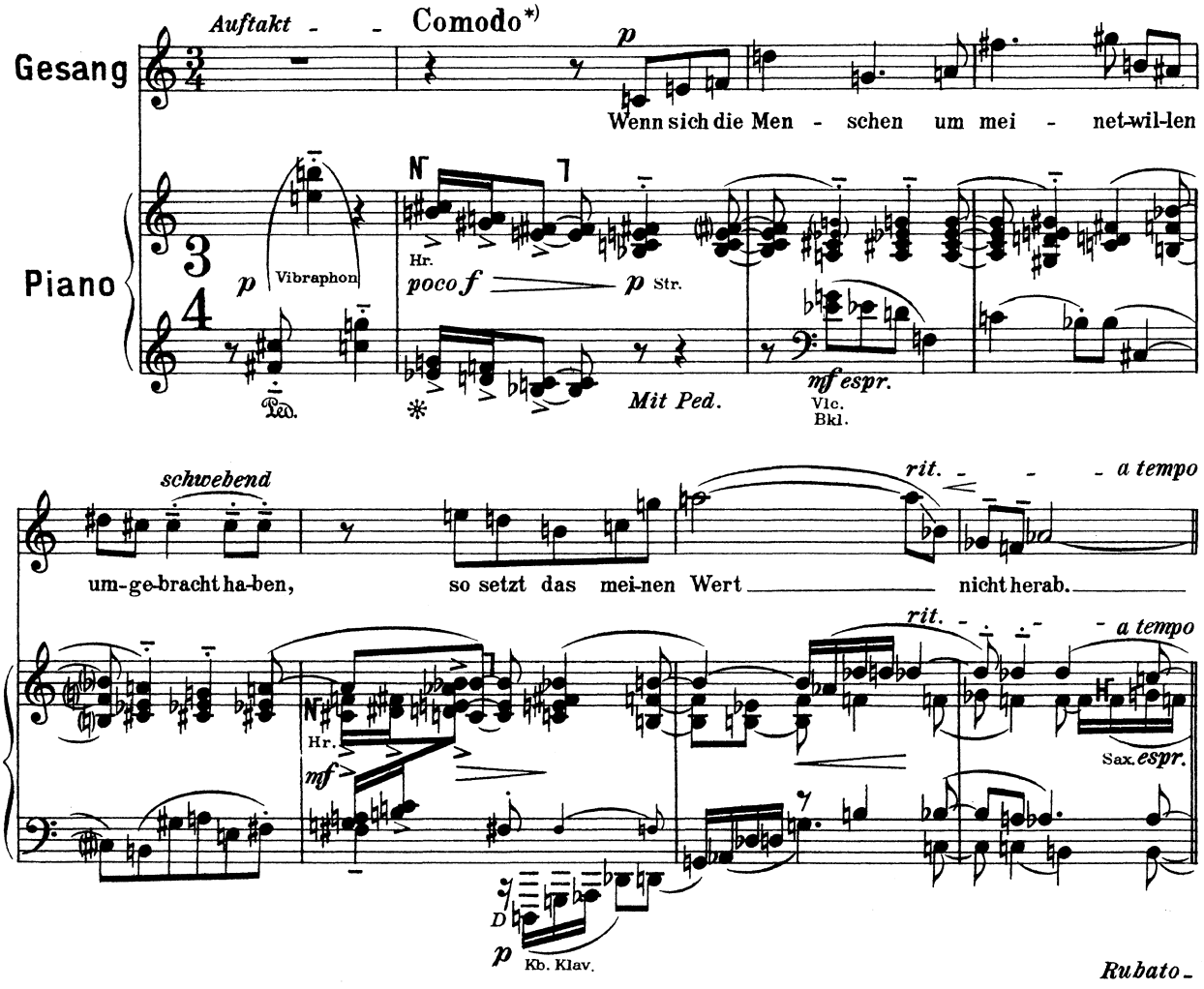


Figure 1-1. Alban Berg's *Lulu* [1, 182].

1.1 Basic Derivation Theory

Derivation is the process of extracting ordered segments from rows in order to generate new compositional materials. It is a technique that dates back to the Second Viennese School. In its most incipient form, a composer may simply extract these segments from a row, and combine them to form another, as we see in Berg's *Lulu*. The basic row used by Berg is $S = \{10, 2, 3, 0, 5, 7, 4, 6, 9, 8, 1, 11\}$. In the Prologue, however, one is greeted with the row $\{10, 3, 4, 9, 2, 7, 8, 1, 0, 5, 6, 11\}$, as depicted in Fig. 1-2. It is clear that the segments that constitute the Prologue's row are ordered segments in the basic row form, and the fact that one row cannot obtained from another via row operations is here irrelevant.

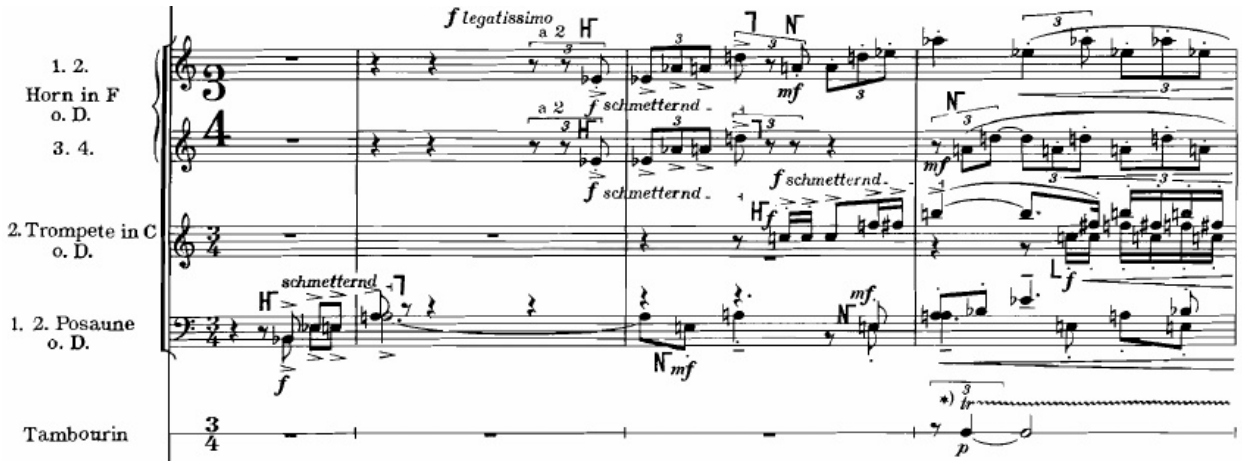


Figure 1-2. Derived row in the *Prologue* of Alban Berg's *Lulu* [1, 182].

Less naive approaches to derivation often involve a derived row that will have more structure. In them, one will usually see a combination matrix where derived and original rows are matched with some twelve-tone or order operation of themselves, or both. This is illustrated in Ex. 1.1.1.

Example 1.1.1. *We may use the basic row in* Lulu *as motivation for a basic derivation procedure. The first step is to create a* 2×24 *array where the first row is* S *followed by* $\mathrm{R}(S)$,

and the second row is initially undefined.

$$\left[\begin{array}{cccccccccccc|cccccccccccc} 10 & 2 & 3 & 0 & 5 & 7 & 4 & 6 & 9 & 8 & 1 & 11 & 11 & 1 & 8 & 9 & 6 & 4 & 7 & 5 & 0 & 3 & 2 & 10 \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right]. \tag{1–1}$$

Next, one chooses an arbitrary segment, and separates it from the the top row by placing in in the bottom row:

$$\left[\begin{array}{cccccccccccc|cccccccccccc} . & 2 & 3 & . & . & . & 4 & 6 & 9 & 8 & 1 & 11 & . & . & . & . & . & . & 7 & 5 & 0 & . & . & 10 \\ 10 & . & . & 0 & 5 & 7 & . & . & . & . & . & . & 11 & 1 & 8 & 9 & 6 & 4 & . & . & . & 3 & 2 & . \end{array}\right]. \tag{1–2}$$

Let $V = \{2, 3, 4, 6, 9, 8, 1, 11, 7, 5, 0, 10\}$*. Then* V *is a row derived from* S*. In particular, the ordered segment* $\{10, 0, 5, 7\}$ *in* S *is preserved by* $\mathrm{R}(V)$*, and a counterpoint that combines* S *vertically with* V *horizontally is possible by construction.*

It is of interest to note at this point that, in this kind of construction, the choice of a particular segment is already an important compositional decision. This choice bears relevance in that it establishes motivic material, that is, the segment itself. It also potentially introduces complementary harmonic regions, one given by the segment, the other given by its set complement. Moreover, and perhaps more importantly, it presents an opportunity for exploring syntax. There are a multitude of ways in which a composer may obtain syntax from a simple derivation procedure such as the one given in the example above. One way would be to find an operation that makes the chosen segment invariant. In particular, it is easily checked that $S_1 = \{10, 0, 5, 7\} = \mathrm{R}\,\mathrm{T}_5\,\mathrm{I}(S_1)$. One can then extend Ex. 1.1.1 into the combination array $[S \mid \mathrm{R}(S) \mid \mathrm{R}\,\mathrm{T}_5\,\mathrm{I}(S) \mid \mathrm{T}_5\,\mathrm{I}(S_1)]$. In the extended array, the segment S_1 would be preserved, but the row derived from $\mathrm{R}\,\mathrm{T}_5\,\mathrm{I}(S)$ would not be a transform of V. If the set complement of S_1 in V were parsed to produce more than one harmonic region, then the complement of S_1 under this new derived row would produce different harmonic regions. This can be very pertinent compositionally, as one would be capable of producing contrasting harmonic regions while maintaining motivic coherence under the S_1 segment.

Yet another way of generating syntax from derivation would be to follow S with V itself. One would then derive a new row from V, say Q, and eventually follow V with Q. Repeating this procedure *ad libitum* could generate many contrasting harmonic regions. In particular, this type of derivation is seen in Donald Martino's *Notturno* of 1974, a composition that won the Pulitzer Prize in the following year [1, 181]. If, by compositional choice, the chain of derived rows picked always the same order numbers, then a potential for rhythmic and agogic coherence could also be explored.

In one of the seminal academic works in the field of twelve-tone theory, [1] utilizes a mostly set-theoretic framework to understand and categorize rows and procedures involved in producing derivation, polyphony, and self-derived combination matrices. The main objective is similar to ours, in it has a bias toward unveiling self-derivation, which unfortunately still remains a somewhat obscure topic. The set-theoretic approach revolves around the idea of looking at collections from the standpoint of their order constraints: a totally constrained set with no precedence contradictions is a twelve-tone row; a completely unconstrained set of twelve tones represents the free aggregate; a maximally constrained one is what the author calls the simultaneous aggregate, that is, a twelve-tone cluster. Sets that live in-between can often be projected in the middle and background of a composition, fact that amounts to a Schenkerian-flavored view of the whole process.

Mathematically, the ideas in [1] translate into considering the set U of all ordered pairs of pitch classes. There are twelve choices for the first position, and twelve choices for the second position. As both choices are independent, this set has cardinality $12^2 = 144$. An element of U is called an order constraint, and a subset C of U is called a pitch-class relation. The latter can be viewed as a 12×12 matrix where the entry c_{ij} is equal to one whenever $\{i, j\} \in C$, and zero otherwise. One can then apply bitwise operations to these matrices in a very computationally efficient manner: bitwise *and* and *or* correspond respectively to set intersection and union. For any pair of pitch classes x and y, we define a relation $x \sim y$ on the power set of U by the set inclusion of the element $\{x, y\}$. A subset C will then be reflexive

if, whenever an element of C (which is a set) contains the pitch class x, then $\{x, x\} \in C$. In words, reflexivity means that if a reflexive collection C of notes contains an element x, then x precedes (and follows) itself in C. The free aggregate is a minimal reflexive subset of U that contains all twelve tones. The relation $\sim$ will be symmetric if $\{x, y\} \in C$ implies $\{y, x\} \in C$, and antisymmetric whenever $\{x, y\} \in C$ implies $\{y, x\} \notin C$, for $x \neq y \in \mathbb{Z}/12\mathbb{Z}$. Similarly, transitivity is defined as $\{x, y\} \in C$ and $\{y, z\} \in C$, then $\{x, z\} \in C$; and trichotomy is defined as either $\{x, y\} \in C$ or $\{y, x\} \in C$ for any $x \neq y \in \mathbb{Z}/12\mathbb{Z}$. The relation $\sim$ is, of course, an order relation on the set of twelve tones by definition. A partial order is one that is reflexive, transitive, and antisymmetric, while a total order (a row), is a partial order that satisfies trichotomy.

Often, pitch-class relations will contain many redundancies due to transitivity. In order to express these relations as oriented graphs, one must first remove, or prune, such redundancies. This process can be reversed and a pitch-class relation can be extended to the point of its transitive closure. It is also common for a pitch-class relation to be absent of any order constraint involving both $\{x, y\}$, in which case we say x and y are incomparable. Such x and y are bound to be struck together, or else be *linearized* by the injection of some constraint that will make them comparable, as long as there remains still a partial order, that is, as long as this process does not introduce a symmetry, for instance. The set of all total orderings that can be linearized out of some partial order is called its total order class. In a completely analogous manner, one can *verticalize* a pitch-class relation by removing constraints, and again minding that the result is still transitive and symmetric. We say a partial order covers another whenever the former is a verticalization of the latter. A simple procedure to guarantee that a verticalization will remain a partial order is to take its union with the free aggregate, then subject this union to an extension operation, thus providing reflexivity in the first step, as well as transitivity in the second. We can say the following about covering, and about unions and intersections of pitch-class relations:

Theorem 1.1.2. *[1, 193]*

i. Covering is transitive;

ii. A pitch-class relation is covered by its extension;

iii. If a pitch-class relation covers another, then the extension of the former covers the extension of the latter.

Theorem 1.1.3. *[1, 194] Let A and B be partial orders and denote by $\mathrm{Toc}(A)$ and $\mathrm{Toc}(B)$ their respective total order classes. Then*

$$\mathrm{Toc}(A) \cap \mathrm{Toc}(B) = \mathrm{Toc}(\mathrm{Ext}(A \cup B)) \ , \tag{1–3}$$

where Ext *is the extension operator. Moreover, if A_i is a finite sequence of n partial orders, then*

$$\bigcap_{i=0}^{n} \mathrm{Toc}(A_i) = \mathrm{Toc}\left[\mathrm{Ext}\left(\bigcup_{i=0}^{n} A_i\right)\right] \ . \tag{1–4}$$

Theorem 1.1.4. *[1, 194] The intersection of two partial orders is again a partial order.*

Th. 1.1.3 will play a crucial role in the development of derivation techniques. Of vital importance is the fact that one can operate on pitch-class relations the same way one operates on pitch-class sets:

Theorem 1.1.5. *[1, 195] Let C be a pitch-class relation and $\{a, b\}$ an element of U such that $\{a, b\} \in C$.*

i. If F is a pitch-class operation, then $\{F(a), F(b)\} \in F(C)$ if and only if $\{a, b\} \in C$. In particular, if $\mathrm{R}(C)$ is the retrograde of C, then $\{a, b\} \in \mathrm{R}(C)$ if and only if $\{b, a\} \in C$.

ii. If C is totally ordered, then $\mathrm{R}(C) = (S \setminus D) \cup F$, where S is the simultaneous aggregate and F is the free aggregate.

iii. If C_1 covers C_2, then $F(C_1)$ covers $F(C_2)$.

iv. Finally, if C is F R-invariant, then all cycles in F have length two.

1.1.1 Aggregate Realizations

In this section we will define the important concept of *aggregate realizations*, which will, in turn, lead to a classification of partial orders, as well as to many musical applications. An aggregate realization is a particular type of partial order C in which, for any pair of pitch

classes a, b, if a and b are incomparable in C, then the set of pitch classes that precede a in C is equal to the set of pitch classes that precede b in C, and also the set of pitch classes that follow a in C is equal to the set of pitch classes that follow b in C. Aggregate realizations arise naturally from a total order in the sense that they belong to the set of all partial orders that are covered by said total order. An interesting compositional application of aggregate realizations is that of projecting a total order as a middle-ground entity.

Example 1.1.6. *[1, 197] Given a sequence S of partial orders, all of which covered by the same total order, say X, if both X is never stated in the foreground and S contains all order constraints in X, then a musical passage in which S is stated in the foreground will bear X as a middle-ground entity. In the case where S does not comprise all the order constraints in X, some other partial order that covers S will be projected in the middle-ground. In the particular case where a composer is dealing with pitch classes, projecting a partial order is equivalent to inducing the listener to infer its order constraints. If, in this case, two pitch classes are incomparable, then they are bound to be struck together.*

Example 1.1.7. *[1, 207] This examples provides a strategy to unravel the basic total order in Webern's Op. 27. It is somewhat paradoxical in the sense that unveiling the row requires prior knowledge of it. Consider the opening bars in Webern's Op. 27 depicted in Fig. 1-3, as well as their aggregate realization given in Fig. 1-4.*

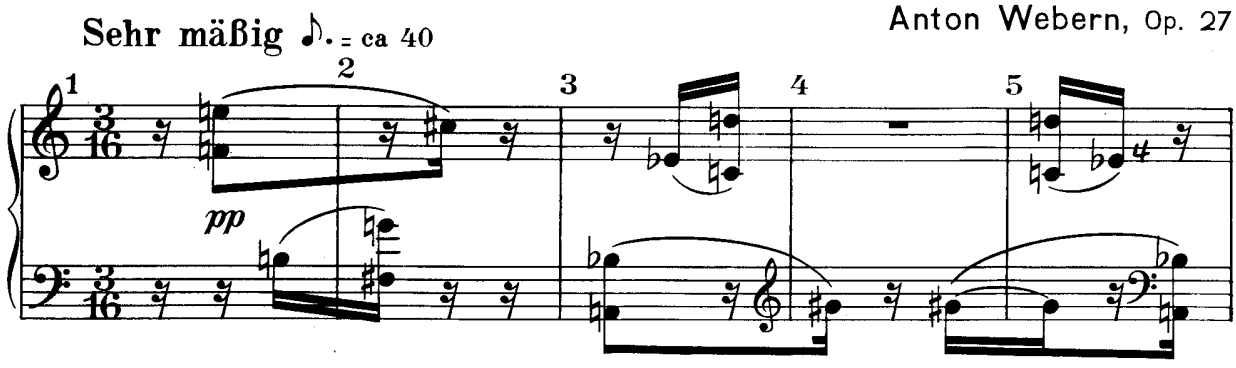


Figure 1-3. The initial bars in Webern's Op. 27.

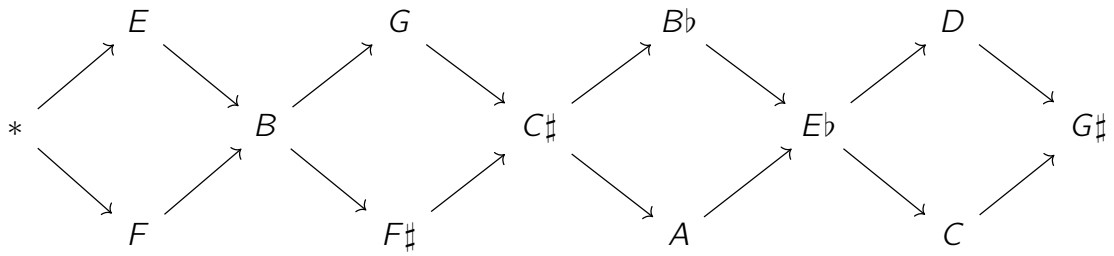


Figure 1-4. An aggregate realization of the initial bars in Webern's Op. 27. At its very first presentation, the total order used to generate the piece cannot be discerned. Moreover, the partial order we are actually able to hear intercalates the total order's two hexachords, a procedure that can be construed as a form of derivation.

Instead of the aggregate realization in Fig. 1-4, however, we use the fact that we know the basic series is $S = \{4, 5, 1, 3, 0, 2, 8, 9, 10, 6, 7, 11\}$*, take the initial bars seen in Fig. 1-3, and rewrite an aggregate realization, call it* D_1*, of them as in Fig. 1-5.*

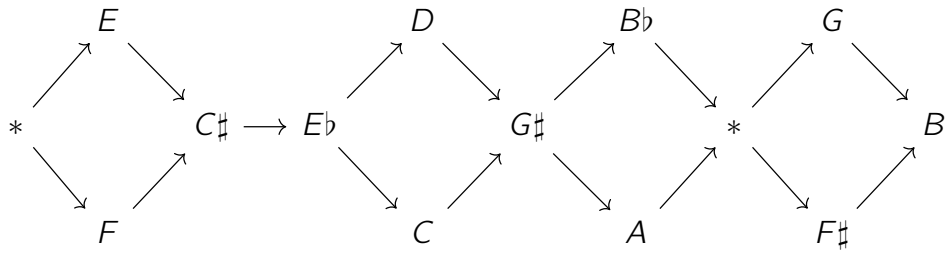


Figure 1-5. We denote the underlying partial order depicted in this aggregate realization by D_1.

It is not too far-fetched to assume such an analysis. Whereas Fig. 1-4 represented a first-time hearing depiction of bars 1–7, Fig. 1-5 can be achieved by, say, a performer who realizes the voice-crossing of the series along its reflection axis. Having come to this conclusion, parsing bars 8–10 becomes a bit less daunting. Fig. 1-6 shows bars 8–10 in Op. 27, and Fig. 1-7 is a normalized aggregate realization of the passage. By normalized we mean that the series being displayed in the music is $\mathrm{T}_{10}\,\mathrm{I}(S) = \{6, 5, 9, 7, 10, 8, 2, 1, 0, 4, 3, 11\}$*, but instead we set the aggregate realization to* S*, since we will want to take intersections of both partial orders later.* $\mathrm{T}_{10}\,\mathrm{I}(S)$ *begins with the right hand in bar 8, moves to the left hand in bar 9, and the very last note (which is not showing) is a high B the right hand again has in bar 11. At this point, our confidence that a series can be heard by the listener is severely damaged. We even start to*

doubt that the average performer will have obtained, or even cared to obtain, a good grasp on what the series really is.

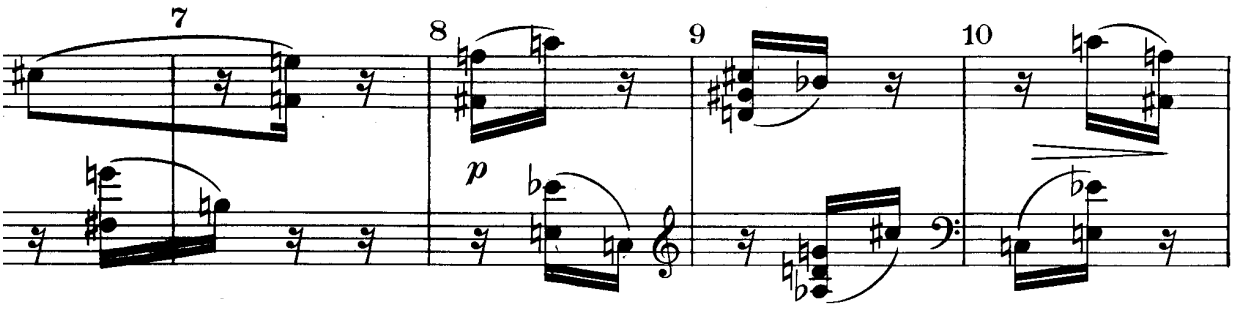


Figure 1-6. Bars 8–10 in Webern's Op. 27.

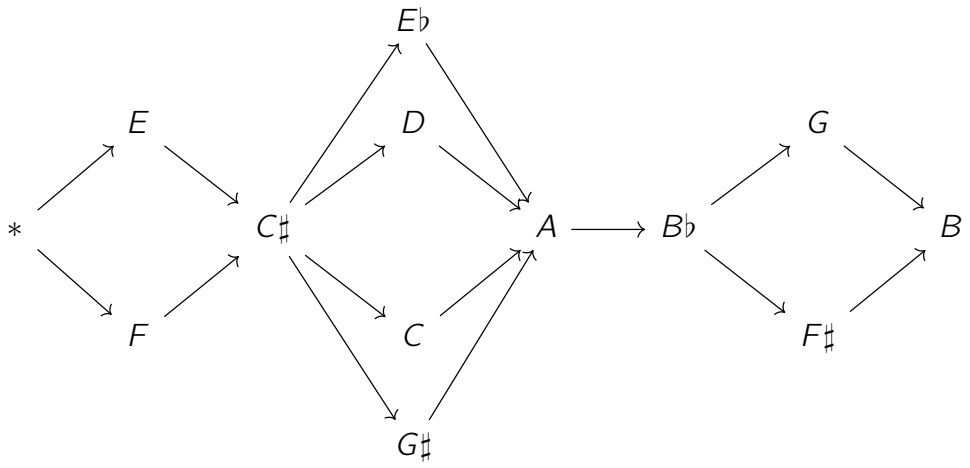


Figure 1-7. We denote the underlying partial order depicted in this aggregate realization by D_2.

The next excerpt we analyze in the first movement is the last one we need in order to infer what series Webern used for the piece. The musical passage, taken from bars 19–23, is given in Fig. 1-8, and an aggregate realization is displayed in Fig. 1-9. We again normalize it with respect to S. This time, the series we are looking for in the excerpt is $\mathsf{T}_3\,\mathsf{I}(S) = \{11, 10, 2, 0, 3, 1, 7, 6, 5, 9, 8, 4\}$*, and again it has a very convoluted presentation, floating around in register, and freely changing hands. We come to the realization that, without the score, it is unlikely that a listener, even a very educated one, will be able to discern the composer's main generative material in one hearing. That well-versed listener might not be*

able to discern it in the hundredth hearing, we even suspect. We shall conclude the analysis before coming back to this discussion, however.

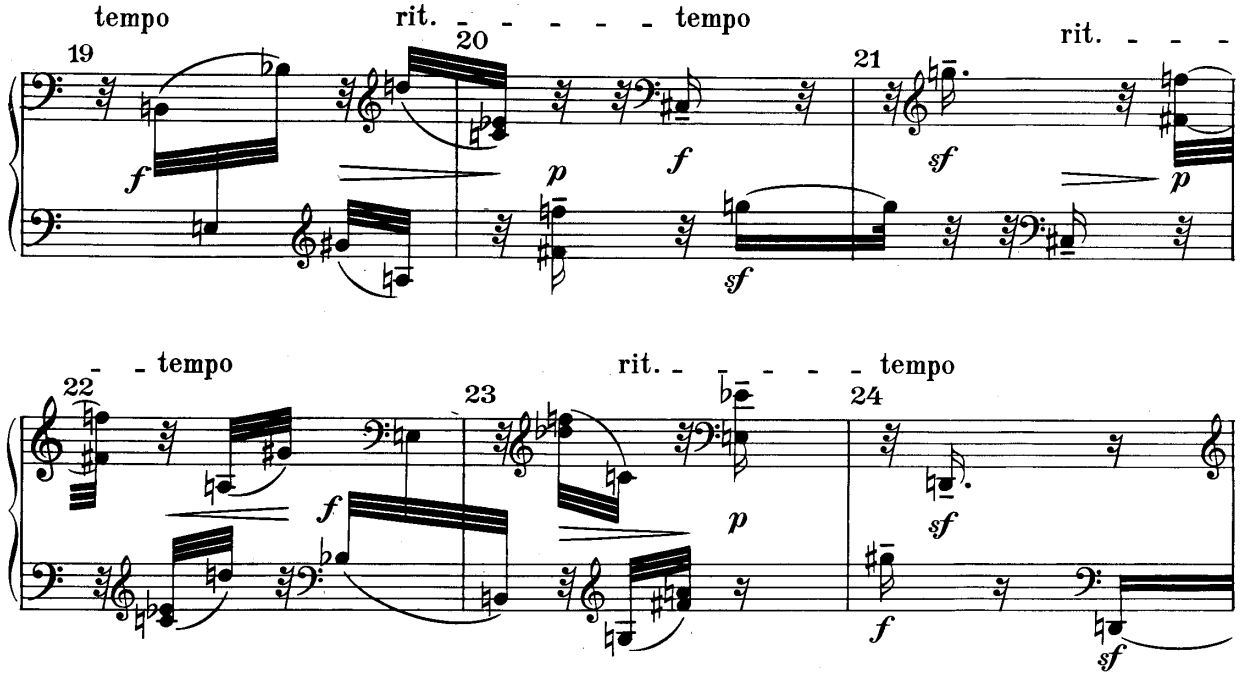


Figure 1-8. Bars 19–23 in Webern's Op. 27.

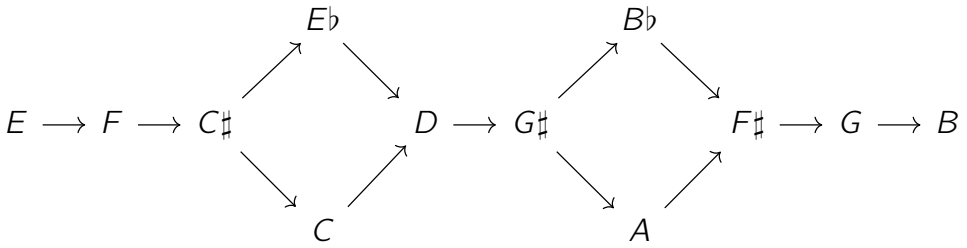


Figure 1-9. We denote the underlying partial order depicted in this aggregate realization by D_3.

The last step in figuring out Webern's total order, according to our theoretical assumptions so far, is to take intersections of the partial orders depicted above with aggregate realizations. Assuming one could actually hear the presentation of the series without being victimized by any of the numerous pitfalls represented by all the registral displacements in the piece, and furthermore assuming one could freely invert and transpose twelve ordered and different pitch classes in their heads almost instantly, one would possibly arrive with the intersection $D_1 \cap D_2$

at measure 10. An aggregate realization of the intersection is shown in Fig. 1-10. One would still be puzzled, however, since there are still three incomparabilities that prevent one from comprehending the underlying series.

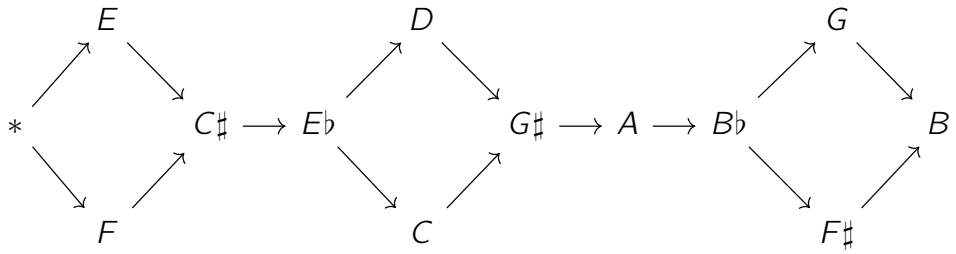


Figure 1-10. The intersection $D_1 \cup D_2$.

The next two pictures depict aggregate realizations for the intersections $D_1 \cap D_3$ and $D_2 \cap D_3$. Having reached measure 23, we are finally in a position to obtain Webern's series. Both intersections, however, still carry incomparabilities, so we will only be able to achieve our goal by taking the intersection of all three aggregate realizations.

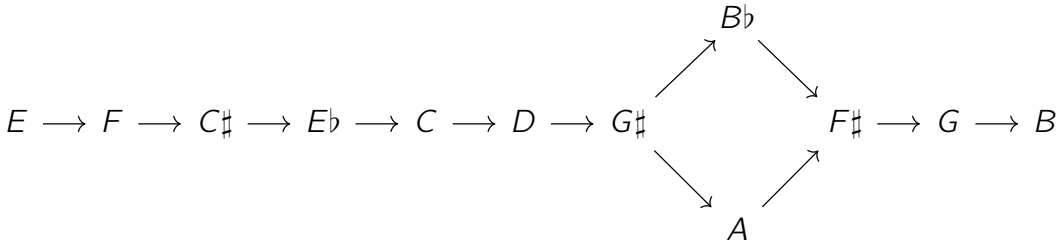


Figure 1-11. The intersection $D_1 \cup D_3$.

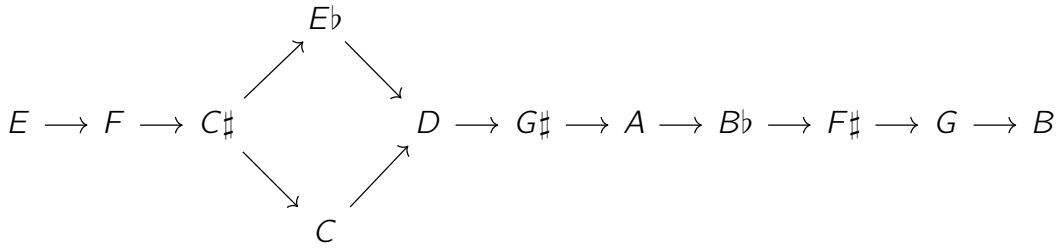


Figure 1-12. The intersection $D_2 \cup D_3$.

It is with bitter-sweet joy that we finally declare victory in Fig. 1-13, as we know we would not stand a chance relying on hearing alone.

$$E \longrightarrow F \longrightarrow C\sharp \longrightarrow E\flat \longrightarrow C \longrightarrow D \longrightarrow G\sharp \longrightarrow A \longrightarrow B\flat \longrightarrow F\sharp \longrightarrow G \longrightarrow B$$

Figure 1-13. The intersection $D_1 \cup D_2 \cup D_3$.

There is much to be said about Ex. 1.1.7, particularly in what regards the aural perception of the composer's constructs. There is even more to be said if we take into consideration the fact that, as the twentieth-century unraveled, it became in many cases increasingly more difficult to establish how a piece of music was constructed with the presence of the score alone, much less simply by aural inference. It becomes less straightforward to determine the role of the listener in these cases, especially whenever one insists that the listener should be able to aurally infer *every* construct in a musical composition. We argue here that, as composers differentiate themselves from common practice, compositional technique becomes more subjective, and often indiscernible in the musical discourse. That does not mean, however, that musical fruition should be hindered in any sense, or that the composer's channel of communication with the listener has been narrowed in any way. It is simply the case that, even when not every single part of a structure is fathomed, we are still able to experience the structure itself. We do not need a blueprint to live in a house, we do not need a wrench to drive a car, and we certainly do not begin to understand the intricacies of the very universe in which we exist. And we do exist nonetheless.

The role of the analyst, on the other hand, also becomes more difficult to define. Whenever the structure of a piece of music becomes impossible to follow from its score alone, a theorist must reconsider the weight of musicological work in the analysis of such repertoire. In other words, it may very well be impossible to analyze a composition without resorting to the composer, or to a musicological study thereof. That is especially the case with algorithmic composition, which is a major avenue to which we shall apply our main object of study. Analyzing an algorithmically generated piece of music without prior knowledge of the algorithm

itself can only go as far as devising hearing strategies for the piece, as well as conjecturing what the algorithm might have been. It may be of substantial value, it may even carry more value than an analysis of the algorithm itself. And, just like with any piece of software, understanding the code is not a prerequisite to using said software, much less to determining whether it fulfills its purpose, or whether it is a good piece of software or not. However, if one intends to determine precisely how a piece of music was *constructed*, then knowledge of its blueprint becomes essential. And that determination is often of primary interest to composers.

The compositional techniques we are about to define and generalize in this study are notoriously of structural relevance. It does not mean that they cannot be aurally discerned, and in fact many authors devote considerable time devising hearing strategies for them, particularly when they are used to structure pitch. Nevertheless, our approach here will be restricted to the intrinsic qualities of these techniques, and we shall not pursue their aural perception in depth, simply because there may not be any intention from the composer's part for them to be heard at all. A familiar example might be the music of Stravinsky. Upon analyzing his use of rotational arrays, it becomes clear that the composer uses them as a generative techniques, rather than as a foreground musical entities. Analyzing such pieces often resort to techniques that have no connection whatsoever to any sort of hearing strategy, like Forte's employment of K and Kh set complexes, which yield a nexus set that ultimately cannot be heard. We aggravate this discussion with the idea that, more often than not, we shall employ the techniques we are about to present to dimensions other than pitch, and in particular to algorithmically determining spectral components and timbre. In this sense, their outcomes become textural elements of a composition, but still well within the canon of tasks a composer needs to exercise in the craft of a piece.

Whereas aggregate realizations correspond to a totally ordered sequence of disjoint subsets of the free aggregate, a *columnar (aggregate) realization* is, on the other hand, a set of disjoint row segments where, even though the internal order of each segment is total, all segments are pairwise incomparable. In addition, we require that a columnar realization contain the

free aggregate as a subset, so that we get all pitch classes belonging to a given base in every column. This is essentially a combination matrix in twelve-tone theory.

Example 1.1.8. *[1, 201, 210] The intersection of the set of all aggregate realizations with the set of all columnar realizations contains the set of all total orders (and thus all row segments), as well as the free aggregate. A total order is trivially an aggregate realization, and it is trivially a columnar realization. Any total order contains the free aggregate per our requirements.*

1.1.2 Row Segments

The concept of a *row segment*, which is fairly straightforward in twelve-tone theory, has an analogue here in the sense that we define it as a pitch-class relation that is reflexive, transitive, and antisymmetric. We impose the additional requirement that some subset of its order constraints also satisfy trichotomy. Arguably more important is the concept of an *embedded segment*. For any partial order that covers some row segment, we say that segment is an embedded segment of the partial order. By transitivity of covering, any other partial order covering the former partial order will have the aforementioned row segment embedded in it, including naturally any total order in its total order class. In this sense, a partial order may be seen as the union (possibly the extension thereof) of its various embedded segments:

Example 1.1.9. *[1, 200] Consider two total orders* $X = \{0, 1, 7, 2, 10, 9, 11, 4, 8, 5, 3, 6\}$ *and* $Y = \{1, 2, 8, 3, 11, 10, 0, 5, 9, 6, 4, 7\}$ *where, in particular,* $Y = \mathsf{T}_1(X)$*. Seen as total orders, one can take the intersection* $X \cap Y$*, then prune it. This process yields a graph whose longest row segments are* $\{1, 2, 10, 9, 4\}$*,* $\{1, 2, 10, 5, 6\}$*,* $\{1, 2, 11, 5, 6\}$*,* $\{1, 2, 8, 5, 6\}$*, and* $\{1, 2, 8, 3, 6\}$*. These row segments are, in turn, the longest that are embedded segments of both* X *and* Y*.*

It is interesting to point out that the procedure described in Ex. 1.1.9 is in fact an algorithm to find common tones under transposition, and can be extended to any other pitch-class operation. This is a much needed addition to a the well-known technique of *counting* common tones.

Theorem 1.1.10. *[2, 10] The number of common tones between a set S and some transposition of itself is given by*

$$|S \cap \mathsf{T}_n(S)| = |\{x - y = n : x, y \in S\}| \ . \tag{1–5}$$

The number of common tones between a set S and some inversion of itself is given by

$$|S \cap \mathsf{T}_n\mathsf{I}(S)| = 2 \cdot |\{x + y = n : x, y \in S\}| + |\{a \in S : 2a = n\}| \ . \tag{1–6}$$

Moreover, the cardinality of the set $\{a \in S : 2a = n\}$ is at most 2.

Proof. We must count the occurrences of pairs of pitch classes that are interchanged by the operation at hand and double them, for if x maps onto y under some $\mathsf{T}_n\mathsf{I}$, then certainly y maps onto x under the same operation, given that every inversion operation has order two. In addition to that, we must account for the occurrences of pitch classes that may map onto themselves under the aforementioned operation. For any pair $a \neq b \in S$, it follows a and b are exchanged by some operation $\mathsf{T}_n\mathsf{I}$ whenever both $\mathsf{T}_n\mathsf{I}(a) = b$ and $\mathsf{T}_n\mathsf{I}(b) = a$ hold. Since $\mathsf{T}_n\mathsf{I}(a) = -a + n$, and similarly $\mathsf{T}_n\mathsf{I}(b) = -b + n$, if the pair is exchanged, then we must have $-a + n = b$ and $-b + n = a$ both true. Adding the last two expressions and yields $a + b = n$, which is the first set in the right-hand side of the formula. As discussed above, the cardinality of this set must be doubled. We have for any a that $\mathsf{T}_n\mathsf{I}(a) = a + n$, hence $a = \mathsf{T}_n\mathsf{I}(a) \iff a = -a + n$, that is, whenever $2a = n$. That is the second set in the formula. Finally, for any pair (a, n) such that $a = \mathsf{T}_n\mathsf{I}(a)$, we also have $a + 6 = -(a + 6) + n \iff 2a = n$, so that by the above it follows $a + 6 = \mathsf{T}_n\mathsf{I}(a + 6)$. Thus the set $\{a \in S : 2a = n\}$ has cardinality at most 2, proving the last assertion. □

Example 1.1.11. *[2, 11] Write $S = \{0, 1, 4, 5, 8, 9\}$ and consider some inversion operation. An application of Th. 1.1.10 yields the table below.*

In practice, however, many composers choose to compute common tones by writing down an entire matrix. This procedure has the advantage of, not only giving all indices of transposition under which a set shares common tones with itself or other set, but it is also

Table 1-1. Common tones under inversion between $S = \{0, 1, 4, 5, 8, 9\}$ and itself.

n	0	1	2	3	4	5	6	7	8	9	10	11
$2 \cdot \lvert\{x + y = n : x, y \in S\}\rvert$	2	6	2	0	2	6	2	0	2	6	2	0
$\lvert\{a \in S : 2a = n\}\rvert$	1	0	1	0	1	0	1	0	1	0	1	0
Total	3	6	3	0	3	6	3	0	3	6	3	0

possible to determine whether these common tones will preserve their ordering after the transform by examining the matrices' diagonals.

Example 1.1.12. *[3, 49] Let X be an n-tone row seen as a column vector, and consider the $n \times n$ matrix $A = [X, \cdots, X]$. In particular, the matrix $B = A - A^T$ will have a main diagonal of zeros, which indicates that X shares with itself n common tones under T_0. If there are k threes in the matrix, then X will share with itself k tones under T_3. There is no requirement that we compare a row with itself. If, for instance, A is given as above, and $\bar{A}$ is the matrix for the row $\bar{X}$, then counting the number k of, say, threes in $B = A - \bar{A}^T$, will mean in turn that X and $\mathsf{T}_3(\bar{X})$ share k tones under T_3. Naturally, the main diagonal of B will not comprise only zeros if $A \neq \bar{A}$. To find common tones under inversion, we let $B = A + A^T$ and, similarly, M and $\mathsf{M} \circ \mathsf{I}$ become $B = A - \mathsf{M}(A)^T$ and $B = A + \mathsf{M}(A)^T$. Finally, if the indices we are counting are disposed in any of the matrix's diagonals, then they will preserve ordering after the transform, thus becoming embedded segments; if, in addition, they are adjacent, then they will in fact be row segments shared by X and the transform of $\bar{X}$.*

The matrix for finding common tones under transposition that is described in Ex. 1.1.12 has the interesting property that it becomes a symmetric matrix when $A = \bar{A}$ and we take its elements as interval classes. This reflects the fact that, if X shares k tones with $T_i(X)$, then surely $T_i(X)$ will share exactly k tones with $T_{12-i}(X)$. If i is an interval class, then $i = 12 - i$, showing why the matrix will be symmetric. The matrix of common tones under inversion

is always symmetric, regardless we take its elements as interval classes. For multiplicative operations, however, we will often lack multiplicative inverses in twelve tones. For p-TET systems where p is prime, we do get symmetric matrices for multiplicative operations as well, but those will sometimes require a proper definition of multiplicative interval classes.

The importance of the procedure described in Ex. 1.1.9 is due to the fact that the technique in Ex. 1.1.12 can only go as far as counting the number of common tones, whereas the former procedure can actually tell us *what* these common tones are in a more straightforward manner. In addition, it can be very computationally effective if we write the rows we wish to compare as binary square matrices describing their order constraints. Taking then their intersection is easy, since we can rely on binary operations for this type of matrix. Pruning, if we wish to express the result as a graph, is also fast and simple. That being said, neither the above techniques addresses one of our primary concerns regarding derivation: given a set of order numbers and a transformation, what series are capable of producing transforms of *themselves* when pulled from those order numbers. This is in essence the exact opposite of what we have demonstrated above, where we depart from a series and attempt to understand its embedded segments. And it is arguably not a coincidence, as much of the twelve-tone theory devised since Forte has a strong analytical bias. What we are mainly interested, however, is in how these techniques fit onto the compositional scheme of things. And for that, having a two-way street where, on one hand we understand raw compositional materials and, on the other, we formulate them, is crucial. Moreover, as we attempt to construe orderedness as a generative procedure in all generality, it becomes imperative that we break our ties with purely analytical music theory and begin to delve into the mathematics of these general constructs. As the example below suggests, mathematics can greatly simplify the way we approach certain concepts and, depending on the task at hand, may even be decisive in determining whether it is feasible or not to pursue certain compositional avenues. For algorithmic composition, in particular, having a firm theoretical understanding of some generative procedure can greatly simplify the composer's algorithm, as well as make it more efficient.

Example 1.1.13. *We can demonstrate 1.1.11, and also the omitted proof of 1.1.10 under transposition in a much simpler way with a little bit of abstract algebra by observing the cycle decomposition of each operation at hand. If $n = 3$, then we have*

$$\mathsf{T}_3\,\mathsf{I} = (0\ 3)(1\ 2)(4\ 11)(5\ 10)(6\ 9)(7\ 8)\ . \tag{1–7}$$

Hence, under $\mathsf{T}_3\,\mathsf{I}$, every pitch-class in $S = \{0, 1, 4, 5, 8, 9\}$ maps to the complement of S. If the operation is, for instance, T_9, then since

$$\mathsf{T}_9 = (0\ 9\ 6\ 3)(1\ 10\ 7\ 4)(2\ 11\ 8\ 5)\ , \tag{1–8}$$

we get straightforwardly that $S = \{0, 1, 4, 5, 8, 9\}$ shares three common tones with $\mathsf{T}_9(S)$, namely $0 \mapsto 9$, $4 \mapsto 1$, and $8 \mapsto 5$.

In a completely analogous manner to the principle of pulling row segments from a series, and in the interest of building two-way roads, we can most certainly concatenate row segments in order to formulate a new series.

Example 1.1.14. *[1, 200] Let $X = \{0, 1, 2\}$ and $Y = \{3, 4, 5\}$ be row segments. We actually require that $X \cap Y = \emptyset$ when seeing X and Y as partial orders. The concatenation $X|Y$ will then be the partial order:*

$$X \cup Y \cup \{\{a, b\} : a \in X, b \in Y\}\ . \tag{1–9}$$

It follows immediately that both X and Y are embedded row segments of $X|Y$. Note that our use of concatenation here is sequential, that is, the entire row segment X precedes the entire row segment Y in $X|Y$. In other words, we do not allow intercalation of segments when concatenating.

1.2 A Survey of Derivation Techniques

In this section we devote our attention to expanding the idea first presented in Ex. 1.1.1. In it, we devise a type of derivation in which the series being displayed vertically is unrelated to the series being derived horizontally, except for the fact that both share a row segment when we retrograde one of the rows. The construction is part of a more general procedure, in

which we always match a row with a retrograded transform of itself. In the particular case of Ex. 1.1.1, the transform, say F, was T_0 for simplicity, but arbitrary pitch-class operations may be used. Denote the given series by S, and define similarly the derived row as $V = V_1|V_2$, that is, V is a concatenation of two row segments. The only requirement is that V_1, the segment we choose to single out, remains invariant under F, which will force V_2 to also be invariant under F. In Ex. 1.1.1 this requirement is satisfied trivially. The aforementioned procedure is given schematically in Table 1-2.

Table 1-2. Schematics of a derivation procedure involving the retrograde and an arbitrary operation F.

	S	$\mathrm{R}\circ F(S)$
V	V_1	V_2
$\mathrm{R}\circ F(V)$	$\mathrm{R}\circ F(V_2)$	$\mathrm{R}\circ F(V_1)$

Example 1.2.1. *In this example, we illustrate a derivation procedure similar to that in Ex. 1.1.1 where the operation F is not trivial. We again take the row in Berg's* Lulu, $S = \{10, 2, 3, 0, 5, 7, 4, 6, 9, 8, 1, 11\}$, *and consider the segment $\vec{s} = [10\ 0\ 5\ 7]^T$ as a column vector. Now let $A^T = [\ \vec{s}\ |\ \vec{s}\ |\ \vec{s}\ |\ \vec{s}\ ]$ be the square matrix whose every column is equal to $\vec{s}$. Then*

$$A + A^T = \begin{bmatrix} 2 & 10 & 3 & 5 \\ 10 & 0 & 5 & 7 \\ 3 & 5 & 10 & 0 \\ 5 & 7 & 0 & 2 \end{bmatrix} \pmod{12}\ . \tag{1–10}$$

In particular, we see that the row segment $\{10, 0, 5, 7\}$ is invariant under $\mathrm{R}\,\mathrm{T}_5\,\mathrm{I}$, since we get a main diagonal of fives when we mirror the matrix above vertically. Thus, by matching S with $\mathrm{R}\,\mathrm{T}_5\,\mathrm{I}(S)$, we may get the row segment $\{10, 0, 5, 7\}$ in the derived row V itself, rather than in its retrograde, as was the case in Ex. 1.1.1. Setting it to V_2, say, yields $V =$

$\{2,3,4,6,9,8,1,11,10,0,5,7\}$, *so we get the combination matrix below.*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} . & 2 & 3 & . & . & . & 4 & 6 & 9 & 8 & 1 & 11 & . & . & . & . & . & . & 10 & 0 & 5 & . & . & 7 \\ 10 & . & . & 0 & 5 & 7 & . & . & . & . & . & . & 6 & 4 & 9 & 8 & 11 & 1 & . & . & . & 2 & 3 & . \end{array}\right] . \tag{1–11}$$

It is usually the case in this type of derivation that, unlike Ex. 1.2.1, the invariance of V under F, or under $\mathrm{R}\circ F$ for that matter, does not preserve any ordering.

Example 1.2.2. *[1, 212] Let* $S = \{0,1,7,2,10,9,11,4,8,5,3,6\}$ *and consider the operation*

$$\mathrm{T}_2\,\mathrm{I} = (0\ 2)(3\ 11)(4\ 10)(5\ 9)(6\ 8)\ . \tag{1–12}$$

Upon inspection of the cycles of $\mathrm{T}_2\,\mathrm{I}$*, we see that the segment* $\{0,1,7,2\}$ *is invariant under it. If we consider a combination matrix involving* $\mathrm{R}\,\mathrm{T}_2\,\mathrm{I}$*, however, then we shall not obtain the same ordered result as in Ex. 1.2.1, since both* (1) *and* (7) *are fixed points. The same would happen to any* $\mathrm{T}_k\,\mathrm{I}$ *where* k *is even. In particular, this shows that, in order to obtain the same ordered row segment in both columns of the combination matrix,* F *cannot be trivial.*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} 0 & . & . & 1 & . & 7 & . & . & . & 2 & . & . & 10 & 9 & . & 11 & 4 & 8 & . & 5 & . & 3 & 6 & . \\ . & 8 & 11 & . & 9 & . & 6 & 10 & 3 & . & 5 & 4 & . & . & 0 & . & . & . & 7 & . & 1 & . & . & 2 \end{array}\right] . \tag{1–13}$$

As a consequence of F and the retrograde operation, combination matrices of this type feature two columns that are upside down F-mirrors of each other. In line with our theoretical remarks so far, more can be said about such combination matrices.

Proposition 1.2.3. *[1, 211, 214] Consider a 2-row combination matrix* C *where a row is derived via the retrograde and some operation* F*. Denote the derived row by* $V = V_1|V_2$*. Then the first column is the partial order* $C_1 = V_1 \cup \mathrm{R}\circ\mathrm{F}(V_2)$*, and similarly the second column is the partial order* $C_2 = V_2 \cup \mathrm{R}\circ\mathrm{F}(V_1)$*, such that* $C_2 = \mathrm{R}\circ F(C_1)$*. If* D *is a partial order that covers* C_1*, then* $\mathrm{R}\circ F(D)$ *will cover* C_2*, and if* D *is in the total order class of* C_1*, that is,* D *is a row that can be linearized from* C_1*, then* $\mathrm{R}\circ F(D)$ *is in the total order class of* C_2*.*

Finally, 2-row derivations of this type exist for arbitrary rows. The number of such combination matrices for any given row depends on the invariances of the chosen partition of V.

It is not clear from the literature, however, how many 2-row combination matrices involving R we can get for an arbitrary row *exactly*. We shall investigate what this number is in the present study.

1.2.1 Folded Derivation

A related technique involving greater than 2-row counterpoint is achieved by *folding* a derivation matrix. The process is equivalent to first constructing a matrix of hexachordal combinatoriality in the traditional sense, that is, where hexachords are seen as tropes rather than as row segments, then deriving rows from this matrix using the techniques above. Let S be a row whose first hexachord is invariant under the operation G. Consider a row $V = V_1|V_2$ and an operation F such that S is in the total order order class of $V_1 \cup \mathrm{R} \circ \mathrm{F}(V_2)$. Then putting S in counterpoint with $G(S)$, as well as deriving V from $S|\,\mathrm{R} \circ F(S)$ yields the schematic representation in Table 1-3:

Table 1-3. Schematics of a folded derivation procedure involving the retrograde.

	S	$\mathrm{R} \circ F(S)$
V	V_1	V_2
$\mathrm{R} \circ F(V)$	$\mathrm{R} \circ F(V_2)$	$\mathrm{R} \circ F(V_1)$
$\mathrm{R} \circ GF(V)$	$\mathrm{R} \circ GF(V_2)$	$\mathrm{R} \circ GF(V_1)$
$G(V)$	$G(V_1)$	$G(V_2)$
	$G(S)$	$\mathrm{R} \circ GF(S) = \mathrm{R} \circ FG(S)$

It is argued without proof in [1, 215] that, for a derivation procedure such as the one in Table 1-3, the folded derivation has to vertically satisfy G, while satisfying F horizontally.

Thus F and G would have to commute in this case. The caveat that the operation chosen for the hexachordal combinatoriality part must commute with the operation chosen for the derivation part still needs to be confirmed, however, as other derivation matrices that have similar constraints do not require commutativity. Ex. 1-3 shows a musical application of folded derivations.

Example 1.2.4. *[1, 215] Let* $S = \{0, 1, 11, 3, 8, 10, 4, 9, 7, 6, 2, 5\}$ *and consider the operation* T_6*:*

$$\mathsf{T}_6 = (0\ 6)(1\ 7)(2\ 8)(3\ 9)(4\ 10)(5\ 11)\ . \tag{1–14}$$

Let $V_1 = \{1, 3, 8, 9, 7, 2\}$*. In particular, since the unordered set* V_1 *is* T_6*-invariant, we get* $V_2 = \{11, 0, 10, 4, 5, 6\}$ *and the following combination matrix:*

$$\left[\begin{array}{cccccc|cccccc} 1 & 3 & 8 & 9 & 7 & 2 & 11 & 0 & 10 & 4 & 5 & 6 \\ 0 & 11 & 10 & 4 & 6 & 5 & 8 & 1 & 3 & 2 & 9 & 7 \end{array}\right]\ . \tag{1–15}$$

Consider further the operation $\mathsf{T}_5\mathsf{I}$*:*

$$\mathsf{T}_5\mathsf{I} = (0\ 5)(1\ 4)(2\ 3)(6\ 11)(7\ 10)(8\ 9)\ . \tag{1–16}$$

Then $S_1 = \{0, 1, 11, 3, 8, 10\}$ *maps to its complement under* $\mathsf{T}_5\mathsf{I}$*, so we can use* S *in a combination matrix where we match* S *with its transform under* $\mathsf{T}_5\mathsf{I}$ *in the usual way, that is, in a matrix of hexachordal combinatoriality. We can then derive from* $\mathsf{T}_5\mathsf{I}(S)$ *the row* $\mathsf{T}_5\mathsf{I}(V)$*. Since* $\mathsf{T}_5\mathsf{I}$ *commutes with* T_6*, as any pitch-class operation does, we get the following folded derivation:*

$$\left[\begin{array}{cccccc|cccccc} 1 & 3 & 8 & 9 & 7 & 2 & 11 & 0 & 10 & 4 & 5 & 6 \\ 0 & 11 & 10 & 4 & 6 & 5 & 8 & 1 & 3 & 2 & 9 & 7 \\ \hline 5 & 6 & 7 & 1 & 11 & 0 & 9 & 4 & 2 & 3 & 8 & 10 \\ 4 & 2 & 9 & 8 & 10 & 3 & 6 & 5 & 7 & 1 & 0 & 11 \end{array}\right]\ . \tag{1–17}$$

1.2.2 Shifted Derivation

There is a different flavor to folded derivations that arises from shifting horizontally one of the two rows of the derivation matrix. Shifted derivations may be equivalent to folding combinatoriality matrices that involve the retrograde, and there is no requirement here that the combinatoriality be hexachordal. In fact, one can choose segments to formulate a combination matrix, then fold the result. One may even match altogether different foldings of a row, as to bridge different derivations of the same generative material [1, 216].

Table 1-4. Schematics of a shifted derivation procedure involving the retrograde. The asterisk indicates there is no requirement we derive the same row class in both foldings, in which case V would be replaced by some other row V^* and G could be the identity.

	S	$\mathrm{R}\circ F(S)$	
V	V_1	V_2	
$\mathrm{R}\circ F(V)$	$\mathrm{R}\circ F(V_2)$	$\mathrm{R}\circ F(V_1)$	
$\mathrm{R}\circ G(V)^*$		$\mathrm{R}\circ G(V_2)^*$	$\mathrm{R}\circ G(V_1)^*$
$GF(V)^*$		$GF(V_1)^*$	$GF(V_2)^*$
		$GF(S)$	$\mathrm{R}\circ G(S)$

Ex. 1.2.5 shows an application of shifted derivation where combinatoriality is not hexachordal, but both foldings derive different transforms of the same row:

Example 1.2.5. *[1, 216] Consider the row* $S = \{0, 1, 7, 2, 10, 9, 11, 4, 8, 5, 3, 6\}$ *and the combination matrix given by* $\mathrm{R}\,\mathrm{T}_{10}\,\mathrm{I}(S)$ *against* $\mathrm{T}_{11}(S)$*:*

$$\left[\begin{array}{cccccccc|cccccccc} 4 & & 7 & & 5 & & 2 & & 6 & 11 & 1 & 0 & 8 & 3 & 9 & 10 \\ 11 & 0 & 6 & 1 & 9 & 8 & 10 & 3 & & 7 & & 4 & & 3 & & 5 \end{array}\right] \tag{1–18}$$

Deriving the row $V = V_1|V_2 = \{1,7,2,9,8,3\}|\{4,5,6,11,0,10\}$ *from* $S|\,\mathrm{R}\,\mathrm{T}_{10}\,\mathrm{I}(S)$*, and following the scheme in Table 1-4, yields the following shifted derivation:*

$$\left[\begin{array}{cccccccccccc|cccccccccccc|c}
 & 1 & 7 & 2 & & 9 & & & 8 & & 3 & & 4 & & & & 5 & & & & & 6 & \\
0 & & & & 10 & & 11 & 4 & & 5 & & 6 & & & & 7 & & & & 2 & & & \\
\hline
 & & & & & & & & & & & & & & 0 & 6 & 1 & & 8 & & & 7 & \\
 & & & & & & & & & & & & 11 & & & & & 9 & & & 10 & 3 & &
\end{array}\right]\ldots$$

$$\ldots\left[\begin{array}{cccccccccccc|cccccccccccc}
 & 6 & 11 & & 0 & & & & & & 10 & & & & & & & & & & & & \\
 & & & & & 1 & & 8 & 3 & 9 & & & & & & & & & & & & & \\
\hline
 & 7 & & & & & & 3 & & & & & 3 & & 4 & & 5 & 10 & & & 11 & & & 9 \\
3 & & & & 4 & & & & & 5 & & & & 6 & & 1 & & & & 0 & & 7 & 2 & 8
\end{array}\right] . \quad (1\text{–}19)$$

Since the constraints to formulating matrices that feature shifted derivations are more flexible, it seems plausible to conjecture that there are far more combination matrices of this type per row, than there are of the non-shifted type. This question is not at all addressed in the literature, much less how well constructs like these, shifted and not, support combination matrices without the retrograde, or even against rotations of a row. Accounting for the exact number of such constructs in a more general setting than that of twelve tones will constitute one or this paper's goals.

1.2.3 Self-Derivation

The next derivation technique to be discussed is similar to the ones above, with the additional requirement that the derived or folded rows be transforms of the original row. This is called self-derivation and is arguably the most difficult type of derivation to achieve, as not every row will lend itself to self-derivation. It is also the technique that requires the biggest theoretical machinery to understand. Self-derivation has many interesting applications, including the whole class of Mallalieu-type rows, which will be discussed in Sec. 1.3. We shall come back to a summary of our goals after exposing self-derivation and other derivation

techniques that, to this day, still lack a deeper theoretical formulation and generalization. The general scheme for a two-row combination matrix of self-derivation involving the retrograde is given in Table 1-5:

Table 1-5. Schematics of a self-derivation procedure involving the retrograde.

	$G(S)$	$\mathrm{R}\circ FG(S)$
S	S_1	S_2
$\mathrm{R}\circ F(S)$	$\mathrm{R}\circ F(S_2)$	$\mathrm{R}\circ F(S_1)$

It is stated without proof in [1, 217] that the operations F and G in Table 1-5 must commute, as well. For this self-derivation on two-row counterpoint case, however, simple inspection of the examples that follow such statement already affords a counter-proof.

Example 1.2.6. *[1, 218] Let* $S = S_1|S_2 = \{3, 8, 1, 0, 9, 6\}|\{4, 7, 10, 5, 2, 11\}$ *and consider the operation* $\mathrm{T}_9\,\mathrm{I}$*:*

$$\mathrm{T}_9\,\mathrm{I} = (0\ 9)(1\ 8)(2\ 7)(3\ 6)(4\ 5)(10\ 11)\ . \tag{1–20}$$

In particular, both S_1 *and* S_2 *are invariant under* $\mathrm{T}_9\,\mathrm{I}$*. What is certainly not obvious, is that we can derive* S *and* $\mathrm{T}_9\,\mathrm{I}(S)$ *from a combination matrix where the first column is* $\mathrm{T}_7(S)$ *and the second is* $\mathrm{T}_9\,\mathrm{I}\circ\mathrm{R}\,\mathrm{T}_7(S) = \mathrm{R}\,\mathrm{T}_2\,\mathrm{I}(S)$*, as shown in the following self-derivation matrix:*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} & & 3 & 8 & & & 1 & & & & 0 & 9 & 6 & & & & 4 & 7 & 10 & & 5 & 2 & & & 11 \\ 10 & & & & 7 & 4 & & & 11 & 2 & 5 & & & 3 & 0 & 9 & & & & & 8 & & & 1 & 6 & \end{array}\right]\ . \tag{1–21}$$

Here we have $F = \mathrm{T}_9\,\mathrm{I}$ *and* $G = \mathrm{T}_7$*. It is certainly not the case that* F *and* G *commute, as* $\mathrm{T}_2\,\mathrm{I} = F\circ G \neq G\circ F = \mathrm{T}_4\,\mathrm{I}$.

Similarly to general derivation, two-row self-derivations can be folded. What is unique to self-derivation, however, is that the *same* pattern of order numbers that give the self-derived transform of a row can be used to fold each row of a combination matrix, as seen in Ex. 1.2.7.

The folded rows can subsequently be folded, and this procedure can generate many levels of self-derivation.

Example 1.2.7. *[1, 221] Let* $S = \{0, 11, 5, 10, 4, 2, 7, 9, 8, 3, 6, 1\}$ *and consider the following combination matrix given by* $T_2(S)|\,\mathrm{R}\,\mathrm{T}_2(S)$*, whose derived rows are* S *and* $\mathrm{R}(S)$*:*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} 0 & & 11 & 5 & & & 10 & & 4 & & & 2 & 7 & & 9 & & & 8 & 3 & & & & 6 & & 1 \\ 1 & & 6 & & & & & 3 & 8 & & & 9 & & 7 & 2 & & 4 & & & 10 & & & & 5 & 11 & & 0 \end{array}\right]. \tag{1–22}$$

Subjecting the entire matrix to T_1 *yields:*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} & 1 & & 0 & 6 & & & 11 & & & 5 & & 3 & & 8 & & & 10 & & & 9 & 4 & & & 7 & & 2 \\ 2 & & & 7 & & & 4 & 9 & & & 10 & & 8 & 3 & & 5 & & & 11 & & & & 6 & 0 & & 1 & \end{array}\right]. \tag{1–23}$$

We can then pull the entire matrix in Eq. 1–22 from the first row of Eq. 1–23:

$$\left[\begin{array}{cccccccccccc|cccccccccccc} & & & 0 & & & & 11 & & & 5 & & & & & & 10 & & & & 4 & & & & & 2 & \\ & 1 & & & & 6 & & & & & & & 3 & & 8 & & & & & & 9 & & & & 7 & & \\ 2 & & & 7 & & & 4 & 9 & & & 10 & & 8 & 3 & & 5 & & & 11 & & & & 6 & 0 & & 1 & \end{array}\right] 2 \cdots. \tag{1–24}$$

The next example provides a musical application of folded self-derivation matrices that constitute the main compositional procedure in Ciro Scotto's *Tetralogy*.

Example 1.2.8. *[4, 180] Let* $S = \{0, 4, 7, 3, 11, 2, 10, 1, 6, 8, 9, 5\}$ *and consider the self-derivation array in Fig. 1-14.*

T_7	7	e 2	t 6	9	5 8	1	3	4 0
T_7R	0 4	3	1	8 5	9	6 t	2 e	7
	T_0				T_0R			

Figure 1-14. Self-derived row in Scotto's *Tetralogy*.

Similarly to Ex. 1.2.7, we can fold the above matrix in order to obtain subsequent levels of derivation. Unlike Ex. 1.2.7, however, Scotto derives a rotated transform of S *from* $\mathrm{R}\,\mathrm{T}_7(S)$*.*

The rows in Fig. 1-15 are thus $\mathrm{T}_2(S), \mathrm{R}\,\mathrm{T}_2(S), \rho_6\,\mathrm{R}\,\mathrm{T}_2(S)$ *and* $\rho_6\,\mathrm{T}_2(S)$*, where* ρ *is the cyclical rotation operator.*

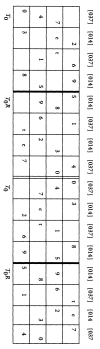


Figure 1-15. Folded derivation in Scotto's *Tetralogy*.

In an entirely Schenkerian fashion, Scotto utilizes the folded array as the source material for the middle-ground structure of Tetralogy. *The only difference between the derivation array and the Schenkerian graph is that the* $\mathrm{T}_2(S)$ *now corresponds to the alto register, as seen in Fig. 1-16. One of the foreground realizations of the Schenkerian graph in Fig. 1-16 is given in Fig. 1-17. Rather than a strict serial composition,* Tetralogy *employs a variety of prolongation procedures which are in line with its Schenkerian orientation, but unfortunately beyond the scope of this discussion.*

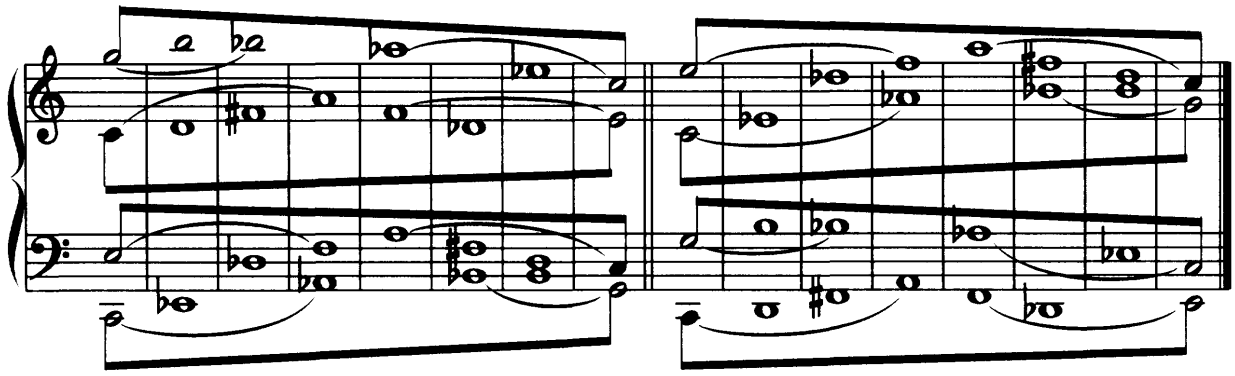


Figure 1-16. Schenkerian middle-ground structure in Scotto's *Tetralogy*.

The idea of prolongation in Tetralogy *extends beyond the foreground musical surface, and is applied as well to the middle-ground structure itself, effectively pushing it further into the*

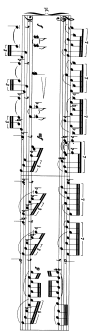


Figure 1-17. Musical realization of the middle-ground in Scotto's *Tetralogy*.

background of the piece. The prolongation of the first bar in Fig. 1-16 is depicted in Fig. 1-18, and a musical realization thereof is displayed in Fig. 1-19.

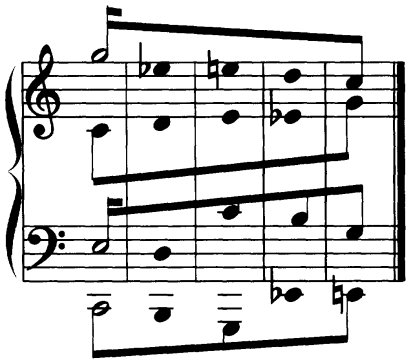


Figure 1-18. Prolongation of the middle-ground structure in Scotto's *Tetralogy*.

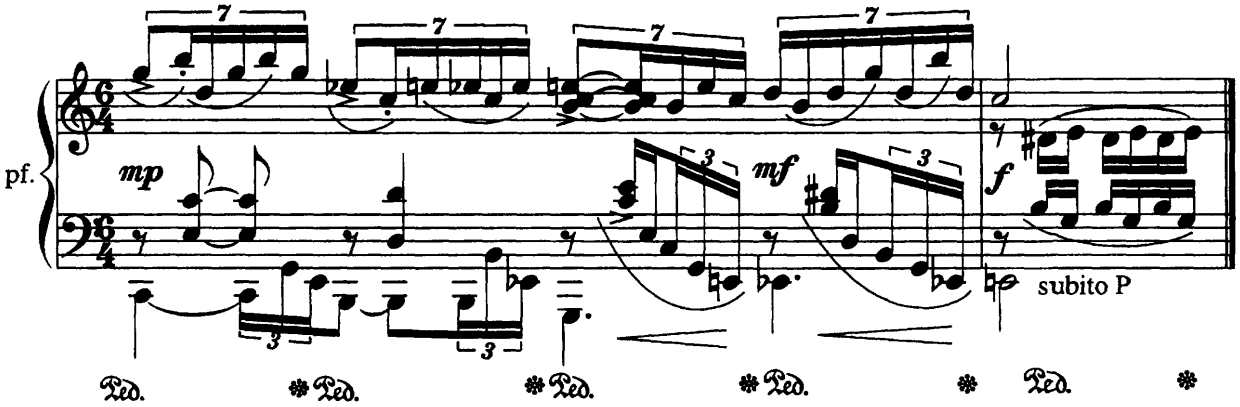


Figure 1-19. Musical realization of the prolonged middle-ground in Scotto's *Tetralogy*.

The only algorithm that currently exists to find self-derivation matrices is brute force, that is, by trial and error. One must often compose the row while trying to derive a transform

of the row itself in a matrix. Self-derivation is the absolute primary concern of this paper, especially because there is so little we know about it. Self-derivation is what makes an order number in a row have multiple functions. Other types of derivation are also important as compositional tools, but they are ultimately special cases of the more constrained type, which is that of self-derivation matrices. In other words, we shall make our priority to investigate the existence of a self-derived matrix for any given row in all generality, that is, for arbitrary n-tone equal temperament systems. We shall then devise a general algorithm to obtain such matrices if they exist, as well as investigate how many such matrices do exist for any particular choice of equal temperament system. We shall finally confront our findings with the results in [5], which uses a brute force algorithm to compute self-derived rows of a few particular types in the 12-TET. For generalized results, there will be nothing with which to compare, as this territory remains, to this day, mostly uncharted.

1.2.4 Derivation from Aggregate Realizations

The remainder of this chapter will be devoted to exposing derivation matrices that do not necessarily involve the retrograde, then exploring the concept of Mallalieu-type rows, which represent a very peculiar class of self-deriving rows. Mallalieu rows are somewhat better understood in the 12-TET than general derivation, and have many beautiful compositional applications.

Example 1.2.9. *[1, 224] Let* $S = \{0,1,7,2\}|\{10,9\}|\{11,4,8,5\}|\{3,6\}$. *Given that we always obtain the complement of a row's first hexachord from its second hexachord, we get the trivial case of hexachordal combinatoriality when we match a row with its retrograde. Such combination matrix will have four columns, because of the way we chose to partition* S*:*

$$A = \left[\begin{array}{c|c|c|c} \{0,1,7,2\} & \{10,9\} & \{11,4,8,5\} & \{3,6\} \\ \{6,3\} & \{5,8,4,11\} & \{9,10\} & \{2,7,1,0\} \end{array}\right] . \tag{1–25}$$

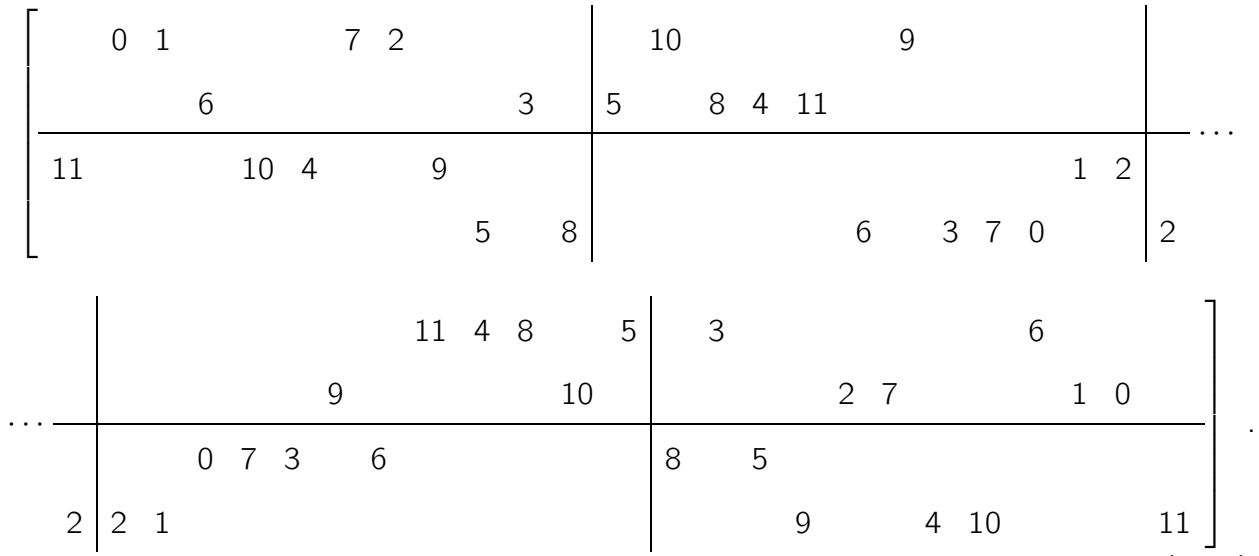
Now consider the cycles of $\mathrm{T}_{11}\mathrm{I} = (0\ 11)(1\ 10)(2\ 9)(3\ 8)(4\ 7)(5\ 6)$. *In particular, the first column of* A *will comprise the partial order* $S_1 \cup \mathrm{R}(S_4)$*, and this union will, in turn, map*

onto its hexachordal complement under $\mathsf{T}_{11}\,\mathsf{I}$*, as it takes precisely one element from each of the operation's cycles, all of which have length two. Since the second column above is the complement of the first, it will also map onto its complement under* $\mathsf{T}_{11}\,\mathsf{I}$*. Because the third and fourth columns are mirrors of the first two, we do get hexachordal combinatoriality under* $\mathsf{T}_{11}\,\mathsf{I}$ *in every column, that is, for the partial orders* $S_1 \cup \mathsf{R}(S_4)$ *and* $S_2 \cup \mathsf{R}(S_3)$*, as well as their retrogrades. We do not get the same result between* S *and* $\mathsf{T}_{11}\,\mathsf{I}(S)$*, that is, the first hexachord of* $\mathsf{T}_{11}\,\mathsf{I}(S)$ *is* not *the complement of the first hexachord of* S*. This procedure gives us the matrix* $\hat{A}$*.*

$$\hat{A} = \left[\begin{array}{c|c|c|c} \{0,1,7,2\} & \{10,9\} & \{11,4,8,5\} & \{3,6\} \\ \{6,3\} & \{5,8,4,11\} & \{9,10\} & \{2,7,1,0\} \\ \{11,10,4,9\} & \{1,2\} & \{0,7,3,6\} & \{8,5\} \\ \{5,8\} & \{6,3,7,0\} & \{2,1\} & \{9,4,10,11\} \end{array}\right]. \tag{1–26}$$

What we basically have above is a sequence of four columnar realizations. We are interested in deriving members of the same row class from each column. As aforementioned, there is no algorithm to accomplish this but brute force, and there is no a priori *way of knowing whether we will succeed at this moment. So far the best we can do is take the intersection of all columns, or transforms thereof and, if the total order class of the intersection is not empty, that is, if it contains the free aggregate and does not contain any symmetry, then Th. 1.1.3 guarantees we will be able to derive representatives of this row class from each column. Let* C_1, C_2, C_3, C_4 *be the four columns of the matrix* $\hat{A}$*. It follows the row* $V = \{11,0,1,6,10,4,7,2,9,5,3,8\}$ *has the property that* $T \in \mathsf{Toc}\{\mathsf{Ext}[C_1 \cup \mathsf{R}\,\mathsf{T}_1\,\mathsf{I}(C_2) \cup \mathsf{T}_1\,\mathsf{I}(C_3) \cup \mathsf{R}(C_4)]\}$*. We can therefore derive* $[V|\,\mathsf{R}\,\mathsf{T}_1\,\mathsf{I}(V)|\,\mathsf{T}_1\,\mathsf{I}(V)|\,\mathsf{R}(V)]$ *from the columns*

of $\hat{A}$.



(1–27)

One of the problems that come with the brute force approach is that, not only is it expensive, but it is ultimately counter-intuitive. It is the same as betting on random numbers on a lottery ticket and asking ourselves whether we have won. We can look in the newspaper after the draw, but that does not increase our odds of winning at all. However, that is exactly what happens when we come up with a series out of sheer guesswork, then ask ourselves whether that series would produce some interesting derivation array. What we want is to be able to increase our odds, to somehow figure out what the winning numbers are. Luckily, this is not the lottery, and the process is certainly not random, so there must be a way to tell exactly how to put together arrays like these without any guessing whatsoever. A partial solution to this problem exists when we use some operation's cycles as building blocks for the columnar aggregates in a derivation matrix.

Example 1.2.10. *[1, 224] Let* $S = \{0,1,7,2\}|\{10,9,11,4\}|\{8,5,3,6\} = S_1|S_2|S_3$ *and consider the matrix* $A = [S|\,\mathsf{T}_4(S)|\,\mathsf{T}_8(S)]^T$.

$$A = \left[\begin{array}{cccc|cccc|cccc} 0 & 1 & 7 & 2 & 10 & 9 & 11 & 4 & 8 & 5 & 3 & 6 \\ 4 & 5 & 11 & 6 & 2 & 1 & 3 & 8 & 0 & 9 & 7 & 10 \\ 8 & 9 & 3 & 10 & 6 & 5 & 7 & 0 & 4 & 1 & 11 & 2 \end{array}\right] . \tag{1–28}$$

Now let V be in the total order class of the first columnar aggregate of A. Next, rewrite the first columnar aggregate of A as the aggregate realization A_1.

$$A_1 = \begin{array}{ccccccccccccc} & & 0 & & & & 1 & & & & 7 & & 2 \\ & \nearrow & & \searrow & & \nearrow & & \searrow & & \nearrow & & \searrow & \\ * & \longrightarrow & 4 & \longrightarrow & * & \longrightarrow & 5 & \longrightarrow & * & \longrightarrow & 11 & \longrightarrow & * & \longrightarrow & 6 & \longrightarrow & * \\ & \searrow & & \nearrow & & \searrow & & \nearrow & & \searrow & & \nearrow & \\ & & 8 & & & & 9 & & & & 3 & & 10 \end{array} \tag{1–29}$$

When we see the first column of A as an aggregate realization, it becomes clearer that any $V \in \mathrm{Toc}(A_1)$ must be a succession of augmented triads, say, $V = \{4, 8, 0, 9, 1, 5, 11, 7, 3, 2, 10, 6\}$. We know we are able to linearize V from A_1, but the question we should be asking at this point is whether we can linearize some transform of V from the other columns of A. The answer depends on the chosen operation, T_4 in this case, and on the chosen S. To verify that A_2 is a transform of A_1, it suffices to check whether there is a base-four $\mathsf{R}\,\mathsf{T}_n\,\mathsf{M}\,\mathsf{I}$ operation that maps $S_1 \pmod 4$ onto $S_2 \pmod 4$. This is easily verified, as indeed

$$S_1 \pmod 4 = \{0, 1, 3, 2\} = \mathsf{T}_2\,\mathsf{I}(\{2, 1, 3, 0\}) = \mathsf{T}_2\,\mathsf{I} \circ S_2 \pmod 4 \ . \tag{1–30}$$

The above method works because the elements in each of A_1's columns are incomparable, so we pick from within each column in any order we want. In other words, we could flip A_1 horizontally, say, and still have the same aggregate realization. Since we can reduce all of A's columns $\bmod 4$ by construction, it becomes enough to only consider each column's residue modulo four, and four-tone operations. Since $S_1 \pmod 4 = S_3 \pmod 4$, we can derive V itself from A_3, and thus obtain the following derivation matrix, where the second column is

$\mathsf{T}_2\,\mathsf{I}(V)$:

$$\left[\begin{array}{cccccccccc|ccccccc}
 & & 0 & & 1 & & & 7 & & 2 & & 10 & & & & & 9 & \\
4 & & & & & 5 & 11 & & & & 6 & & & 2 & & 1 & & \cdots \\
 & 8 & & 9 & & & & & 3 & & 10 & & 6 & & 5 & & &
\end{array}\right]$$

$$\left[\begin{array}{cccccccccc|ccccccc}
 & & & & & & 11 & & 4 & & & 8 & & & 5 & & 3 & & 6 \\
 & & \cdots & 3 & & & & & & & 8 & & 0 & 9 & & & 7 & & 10 \\
 & & & & & 7 & & 0 & & & & 4 & & & 1 & & 11 & & 2
\end{array}\right] . \quad (1\text{–}31)$$

It would be rather undesirable, however, that we should be restricted to rows that are merely the concatenation of cycles from an operation. We can certainly use the machinery hitherto developed to evaluate what other rows can be derived from A. Knowing that the columns of A are related as aggregate realizations by the operation tuple $\mathcal{A} = [\mathsf{T}_0\ \ \mathsf{T}_2\,\mathsf{I}\ \ \mathsf{T}_0]$, we can regard the A_i as columnar aggregates, then take $\hat{A} = \mathsf{Ext}[\bigcup_i(\mathcal{A}_i \circ A_i)]$. By Th. 1.1.3, any row we are able to linearize from $\hat{A}$, will be in $\bigcap_i \mathsf{Toc}(A_i)$, and thus we can derive its $\mathcal{A}_i$-transform from each i-column of A. Below is the columnar aggregate $\hat{A}$.

$$\hat{A} = \begin{array}{ccccccccccc}
 & & 0 & \longrightarrow & 1 & \longrightarrow & 7 & \longrightarrow & 2 & & \\
 & \nearrow & & \searrow\nearrow & & \searrow\nearrow & & \searrow\nearrow & & \searrow & \\
* & \longrightarrow & 4 & \longrightarrow & 5 & \longrightarrow & 11 & \longrightarrow & 6 & \longrightarrow & * \\
 & \searrow & & \nearrow\searrow & & \nearrow\searrow & & \nearrow\searrow & & \nearrow & \\
 & & 8 & \longrightarrow & 9 & \longrightarrow & 3 & \longrightarrow & 10 & &
\end{array} \quad (1\text{–}32)$$

It follows $\hat{V} = \{0, 1, 4, 8, 9, 5, 7, 2, 11, 3, 10, 6\}$ *can be linearized from* $\hat{A}$*, yielding the derivation matrix below:*

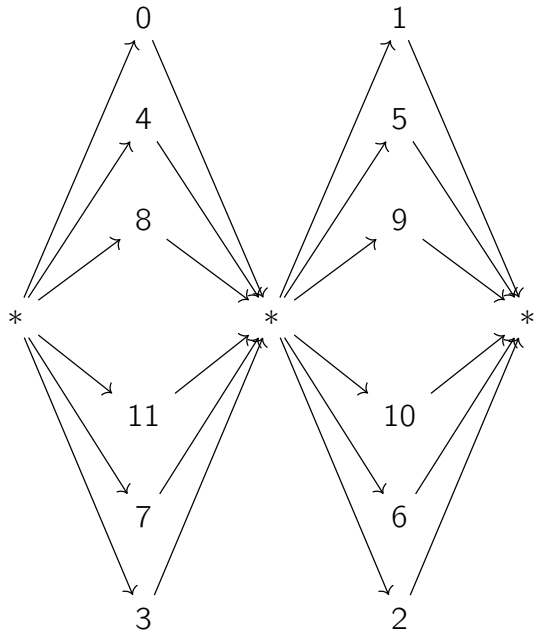
$$\left[\begin{array}{cccccccccc|cccccc} 0 & & & & 1 & 7 & & & 2 & & & & & 10 & & \\ & & & 4 & & & & 5 & 11 & & & 6 & 2 & & & 1 & \cdots \\ & 8 & 9 & & & & & & & & 3 & 10 & & 6 & 5 & & 7 \end{array}\right]$$

$$\left[\begin{array}{cccccccccc|cccccc} & & & 9 & & & 11 & & 4 & & 8 & & & & 5 & & 3 & 6 \\ & & \cdots & & 3 & & & & 8 & & 0 & & 9 & & 7 & & & 10 \\ & & & & & 0 & & & & & & & & 4 & 1 & & 11 & 2 \end{array}\right] . \quad (1\text{–}33)$$

We conclude this section with a musical example of how self-derivation may be achieved in a 6×144 matrix by exploring the symmetries of an aggregate realization.

Example 1.2.11. *Consider the aggregate realization A.*

$A =$



. (1–34)

It is easily seen that A is invariant under the set of operations $\Omega = \{\mathsf{T}_0, \mathsf{T}_4, \mathsf{T}_8, \mathsf{T}_3\,\mathsf{I}, \mathsf{T}_7\,\mathsf{I}, \mathsf{T}_{11}\,\mathsf{I}\}$*. Thus if* $\rho \in \mathrm{Toc}(A)$*, then also* $\Omega_i(\rho) \in \mathrm{Toc}(A)$*. It is also easy to see that* $\mathsf{R}(\rho) \in$

$\mathsf{Toc}(\mathsf{R}\circ\Omega_i(A))$. *Now let* $S = \{0,1,5,8,9,4,10,3,7,6,2,11\}$, *and consider the combination matrix* $\mathcal{A} = [\mathcal{A}_1|\cdots|\mathcal{A}_6]$.

$$\mathcal{A} = \left[\begin{array}{cc|cc|cc|cc|cc|cc} 0 & 1 & 5 & 8 & 9 & 4 & 10 & 3 & 7 & 6 & 2 & 11 \\ 4 & 5 & 9 & 0 & 1 & 8 & 2 & 7 & 11 & 10 & 6 & 3 \\ 8 & 9 & 1 & 4 & 5 & 0 & 6 & 11 & 3 & 2 & 10 & 7 \\ 11 & 10 & 6 & 3 & 2 & 7 & 1 & 8 & 4 & 5 & 9 & 0 \\ 7 & 6 & 2 & 11 & 10 & 3 & 9 & 4 & 0 & 1 & 5 & 8 \\ 3 & 2 & 10 & 7 & 6 & 11 & 5 & 0 & 8 & 9 & 1 & 4 \end{array}\right] . \tag{1–35}$$

By construction, every row of $\mathcal{A}$ *is an* Ω*-transform of* S. *Also by construction, every* $\mathcal{A}_i$ *is an instance of either* A *or* $\mathsf{R}(A)$, *seen as a columnar aggregate, and thus* Ω*-invariant. It follows we can derive a transform of* S *from every* $\mathcal{A}_i$. *Since* $\mathsf{T}_7(S) \in \mathsf{Toc}(\mathcal{A}_1)$ *and* $\mathsf{R}\,\mathsf{T}_0\,\mathsf{I}(S) \in \mathsf{Toc}(\mathcal{A}_2)$, *we get the self-derivation matrix* X.

$$X = \left[\begin{array}{cccccccccccc} & & & & 11 & & 10 & & & & & \\ & & & 4 & & 5 & & & & & & \\ & & 3 & & & & & 2 & & & & \\ & 0 & & & & & & & 1 & & & \\ 8 & & & & & & & & & 9 & & \\ 7 & & & & & & & & & & & 6 \end{array}\right| \left.\begin{array}{cccccccccccc} & & 6 & & & & & 3 & & & & \\ & & & & & 9 & & & & & & 0 \\ & 10 & & & & & & & & & 7 & \\ & & & 5 & & & 8 & & & & & \\ 1 & & & & & & & & 4 & & & \\ & & & & & 2 & & & & & 11 & \end{array}\right] . \tag{1–36}$$

It is important to point out that, although we could extend the matrix X *to a* 6×144 *derivation matrix wherein all columns of* $\mathcal{A}$ *are presented, we could not do so with arbitrary transforms of* S. *Upon inspection, we see that the transforms of* S *we can derive from* $\mathcal{A}_1$ *and* $\mathcal{A}_5$ *are in*

$$\{\mathsf{T}_3, \mathsf{T}_7, \mathsf{T}_{11}, \mathsf{R}\,\mathsf{T}_1, \mathsf{R}\,\mathsf{T}_5, \mathsf{R}\,\mathsf{T}_9, \mathsf{T}_0\,\mathsf{I}, \mathsf{T}_4\,\mathsf{I}, \mathsf{T}_8\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_2\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_6\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_{10}\,\mathsf{I}\} \ . \tag{1–37}$$

For the columns $\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_4$ and $\mathcal{A}_6$, we can derive transforms of S that are in

$$\{\mathrm{T}_1, \mathrm{T}_5, \mathrm{T}_9, \mathrm{R}\,\mathrm{T}_3, \mathrm{R}\,\mathrm{T}_7, \mathrm{R}\,\mathrm{T}_{11}, \mathrm{T}_2\,\mathrm{I}, \mathrm{T}_6\,\mathrm{I}, \mathrm{T}_{10}\,\mathrm{I}, \mathrm{R}\,\mathrm{T}_0\,\mathrm{I}, \mathrm{R}\,\mathrm{T}_4\,\mathrm{I}, \mathrm{R}\,\mathrm{T}_8\,\mathrm{I}\} \ . \tag{1–38}$$

Rather than a hinderance, one could take advantage of this fact by exploring contrasting harmonic regions. Fig. 1-20 shows a musical realization of X:

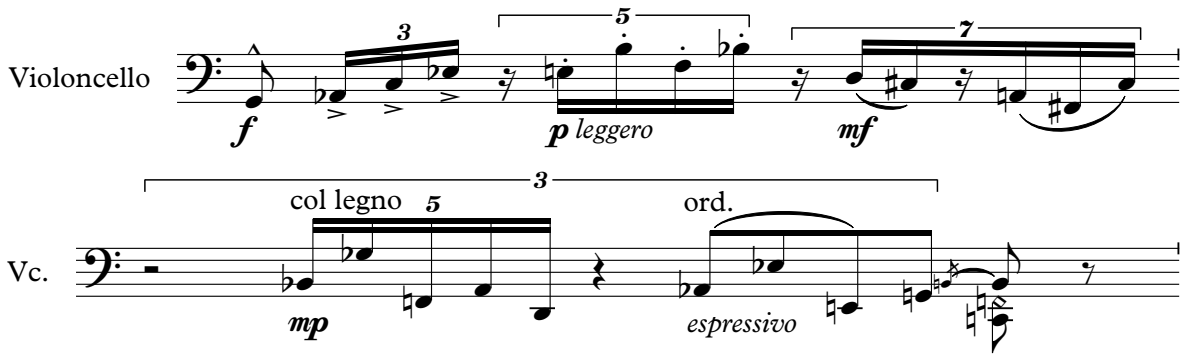


Figure 1-20. Self-derivation in Damiani's *Stingray*.

1.3 The Mallalieu Property

We now consider the $\mathrm{T}_n\,\mathrm{M}\,\mathrm{I}$ class of 12-tone rows with representative

$$S = \{0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6\} \ . \tag{1–39}$$

This series has the remarkable property that, if we include a dummy 13^{th} element, usually represented by an asterisk, then taking every n^{th} element of S produces a transposition of it, so that deriving transforms of a S becomes a mechanical process.

Example 1.3.1. *Put $S^* = \{0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6, *\}$. Then taking every zeroth order number of S^* $\mod 13$ yields trivially S^* itself. Taking every first order number yields the row $\{1, 2, 5, 3, 10, 6, 0, 4, 9, 11, 8, 7, *\}$ which, upon removing the dummy symbol, becomes $\mathrm{T}_1(S)$. Repeating this procedure for every n^{th} order number gives the sequence of T_i-transforms of S where the indices of transposition are totally-ordered, and correspond to the elements of S.*

This most peculiar property, commonly known as the *mallalieu* property, was first discovered by Pohlman Mallalieu [6, 285]. It is natural to ask at this point how many different

12-tone rows are there sharing this property. Unfortunately, there is only one such 12-tone row class under $\mathsf{T}_n\,\mathsf{M}\,\mathsf{I}$. We phrase below a little differently an argument given in [7, 17].

Proposition 1.3.2. *[7, 17] A 12-tone row has the mallalieu property if and only if it is related by* $\mathsf{T}_n\,\mathsf{M}\,\mathsf{I}$ *to the row* $S = \{0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6\}$.

Proof. One direction is just the straightforward check that every $\mathsf{T}_n\,\mathsf{M}\,\mathsf{I}$ transform of S possesses the mallalieu property and is left to the reader. Conversely, if a row R in its untransposed prime form has the mallalieu property, then there is a transposition that takes its order numbers in zeroth rotation, that is, the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ to its order numbers in, say, first rotation, id est, the set $\{1, 3, 5, 7, 9, 11, 0, 2, 4, 6, 8, 10\}$. We can write this transposition as a permutation $0 \mapsto 1, 1 \mapsto 3, 2 \mapsto 5, \cdots, 11 \mapsto 10$, or in cycle notation as $\hat{\mathsf{T}}_k = (0\ 1\ 3\ 7\ 2\ 5\ 11\ 10\ 8\ 4\ 9\ 6)$. Note that $\hat{\mathsf{T}}_k$ is an operation on order numbers. Since $\hat{\mathsf{T}}_k$ corresponds to a transposition, there are only four candidates for T_k, its pitch-class domain counterpart, namely $k \in \{1, 5, 7, 11\}$, because these are the only indices for which a transposition of pitch classes in cycle notation is a 12-cycle. Moreover, we do not need to consider the cases where $k \in \{5, 7, 11\}$, as $\mathsf{T}_5 = \mathsf{M} \circ \mathsf{T}_1$, $\mathsf{T}_7 = \mathsf{M}\,\mathsf{I} \circ \mathsf{T}_1$, and $\mathsf{T}_{11} = \mathsf{I} \circ \mathsf{T}_1$. Hence, without loss, we can set $k = 1$. But then S is the only row in untransposed prime form where T_1 induces the permutation $\hat{\mathsf{T}}_k$ from its order numbers in zeroth rotation to its order numbers in first rotation. To see that, one needs to equate the cycles of $\hat{\mathsf{T}}_k$ with those of T_1. □

A way of looking at the mallalieu property from the standpoint of replacing, for any 12-tone row, its order-number row by the array of integers $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ modulo 13 is provided in [6, 278]. It is easy to see that such an array has the same structure as the array S^* constructed above, if we consider multiplication as the group operation. This is easily seen to be an isomorphism between the integers modulo 12 and the group of units modulo 13. One of the advantages of this approach is that we can dispense with the extra symbol altogether, and just use the indices from 1 to $p - 1$. We shall, however, still refer to the row of order numbers as S^*, the context making it clear whether we are constructing it with an

asterisk or not. The process of taking every n^{th} element of a 12-tone row becomes then just the aforementioned multiplicative group operation on order numbers, that is, multiplying order numbers by $k \pmod{13}$ is the same as taking every k^{th} element of a row.

Example 1.3.3. *Let* $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ *and* S^* *be as above. Then* $\mathrm{M}_3(S^*) = \{3, 6, 9, 12, 2, 5, 8, 11, 1, 4, 7, 10\}$*, which corresponds to the row* $V = \{2, 5, 8, 11, 1, 4, 7, 10, 0, 3, 6, 9\}$*. The row* V *can be equivalently constructed by placing an asterisk at the* 13^{th} *order number of* S*, then taking every third element. However being able to express the process through multiplication, rather than mechanically, greatly facilitates its theoretical description, as well as any algorithmic implementation thereof. The fact that* V *and* S *are not related by* $\mathrm{T}_n\,\mathrm{M}\,\mathrm{I}$ *reflects the fact that neither* S *nor* V *have the mallalieu property.*

It should be of interest to many composers whether other n-TET systems are capable of producing mallalieu rows, and if so, how many. Unfortunately, answering this question is not as straightforward as the above discussion, since we cannot in general rely on the isomorphism that constitutes the proof of 1.3.4. Whenever we can, however, the existence of mallalieu rows is easily verified.

Proposition 1.3.4. *[6, 285] For* p *a prime, every* $(p-1)$*-TET system is capable of producing a mallalieu row.*

Proof. For every prime p, the group of units modulo p is isomorphic to $\mathbb{Z}/(p-1)\mathbb{Z}$. The mallalieu property in these cases can be seen as the aforementioned isomorphism, where $\mathbb{Z}/(p-1)\mathbb{Z}$ is the group of transpositions of a row, and $(\mathbb{Z}/p\mathbb{Z})^\times$ is its multiplicative group on order numbers. The number of mallalieu rows in each $(p-1)$-TET system is then the number of isomorphisms $\mathbb{Z}/(p-1)\mathbb{Z} \to (\mathbb{Z}/p\mathbb{Z})^\times$, that is, the order of the group of automorphisms of $\mathbb{Z}/(p-1)\mathbb{Z}$. Since for every prime p we have $|\operatorname{Aut}(\mathbb{Z}/(p-1)\mathbb{Z})| \geq 1$, every $(p-1)$-TET system is capable of producing a mallalieu row, as desired. □

In face of 1.3.4, 1.3.2 becomes just the special case where $p = 13$, as demonstrated in the next example.

Example 1.3.5. *[8, 8] [9, 9] The number of isomorphisms* $\mathbb{Z}/12\mathbb{Z} \to (\mathbb{Z}/13\mathbb{Z})^\times$ *is equal to*

$$|\operatorname{Aut}\left((\mathbb{Z}/12\mathbb{Z})^\times\right)| = 4 \ . \tag{1–40}$$

We can construct these isomorphisms by mapping a generator of $\mathbb{Z}/12\mathbb{Z}$*, say* $\overline{1}$*, to the generators of* $(\mathbb{Z}/13\mathbb{Z})^\times$*, namely* $\overline{2}, \overline{6}, \overline{7}$ *and* $\overline{11}$*. Explicitly, we get the four maps* $i \pmod{12} \mapsto 2^i \pmod{13}$*,* $i \pmod{12} \mapsto 6^i \pmod{13}$*,* $i \pmod{12} \mapsto 7^i \pmod{13}$*, and* $i \pmod{12} \mapsto 11^i \pmod{13}$*. We leave the verification that these maps are well defined and bijective to the reader. Denote the first map by* φ*. Then*

$$\varphi(a+b) = 2^{a+b} = 2^a \cdot 2^b = \varphi(a) \cdot \varphi(b) \ , \tag{1–41}$$

so φ *is an isomorphism. The verification that the other three maps are isomorphisms is identical. Define* $\varphi^{-1} : (\mathbb{Z}/13\mathbb{Z})^\times \to \mathbb{Z}/12\mathbb{Z}$ *by* $\varphi^{-1}(\log i \pmod{13}) = i \pmod{12}$*. Then* φ^{-1} *is easily seen to be the inverse of* φ*. Let* $S^* = \{1, 2, \dots, 11, 12\}$ *be a series of order numbers written multiplicatively. Then*

$$\varphi^{-1}(S^*) = \{\log 1, \log 2, \dots, \log 12\} \pmod{13} = \{0, 1, 4, \dots, 7, 6\} \ , \tag{1–42}$$

which by 1.3.2 is one of the untransposed 12-tone rows that have the mallalieu property.

We conclude this section by exposing some well-known isomorphisms between groups of pitch-class operations and abstract groups.

Proposition 1.3.6. *[10, 127] We have the following isomorphisms:*

$$\langle \mathsf{T} \rangle \cong C_{12} \tag{1–43}$$

$$\langle \mathsf{T}, \mathsf{I} \rangle \cong D_{24} \tag{1–44}$$

$$\langle \mathsf{I}, \mathsf{R} \rangle \cong V_4 \tag{1–45}$$

$$\langle \mathsf{T}, \mathsf{I}, \mathsf{R} \rangle \cong D_{24} \oplus \mathbb{F}_2 \tag{1–46}$$

$$\langle \mathsf{T}, \mathsf{I}, \mathsf{M} \rangle \cong \mathsf{Aff}_1(C_{12}) \tag{1–47}$$

$$\langle \mathsf{T}, \mathsf{I}, \mathsf{M}, \mathsf{R} \rangle \cong \mathsf{Aff}_1(C_{12}) \oplus \mathbb{F}_2 \tag{1–48}$$

1.4 Final Remarks

We now devote our attention to summarizing the theoretical objectives of this research. The main motivation is to understand the construction of self-derivation precisely, that is, for any given row, whether it is capable of producing any known form of self-derivation. Conversely, given some self-derivation procedure, we would like to be able to provide a complete description of the class of rows that participate therein. All the above shall be done in all generality, that is, for arbitrary n-tone temperament systems. After having dealt with the more constrained case of self-derivation, we shall attempt to extend the above ideas to the general derivation setting, id est, without the requirement that the derived row be in the same row class as the originating row. General derivation is less strict and better understood than self-derivation, but lacks the latter's potential for cohesiveness. Whatever is already present in the literature, shall be extended to other equal temperament systems than the 12-TET. We shall also attempt to describe how specific patterns of derivation arise. In other words, given a pattern of order numbers, for what rows, and under which operations, do we get known forms of derivation and self-derivation. Analogously, we would like to investigate how chains of pitch-class operations are induced by derivation procedures, as to determine how a piece may be composed from the standpoint of its syntax, perhaps even prior to establishing any compositional surface. Many well-known derivation procedures described in this chapter also lack a more rigorous treatment. We shall make precise, providing proofs when necessary, what the constraints regarding the choice of operation are for folded and shifted derivations. In particular, we will investigate the need for commutativity, as well as determine which of the aforementioned forms support derivation matrices that do not involve the retrograde. We shall also take into account the role of the cyclic rotation operator, particularly in what regards folded self-derivations. Cyclic rotation and its invariances will be intimately connected with our combinatorial findings (in the mathematical sense). We shall make use of known brute-force algorithms, such as [5], to confront our findings, and also verify their correctness if appropriate. Finally, we will generalize mallalieu-type rows to arbitrary equal temperament systems, giving

a precise count and description of their untransposed representatives. In addition, we will investigate generalizations of the mallalieu property by relaxing the requirement that we get a copy of the row at every single index, as proposed in [11]. We shall also investigate whether transposition is the only operation that affords the mallalieu property.

Our theoretical framework will be purely mathematical, with the sole exception of a few brute-force algorithms that will be used to guide our findings. As previously discussed, mathematics can greatly simplify our search for answers, and a solid mathematical theory of the musical objects that pertain to this study are a requirement for mature compositional decisions that involve the techniques here discussed. Anticipating, as much as possible, the symmetries of a musical construct of the sort with which we will be concerned can help a composer determine the outlook and feasibility of an entire piece. Mathematics is also crucial for constructing efficient computer algorithms that employ such techniques, and is the only resource we actually have against brute force. Much of the mathematics required by our findings will be mentioned in the theoretical framework, although some basic knowledge is omitted. We assume the reader will have some basic knowledge of abstract algebra, to the extend that many graduate texts in atonal music theory will require. Our treatment shall depart from the concept of group actions. A working knowledge of twelve-tone theory is also required. We assume the reader is familiar with such basic concepts as contour, pitch, and pitch-class spaces, intervals and interval classes, set classes, operations, and associated group-theoretic notions.

The last chapter of the present work will deal with some musical applications of derivation techniques. Given the historical application of these techniques to almost exclusively the pitch domain, and in the interest of shedding new light onto such procedures, the compositional applications we shall describe will have a certain bias toward employing derivation in dimensions other than pitch. Also, given that most examples of derivation in the twentieth-century repertoire are of instrumental music, we shall here pursue the electroacoustic music avenue, with a particular interest in algorithmic processes. Beyond uniqueness and innovation, we justify and

motivate our choices by the fact that many of our generalizations, such as, say, constructing a mallalieu row in 50 elements, become problematic, if not incompatible, with instrumental music.

CHAPTER 2
LITERATURE REVIEW

2.1 Discovering Derivation

The beginnings of derivation techniques can be traced back to Donald Martino's *The Source Set and Its Aggregate Formations* [12], a paper whose main purpose is another, namely to generalize the construction of columnar realizations, although this latter term appears to have been coined by [1]. The techniques employed by Martino are heavily influenced by Babbitt's ideas on combinatoriality [12, 224], and make extensive use of tables. Below is a fragment of the source hexachords table given in [12, 229]. The author, in the interest of generalizing the procedure, provides also trichordal, tetrachordal, and even pentachordal combinatoriality tables, as well as a somewhat brief discussion on uneven partitions of a row and their combinatoriality implications [12, 267].

Table 2-1. Fragment of a table for consulting source hexachords in [12, 229].

No.	Set	Interval Vector	TTO's
A1	$\{0,1,2,3,4,5\}$	$\langle 5,4,3,2,1,0\rangle$	$\{T_6, T_{11}I, RT_{11}, RT_6I\}$
B2	$\{0,2,3,4,5,7\}$	$\langle 3,4,3,2,3,0\rangle$	$\{T_6, T_1I, RT_1, RT_6I\}$
3	$\{0,1,3,4,5,8\}$	$\langle 3,2,3,4,3,0\rangle$	$\{T_6, RT_2\}$
E4	$\{0,1,4,5,8,9\}$	$\langle 3,0,3,6,3,0\rangle$	$\{T_{2,6,10}, T_{3,7,11}I, RT_{3,7,11}, RT_{2,6,10}I\}$
C5	$\{0,2,4,5,7,9\}$	$\langle 1,4,3,2,5,0\rangle$	$\{T_6, T_3I, RT_3, RT_6I\}$

Martino acknowledges the aggregates formed by combinatoriality are not necessarily ordered [12, 228], rather seeing it from the bright side of diversity, in which the set union of a columnar aggregate being capable of *deriving* a different row class is actually more appealing than self-similarity, which is not at all considered. The same passage brings, to our knowledge, the first *explicit* mention in the literature of a technique for deriving new rows from columnar

aggregates in more generality than what we see in Schoenberg's fourth string quartet, as seen in 2.1.1. In mentioning derivation, Martino juxtaposes it to another, in his view, way of progressing through hexachords, namely the *fragmentation* or partitioning of the original series, ultimately deeming both procedures essentially the same, as derivation via aggregate realizations can surely be seen as the fragmentation of the (new) series obtained vertically.

Example 2.1.1. *[12, 230] Let* $S = \{0, 4, 11, 3, 1, 2, 5, 6, 9, 8, 10, 7\}$ *and consider the trivial combination matrix given below.*

$$\hat{A} = [\hat{A}_1|\hat{A}_2|\hat{A}_3|\hat{A}_4] = \left[\begin{array}{ccc|ccc|ccc|ccc} 0 & 4 & 11 & 3 & 1 & 2 & 5 & 6 & 9 & 8 & 10 & 7 \\ 7 & 10 & 8 & 9 & 6 & 5 & 2 & 1 & 3 & 11 & 4 & 0 \end{array}\right] . \tag{2–1}$$

In particular, the hexachords given by $\hat{A}_i$ *can be combined with their transforms under* $\mathsf{T}_1\,\mathsf{I}$*, yielding the following matrix:*

$$A = [A_1|A_2|A_3|A_4] = \left[\begin{array}{ccc|ccc|ccc|ccc} 0 & 4 & 11 & 3 & 1 & 2 & 5 & 6 & 9 & 8 & 10 & 7 \\ 7 & 10 & 8 & 9 & 6 & 5 & 2 & 1 & 3 & 11 & 4 & 0 \\ \hline 1 & 9 & 2 & 10 & 0 & 11 & 8 & 7 & 4 & 5 & 3 & 6 \\ 6 & 3 & 5 & 4 & 7 & 8 & 11 & 0 & 10 & 2 & 9 & 1 \end{array}\right] . \tag{2–2}$$

We can then **almost** *derive transforms of the row* $Q = \{7, 0, 10, 6, 4, 1, 3, 5, 8, 9, 2, 11\}$ *from the columns of* A*, except for a single symmetry between 6 and 5, as seen below.*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} & 0 & & & & 4 & & & & & & 11 & & & & 3 & 1 & & & 2 & & & & \\ 7 & & & 10 & & & & & 8 & & & & & 9 & & & & & & & & \boxed{6} & \boxed{5} & \\ \hline & & & & & & 1 & & & 9 & 2 & & & & & & & & 10 & 0 & & & & 11 \\ & & & & 6 & & & 3 & 5 & & & & 4 & & 7 & & & & & & & & & 8 \end{array}\right] \ldots \tag{2–3}$$

This apparent failure did not seem to bother Martino at all, as his focus remained in establishing a concrete foundation for combinatoriality. In order to extend the above procedure to eight rows of counterpoint, some adjustments must be made, namely we need columnar aggregates whose rows are allowed to have different numbers of elements, as seen below. This second

folding is otherwise accomplished much the same way, by choosing a hexachord from A_1, and determining its combinatoriality properties via table lookup.

$$\left[\begin{array}{cc|cc|cc|cc} 0 & 4 & 11 & & 3 & 1 & 2 & \\ 7 & 10 & 8 & & 9 & & 6 & 5 \\ 1 & & 9 & 2 & 10 & & 0 & 11 \\ 6 & & 3 & 5 & 4 & 7 & 8 & \\ 2 & & 6 & 1 & 5 & & 3 & 4 \\ 9 & & 0 & 10 & 11 & 8 & 7 & \\ 3 & 11 & 4 & & 0 & 2 & 1 & \\ 8 & 5 & 7 & & 6 & & 9 & 10 \end{array}\right] \cdots \tag{2–4}$$

It is easy to understand why Martino was so motivated by creating as much diversity as possible from a single row in a structured manner. If not because creating variation upon some fixed foundation has been a constant in music composition since Bach, serialism up to that point had seen countless pieces in which the very same series was presented over and over again, frequently in the same $\mathrm{R}\,\mathrm{T}_n\,\mathrm{I}$ form for entire passages. As innovative as it is, not even the first movement of Schoenberg's fourth string quartet escapes this paradigm. The idea of having a systematic approach to syntactically moving from one row class to another, although latent in late Schoenberg, ultimately defines one one of the most remarkable trends in the next chapter of serialism. Martino actually manages to solve both problems at once with derivation, as it gives the composer the opportunity to focus on the derived rows, while using a somewhat more abstract row to generate syntax, even though we only see this potential explored in his later pieces, and not quite yet in [12]. For now, what is most important is to derive as much harmonic diversity from the rigidity of an omnipresent series as possible.

What [1] terms *skewed polyphonization*, and what we called shifted derivation in Sec. 1.2.2, has too an origin in what Martino calls, in turn, *oblique combination*. Ex. 2.1.2 deals with the tetrachordal case, although other cases, including those where the overlap occurs at unequal partitionings of the row are considered. Similarly to our intuition in Sec. 1.2.2,

oblique combinations are seen as special, yet straightforward cases of other types of combinatoriality [12, 267].

Example 2.1.2. *[12, 241] Let $S = S_1|S_2|S_3$ be a twelve-tone row, partitioned into tetrachords. Let f and g be TTO's, and considered the following oblique combination matrix:*

$$\left[\begin{array}{c|c|c|c|c} S_1 & S_2 & S_3 & . & . \\ . & f(S_3) & f(S_2) & f(S_1) & . \\ . & . & g(S_1) & g(S_2) & g(S_3) \end{array}\right] . \tag{2–5}$$

It follows:

i. If $S_1 = \mathrm{T}_i\,\mathrm{I}(S_1)$ **only** *for some i, then $f = \mathrm{R}\,\mathrm{T}_j$ for some j, and $g = \mathrm{T}_k\,\mathrm{I}$ for some k;*

ii. If $S_3 = \mathrm{T}_i\,\mathrm{I}(S_3)$ **only** *for some i, then $f = \mathrm{R}\,\mathrm{T}_j\,\mathrm{I}$ for some j, and $g = \mathrm{T}_k$ for some k;*

iii. If $S_1 = \mathrm{T}_i\,\mathrm{I}(S_1)$ **only** *for some i and $S_3 = \mathrm{T}_j\,\mathrm{I}(S_3)$ for some j, then $f = \mathrm{R}\,\mathrm{T}_k\,\mathrm{I}$ for some k, and $g = \mathrm{T}_\ell\,\mathrm{I}$ for some ℓ;*

iv. If $S_1 = \mathrm{T}_i\,\mathrm{I}(S_1)$ **only** *for some i, $S_3 = \mathrm{T}_j\,\mathrm{I}(S_3)$ for some j, and $S_1 = \mathrm{T}_k(S_3)$ for some k, then $f = \mathrm{T}_\ell\,\mathrm{I}$ or $f = \mathrm{R}\,\mathrm{T}_\ell\,\mathrm{I}$ for some ℓ, and $g = \mathrm{T}_m\,\mathrm{I}$ or $g = \mathrm{R}\,\mathrm{T}_m\,\mathrm{I}$ for some m;*

v. If $S_1 = \mathrm{T}_i(S_3)$ **only** *for some i, then $f = \mathrm{T}_j$ or $f = \mathrm{R}\,\mathrm{T}_j$ for some j, and $g = \mathrm{T}_k$ or $g = \mathrm{R}\,\mathrm{T}_k$ for some k;*

vi. If $S_3 = \mathrm{T}_i\,\mathrm{I}(S_1)$ **only** *for some i, then $f = \mathrm{T}_j\,\mathrm{I}$ or $f = \mathrm{R}\,\mathrm{T}_j$ for some j, and $g = \mathrm{T}_k$ or $g = \mathrm{R}\,\mathrm{T}_k\,\mathrm{I}$ for some k.*

CHAPTER 3
THEORETICAL FRAMEWORK

3.1 Group Actions

Definition 3.1.1. *[13, 99] [14, 41] Let X be a set and G a group. An* **action** *of G on X is a function $G \times X \to X$ given by $(g, x) \mapsto gx$, such that:*

i. $(gh)x = g(hx)$ *for all $g, h \in G$ and $x \in X$;*

ii. $1x = x$ *for all $x \in X$, where $1 \in G$ is the identity.*

Proposition 3.1.2. *[13, 99] [14, 42] If a group G acts on a set X then, for every $g \in G$, the function $f_g : X \to X$ given by $f_g(x) = gx$ is a permutation of X. Further, the function $f : G \to S_X$ given by $f(g) = f_g$ is a homomorphism and, conversely, for any homomorphism $\phi : G \to S_X$, there is a corresponding group action given by $\phi(g)(x)$.*

Theorem 3.1.3 (Cayley)**.** *[13, 96] [14, 120] Every group is isomorphic to a subgroup of the symmetric group S_G. In particular, if $|G| = n$, then G is isomorphic to a subgroup of S_n.*

Theorem 3.1.4. *[13, 97] Let $H \leq G$ be a subgroup of finite index n. Then there exists a homomorphism $\phi : G \to S_n$ such that $\ker \phi \leq H$. In particular, when $H = \{1\}$, we get Cayley's theorem.*

Example 3.1.5. *[14, 122] A group acts on itself by conjugation. Let $g, h \in G$ and $x \in G$. Then $1x1^{-1} = x$ and*

$$g(hx) = g(hxh^{-1}) = ghxh^{-1}g^{-1} = (gh)x(gh)^{-1} = (gh)x \ . \tag{3–1}$$

It is also immediate from the above that a group acts on its power set by conjugation. In particular, a group acts on the set of all its subgroups.

Definition 3.1.6. *[13, 100] [14, 112] If G acts on X, then the* **orbit** *of $x \in X$ is the set*

$$\mathcal{O}(x) = \{gx : g \in G\} \subseteq X \ . \tag{3–2}$$

We say an action is **transitive** *if there is only one orbit. The* **kernel** *of the action is the set*

$$\{g \in g : gx = x, \forall x \in X\} \ . \tag{3–3}$$

We say an action is **faithful** *if the kernel is the identity. The* **stabilizer** *of x in G is the group*

$$G_x = \{g \in g : gx = x\} \leq G \ . \tag{3–4}$$

When a group acts on itself by conjugation, we call the orbits **conjugacy classes**. *The stabilizer of some $g \in G$ is the* **centralizer** *of g in G, denoted $C_G(g)$. When a group acts on the set of its subgroups by conjugation, the stabilizer of a subgroup $H \leq$ is the* **normalizer** *of H in G, denoted by $N_G(H)$.*

Proposition 3.1.7. *[13, 102] [14, 114] [15, 250] If G acts on X, for $x_1, x_2 \in X$, the relation $x_1 \sim x_2$ given by $x_1 = gx_2$ is an equivalence relation. It follows immediately that the equivalence classes are the orbits of the action of G on X and that*

$$|X| = \sum_i |\mathcal{O}(x_i)| \ , \tag{3–5}$$

where x_i is a single representative from each orbit.

Theorem 3.1.8 (Orbit-Stabilizer)**.** *[13, 102] If G acts on X, then for each $x \in X$*

$$|\mathcal{O}(x)| = [G : G_x] \ . \tag{3–6}$$

Corollary 3.1.9. *[13, 103] If G is finite and acts on X, then the size of any orbit is a divisor of $|G|$.*

Proposition 3.1.10. *[14, 123] The number of conjugates of a subset S in a group G is $|G : N_G(S)|$, the index of the normalizer of S. In particular, the number of conjugates of an element s is $|G : C_G(s)|$, the index of the centralizer of s.*

Proposition 3.1.11. *[14, 125] Let $\sigma, \tau \in S_n$. If*

$$\sigma = (a_1\ a_2\ \cdots a_j)(b_1\ b_2\ \cdots a_k)\cdots \ , \tag{3–7}$$

then

$$\tau\sigma\tau^{-1} = (\tau(a_1)\ \tau(a_2)\ \cdots \tau(a_j))(\tau(b_1)\ \tau(b_2)\ \cdots \tau(b_k))\cdots \ . \tag{3–8}$$

Example 3.1.12. *[14, 127] Let $\sigma \in S_n$ be an m-cycle. The number of conjugates of σ is*

$$\frac{|S_n|}{|C_{S_n}(\sigma)|} = \frac{n(n-1)(n-m+1)}{m} \ , \tag{3–9}$$

so that $|C_{S_n}(\sigma)| = m(n-m)!$. Since σ commutes with its powers, and also with any permutation in S_n whose cycles are disjoint from it, and there are $(n-m)!$ of those, the number computed above is the full centralizer of σ.

Example 3.1.13. *[14, 132] The size of each conjugacy class in S_n is*

$$\frac{n!}{\prod_r r^{n_r} n_r!} \ , \tag{3–10}$$

where, for each r-cycle, we divide by r to account for the cyclical permutations of elements within a cycle. Further, if there are n_r cycles of length r, we divide by $n_r!$ to account for the different orders in which those cycles may appear.

Proposition 3.1.14. *[14, 126] Two elements in S_n are conjugate if and only if they have the same cycle type. The number of conjugacy classes of S_n is the number of partitions of n.*

Definition 3.1.15. *[14, 133] An isomorphism from a group G to itself is called an* **automorphism** *of G. The group under composition of all automorphisms of G is denoted by* $\mathrm{Aut}(G)$.

Proposition 3.1.16. *[14, 133] If H is a normal subgroup of G, then the action of G by conjugation on H is, for each $g \in G$, an automorphism of H. The kernel of the action is $C_G(H)$. In particular, $G/c_G(H)$ is a subgroup of* $\mathrm{Aut}(H)$.

Corollary 3.1.17. *[14, 134] For any subgroup $H < G$ and $g \in G$, $H \cong gHg^{-1}$. Moreover, $N_G(H)/C_G(H)$ (and also $G/Z(G)$ when $G = H$) is isomorphic to a subgroup of* $\mathrm{Aut}(H)$.

Definition 3.1.18. *[14, 135] A subgroup $H < G$ is called* **characteristic** *if every automorphism of G maps H to itself.*

Proposition 3.1.19. *[14, 135] Characteristic subgroups are normal. Unique subgroups of a given order are characteristic. A characteristic subgroup of a normal subgroup is normal. In particular, every subgroup of a cyclic group is characteristic.*

Proposition 3.1.20. *[14, 135] If G is cyclic of order n, then $\mathrm{Aut}(G) \cong (\mathbb{Z}/n\mathbb{Z})^\times$, and $|\mathrm{Aut}(G)| = \varphi(n)$, where φ is Euler's totient function.*

Corollary 3.1.21. *[14, 136] Let $|G| = pq$, with $p \leq q$ primes. If $p \nmid q - 1$, then G is abelian. If, further, $p < q$, then G is cyclic.*

Proposition 3.1.22. *[14, 136] If $|G| = p^n$, with p an odd prime, then*

$$|\mathrm{Aut}(G)| = p^{n-1}(p-1) \tag{3–11}$$

and the latter group is also cyclic. If $|G| = 2^n$ is cyclic, with $n \geq 3$, then

$$\mathrm{Aut}(G) \cong \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2^{n-2}\mathbb{Z} \tag{3–12}$$

and the latter is not cyclic, but has a cyclic subgroup of index 2. If V is the elementary abelian group of order p^n, then $pv = 0$ for all $v \in V$ and V is an n-dimensional vector space over the field $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$. The automorphisms of V are the nonsingular linear transformations from V to itself:

$$\mathrm{Aut}(V) \cong GL(V) \cong GL_n(\mathbb{F}_p) \ , \tag{3–13}$$

and if F is the finite field of order q, then

$$|GL_n(F)| = (q^n - 1)(q^n - q)(q^n - q^2) \cdots (q^n - q^{n-1}) \ . \tag{3–14}$$

For all $n \neq 6$, we have $\mathrm{Aut}(S_n) \cong S_n$. Finally, $\mathrm{Aut}(D_8) \cong D_8$ and $\mathrm{Aut}(Q_8) \cong S_4$.

Example 3.1.23. *[14, 137] The Klein 4-group is the elementary abelian group of order 4. It follows $\mathrm{Aut}(V_4) \cong GL_2(\mathbb{F}_2)$, and $|\mathrm{Aut}(V_4)| = 6$. Since the action of $\mathrm{Aut}(V_4)$ on V_4 permutes the latter's 3 nonidentity elements, by order considerations we have*

$$\mathrm{Aut}(V_4) \cong GL_2(\mathbb{F}_2) \cong S_3 \ . \tag{3–15}$$

Proposition 3.1.24. *[14, 314] A finite subgroup of the multiplicative group of a field is cyclic. In particular, if F is a finite field, then $F^\times$ is cyclic.*

Corollary 3.1.25. *[14, 314] For p a prime, $(\mathbb{Z}/p\mathbb{Z})^\times$ is cyclic.*

3.2 Polya's Enumeration Formula

Definition 3.2.1. *[16, 85] Let N be a set of n beads and R be a set of r colors. A colored necklace is a function $f : N \to R$. Denote the set of all such functions by R^N, so that $|R^N| = r^n$. Define the* **weight** *associated to a coloring f by*

$$w(f) = \prod_{i \in N} x_{f(i)} \ , \tag{3–16}$$

where x_j is a variable associated with the color $j \in R$.

Proposition 3.2.2. *[13, 110] Let G be a group and $X = \{1, \cdots, n\}$ be a set. Let $\mathcal{C}$ be a set of q colors. Then G acts on the set $\mathcal{C}^n$ of n-tuples of colors by*

$$\tau(c_1, \cdots, c_n) = (c_{\tau 1}, \cdots, c_{\tau n}), \forall \tau \in G \ . \tag{3–17}$$

Proposition 3.2.3. *[16, 85] [13, 110] If G is a group acting on the set N, then two colorings f and f' are equivalent whenever $f = f' \circ g$ for some $g \in G$. This is an equivalence relation that partitions R^N into equivalence classes denoted* **patterns***. Under the conditions of 3.2.2, an orbit $(c_1, \cdots, c_n) \in \mathcal{C}^n$ is called a (q, G)-***coloring** *of X. Moreover, two equivalent colorings have the same weight, so that we may refer to the weight of a class of colorings rather than the weights of its representatives.*

Definition 3.2.4. *[16, 85] Let N be a set of n beads and R be a set of r colors. Let G be a group acting on the set N and x_j the variable associated with the color $j \in R$. Let $\mathcal{M}$ be the set of of patterns. Define the pattern* **enumerator** *by*

$$w(R^N, G) = \sum_{M \in \mathcal{M}} w(M) \ . \tag{3–18}$$

In particular, when $x_j = 1$ for all $j \in R$, we get $w(R^N, G) = |\mathcal{M}|$.

Definition 3.2.5. *[16, 86] Let G be a group acting on a set X. Define for every $g \in G$ its* **fixed-point set** *by*

$$\mathrm{Fix}(g) = \{x \in X : gx = x\} \ . \tag{3–19}$$

Lemma 3.2.6. *[13, 112] Let $G < S_n$ be a group and let $\mathcal{C}$ be a set of q colors. For $\tau \in G$,*

$$|\operatorname{Fix}(\tau)| = q^{t(\tau)} \ , \tag{3–20}$$

where $t(\tau)$ is the number of cycles in the complete factorization of τ.

Lemma 3.2.7 (Burnside)**.** *[13, 109] [15, 251] If G acts on a finite set X, then the number of orbits N is*

$$N = \frac{1}{|G|} \sum_{\tau \in G} |\operatorname{Fix}(\tau)| \ . \tag{3–21}$$

Corollary 3.2.8. *[13, 112] Let G be a group acting on a finite set X. The number N of (q, G)-colorings of X is*

$$N = \frac{1}{|G|} \sum_{\tau \in G} q^{t(\tau)} \ . \tag{3–22}$$

Corollary 3.2.9. *[17, 54] [10, 127] The number of n-tone rows under $\mathrm{RT}_n\mathrm{I}$ is*

$$\begin{cases} \frac{1}{4}\left[(n-1)! + (n-1)(n-3)\cdots(2)\right] & n \text{ odd} \\ \frac{1}{4}\left[(n-1)! + (n-2)(n-4)\cdots(2)(1+\frac{n}{2})\right] & n \text{ even} \end{cases} \tag{3–23}$$

In particular, there are 9985920 twelve-tone rows.

Definition 3.2.10. *[16, 87] Let g be a group acting on R^N. Define the* **cycle indicator** *of G by*

$$P_G(z_1, z_2, \cdots, z_n) = \frac{1}{|G|} \sum_{g \in G} z_1^{b_1(g)} z_2^{b_2(g)} \cdots z_n^{b_n(g)} \ , \tag{3–24}$$

where $z_i^{b_i(g)}$ corresponds to the number of cycles in the complete factorization of g that have length i.

Example 3.2.11. *The cycle indicator of S_3 is*

$$P_{S_3}(x_1, x_2, x_3) = \frac{1}{6}(x_1^3 + 3x_1^1 x_2^1 + 2x_3^1) \ , \tag{3–25}$$

since there are two elements in S_3 that comprise one cycle of length three, three elements that comprise one cycle of length one, and one cycle of length two, and one element that comprises three cycles of length one.

Example 3.2.12. *Let C_n be the cyclic group of order n. Then*

$$P_{C_n} = \frac{1}{n}\sum_{d|n}\varphi(d)z_d^{n/d} \quad . \tag{3–26}$$

Example 3.2.13. *For S_n we have*

$$P_{S_n} = \sum_{j_1+2j_2+\cdots+nj_n=n}\frac{1}{\prod_{k=1}^{n}k^{j_k}j_k!}\prod_{k=1}^{n}a_k^{j_k} \quad . \tag{3–27}$$

In words, there is a summand for each conjugacy class in S_n, and we divide each summand by the size of its corresponding conjugacy class.

Example 3.2.14. *For D_n we have*

$$P_{D_n} = \frac{1}{2}P_{C_n} + \begin{cases} \frac{1}{2}a_1a_2^{(n-1)/2} & n \text{ odd} \\ \frac{1}{4}(a_1^2a_2^{(n-2)/2} + a_2^{n/2}) & n \text{ even} \end{cases} \tag{3–28}$$

In particular, we obtain

$$P_{D_{24}}(1+x,\cdots,1+x^{12}) = \frac{1}{24}(x_1^{12} + 6x_1^2x_2^5 + 7x_2^6 + 2x_3^4 + 2x_4^3 + 2x_6^2 + 4x_{12}) \quad . \tag{3–29}$$

Example 3.2.15. *[10, 120] For $\mathrm{Aff}_1(C_{12})$ we have*

$$P_{\mathrm{Aff}_1(C_{12})}(1+x,\cdots,1+x^{12}) = \frac{1}{48}(x_1^{12} + 2x_1^6x_2^3 + 3x_1^4x_2^4 + 6x_1^2x_2^5 + 12x_2^6 + 4x_2^3x_6 + 2x_3^4 + 8x_4^3 + 6x_6^2 + 4x_{12}) \quad . \tag{3–30}$$

Theorem 3.2.16 (Polya)**.** *[16, 88] [15, 256] Let N be a set of cardinality n and R be a set of cardinality r. Let G be a group acting on the set N and x_j be arbitrary variables with $j \in R$. Let $w(R^N, G)$ be the pattern enumerator for the action of G on R^N. Then*

$$w(R^N, G) = \sum_{M\in\mathcal{M}} w(M) = P_G(\sum_{j\in R}x_j, \sum_{j\in R}x_j^2, \cdots, \sum_{j\in R}x_j^n) \quad . \tag{3–31}$$

Corollary 3.2.17. *[16, 89] [15, 254] Let $x_j = 1$ for all $j \in R$. Then $\sum_{j\in R} x_j^k = |R| = r$ for all k, hence*

$$P_G(r, \cdots, r) = |\mathcal{M}| \ . \tag{3–32}$$

Corollary 3.2.18. *[16, 89] Let $|R| = r = 2$. Let $x_1 = x_{white} = x$ and $x_2 = x_{black} = 1$. Then*

$$P_G(x+1, x^2+1, \cdots, x^n+1) = \sum_{k=0}^{n} a_k x^k \ , \tag{3–33}$$

where a_k is the number of patterns in which the color white occurs exactly k times.

Example 3.2.19. *[16, 89] The number of necklaces with n beads and r colors is*

$$\frac{1}{n}\sum_{d|n} \varphi(d) r_d^{n/d} \ . \tag{3–34}$$

Example 3.2.20. *[16, 86] The definition of a weight function is useful in order to count how many necklaces contain precisely x_j beads of color j. Set W to be the color white and B to be the color black. Then a 4-bead necklace with exactly 2 white and 2 black beads is expressed by W^2B^2. If we do not wish to account for a particular color, per 3.2.18 we may assign its weight to 1. If we chose not to account for the color black, say, then the representation above would become simply $W^2 1^2 = W^2$.*

Example 3.2.21. *We use Polya's theorem to describe the pattern inventory of 3-bead necklaces under the action of S_3. Let the colors be A and B. Then*

$$P_{S_3}(A+B, A^2+B^2, A^3+B^3) = \frac{1}{6}[(A+B)^3 + 3(A+B)(A^2+B^2) + 2(A^3+B^3)] \tag{3–35}$$

$$= A^3 + A^2B + AB^2 + B^3 \ . \tag{3–36}$$

In words, we have one necklace with three A beads, one with two A and one B bead, one with one A and two B beads, and one with three B beads.

Example 3.2.22. *We can use Polya's theorem to count chords in 12 tones that are equivalent under transposition by setting our set of colors to be $R = \{r_0, r_1\}$. We then disregard one of the colors, say r_0, by setting $w(r_0) = 1$ and $w(r_1) = C$. The group of transpositions is just*

$C_{12} = \mathbb{Z}/12\mathbb{Z}$, *so we get*

$$P_{C_{12}}(1+x,\cdots,1+x^{12}) = \frac{1}{12}\sum_{d\in\{1,2,3,4,6,12\}} \varphi(d)x_d^{12/d} \tag{3–37}$$

$$= \frac{1}{12}\left(x_1^{12}+x_2^6+2x_3^4+2x_4^3+2x_6^2+4x_{12}\right) \tag{3–38}$$

$$= \frac{1}{12}\left[(1+C)^{12}+(1+C^2)^6+2(1+C^3)^4\right. \tag{3–39}$$

$$\left. +\,2(1+C^4)^3+2(1+C^6)^2+4(1+C^{12})\right] \tag{3–40}$$

$$= C^{12}+C^{11}+\cdots \tag{3–41}$$

$$\cdots+80C^6+66C^5+43C^4+19C^3+6C^2+C^1+C^0\ . \tag{3–42}$$

In particular, there are 43 tetrachords and 19 trichords that are transpositionally equivalent.

Example 3.2.23. *[17, 53] More generally, the number of k-chords under the action of C_n is*

$$\frac{1}{n}\sum_{j|(n,k)}\varphi(j)\binom{n/j}{k/j}\ . \tag{3–43}$$

Under the action of the dihedral group D_{2n}, we obtain

$$\begin{cases} \frac{1}{2n}\left[\sum_{j|(n,k)}\varphi(j)\binom{n/j}{k/j}+n\binom{(n-1)/2}{k/2}\right] & n \text{ odd} \\ \frac{1}{2n}\left[\sum_{j|(n,k)}\varphi(j)\binom{n/j}{k/j}+n\binom{n/2}{k/2}\right] & n \text{ even and } k \text{ even} \\ \frac{1}{2n}\left[\sum_{j|(n,k)}\varphi(j)\binom{n/j}{k/j}+n\binom{(n/2)-1}{k/2}\right] & n \text{ even and } k \text{ odd} \end{cases} \tag{3–44}$$

In particular, the pattern inventory of k-chords in 12 tones that are equivalent under $\mathsf{T}_n\mathsf{I}$ is

$$P_{D_{24}}(1+x,\cdots,1+x^{12}) = C^{12}+\cdots+50C^6+38C^5+29C^4+12C^3+6C^2+C^1+C^0\ . \tag{3–45}$$

Example 3.2.24. *[15, 249] Consider the action of the dihedral group D_8 on the 16-element set of colorings in black and white of the corners of a square. The cycle indicator of D_8 is*

$$P_{D_8}(x_1,x_2,x_3,x_4) = \frac{1}{8}(x_1^4+2x_1^2x_2+3x_2^2+x_4^1)\ . \tag{3–46}$$

Note that, in particular, we get no x_3 factors by Lagrange. The identity element in D_8 fixes all collorings, and is represented as a permutation by four cycles of length one. By Polya's formula, we make the substitution

$$x_1^4 = (B^1 + W^1)^4 = B^4 + 4B^3W + 6B^2W^2 + 4BW^3 + W^4 \ , \tag{3–47}$$

where the exponent outside the parenthesis is the number of cycles, and the exponents inside the parenthesis correspond to the lengths of the cycles, for each color. The element of order four in D_8 has one cycle of length four, so we get

$$x_4^1 = (B^4 + W^4)^1 = B^4 + W^4 \ , \tag{3–48}$$

that is, the permutation $r = (1\ 2\ 3\ 4)$ fixes the squares whose corners are all black or all white. Next, there are three elements in D_8 which comprise two cycles of length two, so each of those yield

$$x_2^2 = (B^2 + W^2)^2 = B^4 + 2B^2W^2 + W^4 \ . \tag{3–49}$$

The last two elements in D_8 both have two cycles of length one, plus one cycle of length two, which gives

$$x_1^2x_2 = (B + W)^2(B^2 + W^2) = B^4 + 2B^3W + 2B^2W^2 + 2BW^2 + W^4 \ . \tag{3–50}$$

Putting it all together, we get the following pattern inventory of orbits:

$$P_{D_8}(B+W, B^2+W^2, B^3+W^3, B^4+W^4) = B^4+4B^3W+6B^2W^2+4BW^3+W^4 \ . \tag{3–51}$$

CHAPTER 4
THEORETICAL FRAMEWORK

This section proposes a model to describe derivation in a general setting. After writing the chapter, Luis will come back here and summarize what he wrote.

4.1 Defining an algorithm to compute a row class

In a traditional setting, a composer will usually define a class of transforms of a basic row by computing by hand what is commonly termed as a 12-tone matrix. The general procedure is usually to fill the top row of the matrix with a transform of the row beginning with zero. The second step is to fill the leftmost column of the matrix with the inverse transform of the top row. The subsequent steps comprise going row-by-row and filling them with a transposition of the top row. The particular transposition for each row is the first pitch-class of the row. So if a row begins with, say, pitch-class 5, then the entire row will be the T_5 transform of the top row. Then 12-tone matrix will hold all 48 transforms of a row: reading every row from left to right gives all transpositions, reading them from right to left gives all retrogrades, reading all columns from top to bottom gives all inversions, and reading them from bottom to top gives all retrograde inversions.

Although somewhat tedious, computing 12-tone matrices by hand is a well-established procedure. The 48 transforms that can be inferred from the matrix are not necessarily unique, given that a 12-tone may be retrograde, or retrograde-inverse invariant. It is straightforward to generalize the procedure to n-tone matrices. However, the graphical idea of reading the matrix from right to left, or bottom to top, is not ideal for computing self-deriving combination matrices, as it is faster to read a transform of a row that is written contiguously in memory. Another caveat is that the procedure does not necessarily define a canonical form for the top row. Every row in a matrix is a representative of an equivalence class of rows under the action of RT_NI. The size of each orbit is $4n$ if the row is not retrograde or retrograde-inverse invariant, and $2n$ otherwise. Defining a canonical form for the first row in a data structure that

holds a row class is desirable in any algorithm that iterates through row classes in lexicographic order.

4.1.1 The canonical form of a row

This section describes an algorithm to obtain a canonical form for a row. The canonical form defined here is simply the least element in a lexicographic ordering of a row class. Given an arbitrary representative of a row class, that representative will be considered retrograded if any interval class between two consecutive elements of the row, counted from right to left, is strictly less than the corresponding interval class counted from left to right. Finding the interval class between two pitch-classes is illustrated in Listing 4.1, which takes as input two pitch-classes a and b, and the base n, returning the interval class between the two. The interval class between $a \pmod n$ and $b \pmod n$ is expressed mathematically as $\min\{(a-b) \pmod n, (b-a) \pmod n\}$.

Listing 4.1. Computing the interval class between two pitch-classes.

```
number intervalClass(number a, number b, number rowSize) {          1
    number interval = abs(a - b);                                   2
    return min(interval, rowSize - interval);                       3
}                                                                   4
```

1. *intervalClass* takes as input two pitch-classes a and b, as well as the base n so that $a, b \in \mathbb{Z}/n\mathbb{Z}$.
2. Line 2 simply computes the distance between a and b.
3. Line 3 returns the least between the above interval and its complement, that is, the interval class between the two given pitch-classes.

The procedure for determining whether a row is retrograded utilizes Listing 4.1 as a subroutine, and is outlined in Listing 4.2. The sole parameter to Listing 4.2 is the row at hand, which can be any of the representatives of its row class, and the returned value is a boolean indicating whether the row is retrograded or not.

Listing 4.2. Determining whether a representative of a row class is retrograded.

```
bool isRetrograded(const number *row, number rowSize, bool isInvariant) {        1
    if (isInvariant || rowSize < 5)                                              2
        return false;                                                            3
                                                                                 4
    number front, back;                                                          5
                                                                                 6
    for (number i = 0; i < rowSize; ++i) {                                       7
        front = intervalClass(row[i + 1], row[i], rowSize);                      8
        back = intervalClass(row[rowSize - i - 1], row[rowSize - i - 2],         9
            rowSize);
                                                                                 10
        if (back < front)                                                        11
            return true;                                                         12
    }                                                                            13
                                                                                 14
    return false;                                                                15
}                                                                                16
```

1. *isRetrograded* has three parameters, namely a pointer to a row, its size, and a boolean representing whether the row is retrograde or retrograde inverse-invariant.
2. Lines 2 and 3 perform a straightforward sanity check. It will be shown below that all rows of size 4 or less are necessarily retrograde or retrograde-invariant. If a row with 5 or more elements is known to be invariant, we skip the test.
5. Line 5 declares the variables that will represent interval classes seen from left to right, and from right to left, respectively.
8. Lines 8 and 9 simply update the variables front and back using Listing 4.1 defined above. As i increases, front traverses all interval classes in the row from left to right, whereas back does the same from right to left, that is, back looks at the row as if it were retrograded.
11. Line 11 compares the front and back interval classes and line 12 returns true if any interval class seen from right to left is strictly less than the corresponding interval class

seen from left to right, which would indicate that retrograding the row would produce a lexicographically lower-positioned row than the one given as input.

15. If all interval classes seen from both directions are equal, then the row is retrograde or retrograde inverse-invariant. Line 15 then returns false to avoid unnecessarily reversing a row that is invariant.

The next step in determining the canonical form of a row consists of finding whether its inverse has a lower lexicographic position within the row class. The procedure is described in Listing 4.5, which in turn depends on two small subroutines, namely Listing 4.3 and Listing 4.4. The former simply computes $modulo(x, n) = x \pmod{n}$. However, it needs to be defined in a subroutine this way because the result of C-language built-in modulo operator is signed, and for practical reasons that will be elaborated below, it is best that all pitch-classes be represented as unsigned integers ranging from 0 to $n - 1$.

Listing 4.3. A subroutine to compute $x \pmod{n}$ such that the result is non-negative.

```
number modulo(number x, number base) {                    1
    x %= base;                                            2
    return x < 0 ? x + base : x;                          3
}                                                         4
```

2. Line 2 simply computes $x \pmod{n}$ using the C-language built operator.
3. Line 3 then returns then same congruence class but in the range $[0, n - 1]$.

Listing 4.4 is a simple convenience method that is used to check whether $x \equiv X \pmod{n}$, which is also needed in Listing 4.5.

Listing 4.4. A subroutine that returns true if a pitch-class is its own inverse.

```
bool isOwnInverse(number x, number base) {                1
    return x == 0 || x == base - x;                       2
}                                                         3
```

1. The input x is assumed to be in the range $[0, n - 1]$.

2. Since x is non-negative, it can only be its own inverse if it is the identity, or if $x \equiv -x \pmod{n}$. It is also possible to check that if n is even and x is equal to $n/2$, then x is its own inverse. But this alternative method is far more computationally expensive that the one in Listing 4.4.

In order to determine whether the inverse transform of a row has a lower lexicographical position than itself, it is necessary in this implementation to consider if its retrograde transform also sits in a lower lexicographical position. The reason for that is because the input row has not been transformed at all up to this point. A prerequisite to calling Listing 4.5 is to have previously called Listing 4.2, the result thereof being used as one of the inputs to Listing 4.5.

Listing 4.5. Determining whether a representative of a row class is inverted.

```
bool isInverted(const number *r, number rowSize, bool R) {
    if (R) {
        for (number i = rowSize - 1; i >= rowSize / 2; --i) {
            number mod = modulo(r[i] - r[rowSize - 1], rowSize);

            if (!isOwnInverse(mod, rowSize))
                return mod > rowSize - mod;
        }
    } else {
        for (number i = 0; i < rowSize / 2; ++i) {
            number mod = modulo(r[i] - r[0], rowSize);

            if (!isOwnInverse(mod, rowSize))
                return mod > rowSize - mod;
        }
    }

    return false;
}
```

1. The inputs are a row, its size, and a boolean representing whether the row is retrograded.

3. If r is retrograded, line 3 begins iteration in reverse order, that is, starting with the last element of the row.
4. Line 4 computes the ordered interval between the pitch-class at hand, and the first pitch-class of the row, which in this case is the last because the row is retrograded.
6. Line 6 checks if the interval is not its own inverse and, if not, line 7 checks if the interval is greater than its inverse. If so, an inverted form of the row would sit in a lower lexicographical position within the entire row class, and the row is inverted.
10. If the input row is not retrograded, then lines 10 to 14 perform essentially the same check as the previous branch, only traversing the row from left to right, instead of the right to left traversal done for a retrograded row. In both branches, it is important to note that the loops iterate at most twice. If the row size is odd, then the very first iteration will decide if the row is inverted. If, on the other hand, the row size is even, then there is the possibility that the very first interval will be its own inverse, in which case the next iteration will invariably be able to tell if the row is inverted.
18. Line 18 should never be reached for well-defined inputs, and is included for correctness.

Determining that an arbitrary representative of a row class is retrograded or inverted are essential subroutines used in Listing 4.8, which in turn does transform the input row into its canonical form. This transformation happens in Listing 4.7, and the latter relies on a straightforward method to in-place swap row entries, described below in Listing 4.6.

Listing 4.6. Swapping two entries in a row.

```
void swapInPlace(number *row, number i, number j) {                1
    number tmp = row[i];                                           2
    row[i] = row[j];                                               3
    row[j] = tmp;                                                  4
}                                                                  5
```

The procedure in Listing 4.6 is very common-place, so the details are omitted. Listing 4.7 is also straightforward and represents the main subroutine in Listing 4.8.

Listing 4.7. Transforming a row in place.

```
void getTransformInPlace(number *row, number rowSize, number T, bool I, bool R  1
    ) {
    if (R) {                                                                    2
        for (number i = 0; i < rowSize / 2; ++i)                                3
            swapInPlace(row, i, rowSize - 1 - i);                               4
    }                                                                           5
                                                                                6
    if (I) {                                                                    7
        for (number i = 0; i < rowSize; ++i)                                    8
            row[i] = (rowSize - row[i] + T) % rowSize;                          9
    } else if (T != 0 % rowSize) {                                              10
        for (number i = 0; i < rowSize; ++i)                                    11
            row[i] = (row[i] + T) % rowSize;                                    12
    }                                                                           13
}                                                                               14
```

1. The inputs are a row, its size, a non-negative integer representing an offset to be added to the entire row, a boolean representing whether the row is inverted, and another boolean representing whether the row is retrograded.
3. If the row is retrograded, lines 3 and 4 simply reverse the row using Listing 4.6.
8. If the row is inverted, lines 8 and 9 invert the row and add the offset to each row entry.
11. Otherwise, if the row is not inverted, lines 11 and 12 only add the offset to the entire row.

The procedure illustrated in Listing 4.7 is used below in Listing 4.8, but it will also be used many times over to construct a row class in memory. In the particular case where the canonical form of the row is desired, that is, the lowest element in a lexicographical ordering of the row class, the offset used will be the additive inverse of the row's first element. It is important to observe that Listing 4.8 requires prior knowledge as to whether the row is retrograde or

retrograde inverse-invariant. In the grand scheme of things, that knowledge will already be available when calling Listing 4.8.

Listing 4.8. Transforming a row into its canonical form.

```
void getCanonicalForm(number *row, number rowSize, bool isInvariant) {          1
    bool R = isRetrograded(row, rowSize, isInvariant);                          2
    bool I = isInverted(row, rowSize, R);                                       3
    number T = row[R ? rowSize - 1 : 0];                                        4
    getTransformInPlace(row, rowSize, I ? T : rowSize - T, I, R);               5
}                                                                               6
```

1. The inputs are a row, its size, and whether the row is retrograde or retrograde inverse-invariant.
2. Lines 2 and 3 simply call Listing 4.2 and Listing 4.5 respectively, storing the results.
4. Line 4 just stores the first element of the row, which may be the last if the row is retrograded.
5. Line 5 is a call to Listing 4.7. The third argument is the additive inverse of the row's start element. It varies, naturally, if the row is inverted.

4.1.2 Storing a row class in memory

The purpose of the procedures described in this section is to store an entire row class contiguously in memory. That consists of creating an array large enough to hold all transforms of a row, and copying sequentially into this array said transforms. Even for rows whose base are considered large, a row class under RTN_I is still very small. Thus holding these transforms in memory makes sense for execution speed purposes. The size of the row class array depends on whether the row is reverse or reverse inverse-invariant. The row class array begins with the canonical for of the row, followed by all its transpositions, in ascending order, followed by all transpositions of the inversion of the canonical for, also in ascending order. If the row is not reverse or reverse inverse-invariant, the row class array is appended by all transpositions of the reverse of the canonical order, followed by all transpositions of the retrograde inverse of the

canonical order. Therefore, the overall size of the row class array is $2n^2$ if the row is retrograde or retrograde inverse-invariant, or $4n^2$ otherwise, where n is the row size. The main routine for creating the row class array is described below in Listing 4.11. It depends on two subroutines, namely Listing 4.9 and Listing 4.10 that are described below.

Lemma 4.1.1. *In a retrograde-invariant row, the interval between every entry seen from left to right, and the corresponding entry seen from right to left, must equal half the size of the row.*

Proof. □

Corollary 4.1.2. *A row with odd size greater than* 3 *cannot be retrograde-invariant.*

Proof. □

Listing 4.9. Determining whether a row is retrograde-invariant.

```
bool isRetrogradeInvariant(const number *row, number rowSize) {          1
    if (rowSize > 3 rowSize % 2 == 1)                                    2
        return false;                                                    3
                                                                         4
    number half = rowSize / 2;                                           5
                                                                         6
    for (number i = 0; i < half; ++i) {                                  7
        if (modulo(row[rowSize - i - 1] - row[i], rowSize) != half)      8
            return false;                                                9
    }                                                                    10
                                                                         11
    return true;                                                         12
}                                                                        13
```

1. The only inputs are a row and its size.
2. Line 2 utilizes Th. 4.1.2 as a sanity check. If the test in line 2 succeeds, line 3 returns false.
5. Line 5 just computes half the row size for convenience. Retrograde-invariance can be determined by iterating through at most half the size of the row.

7. Line 7 iterates through the row from left to right, and line 8 applies Th. 4.1.1. Line 9 then returns false if any of the intervals is not equal to half the row size.
12. If all pairs coming from both directions are equal to half the row size, then the row is retrograde-invariant and line 12 returns true.

Unlike retrograde-invariance, every row size can produce retrograde inverse-invariance. A simple example is the lowest lexicographically-order row of any size, that is, a chromatic scale. Retrograding, inverting, and transposing by -1 always yields back the initial row. Listing 4.10 outlines the procedure in all generality.

Lemma 4.1.3. *In a retrograde inverse-invariant row r, the interval between every entry seen from left to right, and the corresponding entry seen from right to left, must be equal to $r_0 + r_{n-1} \pmod{n}$, where n is the base of the row.*

Proof. □

Listing 4.10. Determining whether a row is retrograde inverse-invariant.

```
bool isRetrogradeInverseInvariant(const number *row, number rowSize) {          1
    number half = rowSize / 2;                                                  2
    number first = modulo(row[rowSize - 1] + row[0], rowSize);                  3
                                                                                4
    for (number i = 1; i < half; ++i) {                                         5
        if (modulo(row[rowSize - i - 1] + row[i], rowSize) != first)            6
            return false;                                                       7
    }                                                                           8
                                                                                9
    return true;                                                                10
}                                                                               11
```

1. The inputs are a row and its size.
2. Line 2 computes half the row size for convenience. Like Listing 4.9, retrograde inverse-invariance can be determined by traversing only half the row size.
3. Line 3 computes $r_0 + r_{n-1} \pmod{n}$ for later use.

5. Line 5 begins iteration from the second entry onward, since the first pair was already computed in line 3. Line 6 then compares the subsequent corresponding pairs, and line 7 returns false if any of them is not equal to the first pair.

10. If line 10 is reached, it means the row is retrograde inverse-invariant, so the algorithm returns true.

The procedure to compute and store an entire row class in memory is outlined below in Listing 4.11. It returns a memory address that must be freed by the caller. While computing combination matrices of self-derivation, this memory location will be held for the duration of the program, and possibly also be shared by many different threads of execution. The procedure makes heavy use of Listing 4.7 to sequentially store in memory all transforms of the canonical form of the row. It also takes the input row and transforms it in-place into its canonical form. Any representative of a row class can therefore be used to construct a row class with Listing 4.11.

Listing 4.11. Computing and storing a row class in memory.

```
number *getRowClass(range *classSize, number *row, number rowSize, bool isInvariant) {
    getCanonicalForm(row, rowSize, isInvariant);
    *classSize = isInvariant ? rowSize * 2 : rowSize * 4;

    number *rowClass = malloc(rowSize * *classSize * sizeof(number));
    memcpy(rowClass, row, rowSize * sizeof(number));

    for (number i = 1; i < rowSize; ++i) {
        number *rep = &rowClass[i * rowSize];
        memcpy(rep, rowClass, rowSize * sizeof(number));
        getTransformInPlace(rep, rowSize, i, false, false);
    }

    for (number i = rowSize; i < rowSize * 2; ++i) {
```

```
        number *rep = &rowClass[i * rowSize];                          15
        memcpy(rep, rowClass, rowSize * sizeof(number));               16
        getTransformInPlace(rep, rowSize, i, true, false);             17
    }                                                                  18
                                                                       19
    if (*classSize == rowSize * 2)                                     20
        return rowClass;                                               21
                                                                       22
    for (number i = rowSize * 2; i < rowSize * 3; ++i) {               23
        number *rep = &rowClass[i * rowSize];                          24
        memcpy(rep, rowClass, rowSize * sizeof(number));               25
        getTransformInPlace(rep, rowSize, i, false, true);             26
    }                                                                  27
                                                                       28
    for (number i = rowSize * 3; i < rowSize * 4; ++i) {               29
        number *rep = &rowClass[i * rowSize];                          30
        memcpy(rep, rowClass, rowSize * sizeof(number));               31
        getTransformInPlace(rep, rowSize, i, true, true);              32
    }                                                                  33
                                                                       34
    return rowClass;                                                   35
}                                                                      36
```

1. The inputs are a pointer to the size of the row class, which will be computed by Listing 4.11 and stored in the address provided, a row, its size, and a boolean representing whether the row is retrograde or retrograde inverse-invariant. This boolean will be computed by the caller before calling Listing 4.11, utilizing the aforementioned procedures Listing 4.9 and Listing 4.10. The output is a memory location containing the entire row class.
2. As mentioned above, line 2 simply computes the canonical form of the given row in place.

3. Line 3 computes the class size based on whether the row is retrograde or retrograde inverse-invariant. It is important to notice here that the row class size is not the length of the row class array, but the number of transforms of the canonical form that the row class array contains. This number is stored in the address provided by the caller.
5. Line 5 allocates in the heap the memory space that will be used to store the entire row class, and line 6 copies the canonical form of the row to the beginning of this allocated space.
8. Line 8 starts from the second row in the row class, which is the first transposition of the canonical form, and the block adds all subsequent transpositions to the row class. Line 9 computes the address where the current transposition should start, line 10 copies the canonical for of the row into this address, and line 11 transforms the canonical for into the desired transposition in place.
14. Line 14 starts from the thirteenth row in the row class, that is, the $T_0 I$ transform of the row. Lines 15 to 17 add the subsequent $T_n I$ transforms of the canonical for of the row in the same way the T_n transforms were added above, only specifying in the call to Listing 4.7 that the transforms now need to be inverted.
20. Line 20 checks if the row is retrograde or retrograde inverse-invariant and, if so, halts the process and returns the memory location allocated in line 5.
23. If the row is is retrograde or retrograde inverse-invariant, lines 23 to 26 add all RT_n transforms of the row to the row class as above.
29. Similarly, lines 29 to 32 add all the $RT_n I$ transforms of the row.
35. Line 36 then simply returns the memory location allocated in line 5.

4.2 Defining an algorithm to compute semi-magical squares

A key component in the definition of a combination matrix of self-derivation is a semi-magic square. By definition, a semi-magic square is a square matrix where the sum of all elements, for each row and column, is a constant. For our purposes, that constant will be the base n of the row. Seeing each entry in a semi-magic square as a real number and dividing

each entry in the matrix by n produces a doubly stochastic matrix, in which every row and every column sum to one. A magic square is a semi-magic square where, in addition, both main diagonals add up to the same constant as the rows and columns.

4.2.1 Computing the partitions of an integer

By the definition above of a semi-magic square, every row therein is a permutation of an integer partition of n. In fact, every column is also a permutation of an integer partition of n, but the algorithm described below in Listing 4.14 shall concentrate on computing all integer partitions of n and their permutations as rows that will ultimately become the rows of a semi-magic square. Since these integer partitions of n are completely independent of the row at hand, they can be computed only once for each n and stored as a file, speeding up the process of computing combination matrices for other rows. Before describing Listing 4.14, a couple subroutines for reading and writing files are outlined.

Listing 4.12. Retrieving the size of a file.

```
long getFileSize(FILE *file) {                  1
    fseek(file, 0, SEEK_END);                   2
    long fileSize = ftell(file);                3
    fseek(file, 0, SEEK_SET);                   4
    return fileSize;                            5
}                                               6
```

Listing 4.12 is a completely common-place procedure in the C-language, and its details are omitted. It is used as a subroutine in Listing 4.13, which is described next.

Listing 4.13. Reading the contents of a file.

```
number *readFile(const char *fileName, length *numLines, number problemSize) {  1
    FILE *file = fopen(fileName, "rb");                                          2
                                                                                 3
    if (file == NULL)                                                            4
        return NULL;                                                             5
                                                                                 6
```

```
    long fileSize = getFileSize(file);                                    7
    number *buffer = malloc(fileSize);                                    8
    *numLines = (length) (fileSize / sizeof(number)) / problemSize;       9
    fread(buffer, fileSize, 1, file);                                     10
    fclose(file);                                                         11
                                                                          12
    return buffer;                                                        13
}                                                                         14
```

1. The inputs are the file name, assuming both the main executable and the file are in the same folder, a memory address where the number of lines read will be stored, and the problem size, which is the size of the magic square. The output is a memory location with the contents of the file, or a NULL pointer if the operation fails.
2. Line 2 simply opens the file at the given path for reading. Line 4 performs a sanity check and line 5 returns a NULL pointer if the file cannot be read.
7. Line 7 is a call to Listing 4.12 so that the correct amount of memory can be allocated. The value computed in line 7 is already in bytes. Line 8 then allocates the memory whose address will be returned.
9. Line 9 computes the number of lines in the file, that is, the number of integer partitions of n with length *problemSize* that have been previously computed and stored in the file. The value is then stored in the memory address provided in the input list to store the number of lines in the file.
10. Line 10 simply reads the entire contents of the file into the buffer allocated in line 8, line 11 frees the FILE data structure created in line 2, and line 13 returns the allocated memory. It is the responsibility of the caller to free this memory when no longer needed.

The procedure described in Listing 4.15 effectively writes all integer partitions of n with a certain size to a file. It depends on Listing 4.14 to compute all said partitions recursively, which is described next. One important feature of Listing 4.14 is that it skips partitions whose all but one entries are zero. The reason for this is because the remaining entry would be forced

to be equal to n. In any semi-magic square where there is an entry equal to n, both the row and column to which this entry belongs will necessarily contain only zeros except, of course, for the entry with value n. This means that the linearized top of the combination matrix at that semi-magic square's column will inevitably be trivially polyphonized, that is, the corresponding square cell would simply have the entire row at the top. Therefore any combination matrix can be extended by adding a column and a row at an arbitrary indices, and making that square entry contain n elements, that is, a linearized statement of a row form. Since this procedure is completely general, the semi-magic squares computed below include entries only up to $n - 1$.

Listing 4.14. Recursively computing all partitions of a number n and permutations thereof with a certain size.

```
void allPartitionsRecursive(number *tmp, number currentSize, number
    problemSize, number currentSum, number rowSize, FILE *file) {
    if (currentSize == problemSize - 1) {
        if (currentSum > 0) { // skip the [0, ..., 0, rowSize] partition
            tmp[currentSize] = rowSize - currentSum;
            fwrite(tmp, sizeof(number), problemSize, file);
        }

        return;
    }

    for (number i = 0; i < rowSize; ++i) {
        if (i + currentSum <= rowSize) {
            tmp[currentSize++] = i;
            allPartitionsRecursive(tmp, currentSize, problemSize, i +
                currentSum, rowSize, file);
            currentSize--;
        }
    }
}
```

1. The inputs are the address of some scratch memory previously allocated with the same length of the problem size, the current number of entries already placed in the scratch memory, the problem size, the current sum of the entries in the scratch memory, the base n, and a pointer to a FILE data structure where the partitions will be stored, line by line.
2. Line 2 deals with the base case of the recursion, that is, when the current number of elements in the scratch memory is one less than the problem size. If that is the case, then the last element in the scratch memory array must be equal to n minus the sum of all previous elements.
3. Line 3 deals with the edge case where all entries up to problem size minus one are zero. That would force the very last element of the scratch memory array to be equal to n which, by the discussion above is avoided in this implementation.
4. Line 4 sets the last element of the scratch memory array to n minus the sum of its previous elements and line 5 writes the current partition to the given file. Having completed an entire partition, line 8 returns, so the function recurses no further.

11. For the recursion cases where the scratch memory array has been filled with less than problem size minus one elements, the loop in line 11 iterates from zero to $n - 1$. Line 12 is a sanity check that the sum of the current element being appended to the scratch memory array with the current sum of all elements so far introduced is no greater than n. Line 13 then appends the current element to the end of the scratch memory array, increases the current size counter, and line 14 pushes to the call stack another recursive call to Listing 4.14.

13. Line 15 backtracks the current size so that all combinations of integers in the range $[0, n - 1]$ are tried.

Listing 4.15 described below is essentially a wrapper around Listing 4.14. Partitions files are saved in the same location where the main executable is built, in order to facilitate dealing with file paths. In general, these files are very small even for large n, but the speed gains in precomputing them are worthwhile, given the recursive nature of Listing 4.14.

Listing 4.15. Writing all partitions to a file.

```
void writeAllPartitions(number problemSize, number rowSize) {                1
    char fileName[32];                                                       2
    sprintf(fileName, "all_partitions_%i_%i.dat", problemSize, rowSize);     3
    FILE *file = fopen(fileName, "wb");                                      4
                                                                             5
    number *tmp = malloc(problemSize * sizeof(number));                      6
    allPartitionsRecursive(tmp, 0, problemSize, 0, rowSize, file);           7
                                                                             8
    free(tmp);                                                               9
    fclose(file);                                                            10
}                                                                            11
```

1. The inputs are a problem size, which is the size of the semi-magic square, and a row size, which is the row base n.
2. Line 2 creates an array of 32 characters to hold the file name. This file name array length is large enough to accommodate all file names in this implementation. Line 3 writes the name of the file into the buffer defined in the previous line, and line 4 opens a C-type FILE for writing at the specified path.
6. Line 6 allocates the scratch memory needed for the call to Listing 4.14 in the following line. Being that Line 7 is the bottommost call on the call stack for Listing 4.14, the arguments given for the current size and current sum parameters are both zero.
9. Line 9 frees the scratch memory used in the calls to Listing 4.14, and line 10 closes and releases the file created in line 4.

The counterpart to Listing 4.15 is Listing 4.16, which is designed to not only read a partitions file, but to also generate one in case the particularly sought partitions file had not been previously created.

Listing 4.16. Reading all partitions from a file.

```
number *readAllPartitions(length *numPartitions, number problemSize, number      1
    rowSize) {
    char fileName[32];                                                            2
    sprintf(fileName, "all_partitions_%i_%i.dat", problemSize, rowSize);          3
                                                                                  4
    number *allPartitions = readFile(fileName, numPartitions, problemSize);       5
                                                                                  6
    if (allPartitions == NULL) {                                                  7
        printf("Writing partitions file...\n");                                   8
        writeAllPartitions(problemSize, rowSize);                                 9
        allPartitions = readFile(fileName, numPartitions, problemSize);           10
    }                                                                             11
                                                                                  12
    return allPartitions;                                                         13
}                                                                                 14
```

1. The inputs are a memory address to store the number of lines in the partitions file to be read, the problem size and the row size. The output is a memory address with the contents of the partitions file. It is the responsibility of the caller to free the returned memory when no longer needed.
2. Lines 2 and 3 store the file name in a buffer, similarly to Listing 4.15.
5. Line 5 attempts to read an existing file.
7. Line 7 checks that the operation in line 5 succeeded. If it failed, lines 8 to 10 print to the console that a partitions file will be created, create the file by calling Listing 4.15, and attempt to read the file again.
13. Line 13 simply returns the memory address with the contents of the file, or NULL if creating the file failed.

4.2.2 Combining partition rows into squares

This section describes an algorithm to combine integer partition rows into semi-magic squares. The overall procedure consists of appending integer partition rows to the square and

adding the the rows as vectors. It is a backtracking recursive algorithm similar to Listing 4.14, the main difference being that the backtracking step consists of subtracting from the vector sum the previously added row after pushing onto the stack another recursive call. Adding two integer partition rows, seen as vectors, is described next in Listing 4.17.

Listing 4.17. Adding two vectors.

```
void plus(number *a, const number *b, number problemSize) {                1
    for (number i = 0; i < problemSize; ++i)                               2
        a[i] += b[i];                                                      3
}                                                                          4
```

1. The inputs are the memory locations of two vectors and their size, which assumed to be the same.
2. Line 2 simply iterates through the length of the vectors and line 3 adds point-wise the second vector to the first.

The counterpart to Listing 4.17 is Listing 4.18. It is important to notice that vector addition and subtraction can be vectorized for speed, that is, SIMD instructions from the processor architecture at hand can replace much of the work the for-loops in Listing 4.17 is Listing 4.18 do to great effect. This option is not explored in this implementation, however. Details are omitted for Listing 4.18, as is essentially the same as Listing 4.17, only with the inverse operation.

Listing 4.18. Subtracting one vector from another.

```
void minus(number *a, const number *b, number problemSize) {              1
    for (number i = 0; i < problemSize; ++i)                               2
        a[i] -= b[i];                                                      3
}                                                                          4
```

Listing 4.19 is also used as subroutine in Listing 4.20 to validate whether the integer partition row that had just been added to the sum of rows vector does not violate the semi-magic square constraints.

Listing 4.19. Validating a sum of integer partition rows.

```
bool validate(const number *currentSum, number problemSize, number rowSize) {   1
    for (number i = 0; i < problemSize; ++i) {                                   2
        if (currentSum[i] > rowSize)                                             3
            return false;                                                        4
    }                                                                            5
                                                                                 6
    return true;                                                                 7
}                                                                                8
```

1. The inputs are the memory location of the current sum of rows, the problem size, and the row size n.
2. Line 2 iterates through the size of the magic square, checking in line 3 whether any entry in the current sum vector is already greater than n, fact that would violate the semi-magic square definition. If so, line 4 returns false.
7. If all entries in the current sum vector are less than or equal to n, this is still a viable semi-magic square, so line 7 returns true.

Listing 4.20 utilizes the previously computed integer partition rows to form all possible semi-magic squares of size n, with entries in the range $[0, n-1]$. It is described here because it facilitates understanding of Listing 4.30, but it is not used in computing combination matrices of self-derivation. For the latter purpose, computing *all* possible semi-magical squares would be impractical, but this discussion is deferred to the next sections. Although Listing 4.20 takes as input a pointer to a FILE data structure, a procedure to save all semi-magic squares is omitted, as the number of such squares is overwhelmingly large and grows with both the problem size and n.

Listing 4.20. Recursively computing all semi-magic squares of a certain size.

```
void allSquaresRecursive(number *tmp, number *currentSum, number currentSize,   1
    number problemSize, number rowSize, const number *partitions, length
    numPartitions, FILE *file) {
```

```
    if (currentSize == problemSize) {                                    2
        fwrite(tmp, sizeof(number), problemSize, file);                  3
        return;                                                          4
    }                                                                    5
                                                                         6
    number start = 0;                                                    7
                                                                         8
    if (currentSize == 0)                                                9
        memset(currentSum, 0, problemSize * sizeof(number));             10
    else                                                                 11
        start = tmp[currentSize - 1];                                    12
                                                                         13
    for (length i = start; i < numPartitions; ++i) {                     14
        const number *p = &partitions[i * problemSize];                  15
        plus(currentSum, p, problemSize);                                16
                                                                         17
        if (validate(currentSum, problemSize, rowSize)) {                18
            tmp[currentSize++] = i;                                      19
            allSquaresRecursive(tmp, currentSum, currentSize, problemSize,  20
                rowSize, partitions, numPartitions, file);
            currentSize--;                                               21
        }                                                                22
                                                                         23
        minus(currentSum, p, problemSize);                               24
    }                                                                    25
}                                                                        26
```

1. The inputs are the memory address for some scratch memory where the semi-magical square will be stored as a sequence of indices into the integer partitions array. The size of this allocated memory must this be equal to the problem size. The memory address of another scratch memory space to hold the current sum of integer partition rows, with size equal to the problem size, the current number of rows already appended to the

square, the problem size, the row size n, a pointer to the contents of a partitions file, previously computed using Listing 4.14, the total number of integer partition rows, and a pointer to a FILE data structure.

2. Line 2 deals with the base case of the recursion, that is, when all rows have been appended to the square. When that is the case, the algorithm writes the square to the file and returns.

7. Line 7 defines a variable for where iteration of integer partition rows should start. This is a mechanism to avoid permutations of integer partition rows within a semi-magic square by enforcing that the rows of the constructed semi-magic square, seen as indices in the range between zero and the total number of integer partitions, be a nondecreasing sequence. The next section shall expand on the discussion of avoiding permutations of rows in semi-magical squares.

9. Even though the current size is normally set to zero on a very first call to Listing 4.20, the backtracking nature of the algorithm will push onto the call stack many other calls to Listing 4.20 where the current sum scratch memory might have been previously used, thus containing left over data. Line 9 then checks if the current size is zero, and line 10 sets all entries in the current sum scratch space to zero, to clear any left over data. If, on the other hand, the current size is greater than zero, then line 12 sets the start of the iteration to the very last index into the integer partition rows seen so far, guaranteeing that the sequence of indices in each square is nondecreasing.

14. Line 14 begins iteration from the last index already in the square. In line 15, the variable p stores the memory location of the integer partition row at the current index of iteration, and line 16 utilizes Listing 4.17 to add this integer partition row to the current sum scratch memory.

18. Line 18 calls Listing 4.19 to validate the current sum and, if the sum is valid, line 19 sets the next index in the current square scratch memory, here represented by the variable

tmp to the current iteration index and increases the current size. Line 20 then pushes another call to Listing 4.20 onto the stack, and line 21 backtracks the current size.

24. Regardless whether the validation test in line 18 succeeded, the current sum must be backtracked inside the same block where Listing 4.17 was called. Line 24 accomplishes that with a balancing call to Listing 4.18.

4.3 Computing all possible combination matrices for a row

This section outlines a procedure that produces all combination matrices of self-derivation of a certain size for a particular row. The main idea relies on departing from a top row, which is a concatenation of transforms of the canonical form of a row r. If the row size is n, and the problem size is m, then the size of the top row is $m \times n$. In terms of derivation theory, the top row is completely linearized, and each of its m transforms of the row r will constitute a column of the combination matrix, if a solution for the top exists. The top itself is not part of the combination matrix, but rather the linearization thereof. It is, nonetheless, the point of departure of the main algorithm in this implementation. The combination matrix per se consists of a semi-magic square of size m by m, where each column in the square defines a partition of the transform of r for the corresponding section of the top. The last component in this view of a combination matrix of self-derivation are the transforms of the row r that can be derived from the top row as a whole from the integer partitions given by the rows of the semi-magic square. These derived rows are a column vector of size m which are called the sides of the combination matrix.

Example 4.3.1. *Let* $S = \{3, 8, 1, 0, 9, 6, 4, 7, 10, 5, 2, 11\}$ *as in Ex. 1.2.6 and consider the combination matrix of self-derivation in Eq. 1–21. In it, the top vector is* $[\mathsf{T}_7(S)|\,\mathsf{R}\,\mathsf{T}_2\,\mathsf{I}(S)]$, *the side vector is* $[S|\,\mathsf{T}_9\,\mathsf{I}(S)]^T$, *and the semi-magic square is*

$$\left[\begin{array}{c|c} 6 & 6 \\ \hline 6 & 6 \end{array}\right] . \tag{4–1}$$

The row S is not in its canonical form. An application of Listing 4.8 reveals that the canonical form of S is in fact the row $r = \{0, 3, 6, 11, 8, 5, 7, 10, 1, 2, 9, 4\}$, so that $S = RT_{11}(r)$.

Theorem 4.3.2. *Let r be a row. If a solution exists for a combination of top, side and square, then it is unique.*

Proof. Suppose there exist two different solutions for the same combination of top, side and square. Then each solution comprises a combination matrix of self-derivation, say M_1 and M_2. By assumption $M_1 \neq M_2$. Now consider the linearized top and the first row of M_1, which is a transform of r, say $S_1(r)$. Since M_1 is a solution, when $S_1(r)$ is partitioned using the partition scheme given by the first row of the square, each partition of $S_1(r)$ fits exactly under each column of the top. Each column of the top is totally ordered, and so is each partition of $S_1(r)$. So there is only one way in which one can fit under another. The same reasoning applies to the first row of M_2, and to all subsequent rows of M_1 and M_2. Hence it must be that $M_1 = M_2$ and the solution is unique, as desired. □

Corollary 4.3.3. *Let r be a row. If a solution exists for a combination of top, side and square, then the set of permutations of the side vector form an equivalence class of solutions.*

Proof. The identity permutation of the side is essentially the same combination matrix, hence a solution. Reordering the entries of the side and applying Th. 4.3.2 shows that a permutation of the side is also a solution. Let the problem size be m and regard the side as a vector of indices with entries in $\mathbb{Z}/m\mathbb{Z}$. Then transitivity comes from the fact that $\mathbb{Z}/m\mathbb{Z}$ is a subgroup of S_m, that is, composition of permutations of the side vector are also a solution. □

Corollary 4.3.4. *Let r be a row, let the problem size be equal to m, and regard the top vector as the direct product of m copies of G_r, the group of RT_nI-transforms of r. If a solution exists for a combination of top, side and square, then right-multiplying the solution by an element of G_r represents a group action that induces an equivalence class of solutions. In particular, this equivalence class preserves the structure of the semi-magic square.*

Proof. □

4.3.1 Computing the side and top rows of a combination matrix

As described above, the side and top vectors are key components in this implementation. Even though the side and the top are both vectors of length equal to the problem size, and they are both vectors of RT_nI transforms of a row, there are some important distinctions between them. The most relevant distinction is that the top vector is used to define the input of a problem, whereas the side vector is used to compute a solution. In derivation theory words, the top vector is the linearization of the combination matrix, and each element in the side vector is a transform of the row r that can be derived from from the top. Another fundamental distinction is that the contents of a side are permutable by Corollary 4.3.3, whereas the contents of the top are usually not. Even though there are permutations of the top vector that yield equivalent solutions, those are special cases and will be discussed in more detail in the next section. Lastly, like Listing 4.20, the algorithm presented in Listing 4.21 below also facilitates understanding of Listing 4.30, but it is not used in computing combination matrices of self-derivation. The algorithm described in Listing 4.22, on the other hand, is not only used in this implementation, but will also be the subject of future research.

Listing 4.21. Recursively computing all side vectors of a certain size.

```
void allSidesRecursive(number *tmp, number currentSize, number problemSize,
    range classSize, FILE *file) {
    if (currentSize == problemSize) {
        fwrite(tmp, sizeof(number), problemSize, file);
        return;
    }

    number start = currentSize == 0 ? 0 : tmp[currentSize - 1];

    for (range i = start; i < classSize; ++i) {
        tmp[currentSize++] = i;
```

```
        allSidesRecursive(tmp, currentSize, problemSize, classSize, file);      11
        currentSize--;                                                          12
    }                                                                           13
}                                                                               14
```

1. The inputs are some pre-allocated scratch memory with size equal to the problem size, the current size of the vector, the problem size itself, the size of the row class, and a pointer to a C-style FILE data structure.
2. Line 2 is the base case of the recursion, and simply checks that the current size of the vector has already reached the problem size. If so, line 3 writes the contents of the scratch memory contiguously to the provided FILE handle and line 4 returns control to the caller.
7. Line 7 is a similar mechanism to that described in line 12 of Listing 4.20. Like the latter, its purpose is to avoid permutations of the size vector by forcing upon them being nondecreasing sequences of indices into the row class array. It will be shown in Listing 4.30 that size vectors can actually be strictly increasing sequences.
9. Line 9 begins iteration from the start index computed in line 7, until the last index possible, which is the size of the row class minus one. Line 10 sets the scratch memory's entry at the current size to the current index of iteration and increases the current size. Line 11 pushes onto the call stack another call to Listing 4.21, and line 12 backtracks by decreasing the current size, thus ensuring that all nondecreasing sequences of indices in the range from zero to row class size minus one are seen and written to the provided FILE.

In this implementation, tops are pre-computed and saved to a file. This is a two-edged sword, as it increases speed at the expense of risking some potentially prohibitive file sizes. Let r be a row of size n, let c be the size of its row class, and let m be a problem size. Then the number of all possible tops is c^m, which can be an incredibly large number. If r is a 12-tone row that is not retrograde or retrograde inverse-invariant, then $c = 48$. The number of all

possible 4×4 tops is then $48^4 = 5308416$. For music composition applications, $m = 4$ is a rather small number. A composer may well be interested in combination matrices of sizes up to n, hence reducing the number of tops into smaller equivalence classes is of utmost importance. Under the action of RT_nI, the number of tops can be reduced substantially, but this reduction is asymptotically irrelevant as row size and problem size grow. In other words, dealing with large n and m can be a very difficult problem. On the bright side, this analysis shows that solutions do exist in abundance, however difficult they may be to compute.

Listing 4.22. Recursively computing all top vectors of a certain size.

```
void allTopsRecursive(number *tmp, number currentSize, number problemSize,      1
    range classSize, FILE *file) {
    if (currentSize == problemSize) {                                           2
        fwrite(tmp, sizeof(number), problemSize, file);                         3
        return;                                                                 4
    }                                                                           5
                                                                                6
    if (currentSize == 0)                                                       7
        tmp[currentSize++] = 0;                                                 8
                                                                                9
    for (range i = 0; i < classSize; ++i) {                                     10
        tmp[currentSize++] = i;                                                 11
        allTopsRecursive(tmp, currentSize, problemSize, classSize, file);       12
        currentSize--;                                                          13
    }                                                                           14
}                                                                               15
```

1. The inputs are the memory address of some scratch memory space, pre-allocated with size equal to the problem size, the current size of the top, the problem size, the row class size, and a pointer to a FILE where the tops should be saved.
2. Line 2 handles the base case of the recursion, line 3 writes the contents of the current top to the FILE, and line 4 returns.

8. Line 7 restricts the number of tops that will be computed by enforcing in line 8 that the first element of the top vector be always zero. If the problem size is m and the class size is c, then Listing 4.22 effectively computes c^{m-1} tops. It will be shown in the next section how all equivalence classes of tops under the action of RT_nI can be computed departing from c^{m-1} elements.

11. In line 10, the for-loop always starts from zero, since from the second element to the last, all possibilities are needed. Lines 11 to 13 describe the same backtracking recursive procedure in other algorithms already discussed above.

Similarly to Listing 4.15, Listing 4.23 wraps around Listing 4.22 to save a file in the same location where the the main executable is built, so the details are omitted. Unlike Listing 4.15, however, the output of Listing 4.23 can be a very large file. Suppose r is a row of size $n = 12$, its row class has size $c = 48$, and the problem size is $m = 8$. Since a number of type char requires one byte of memory, the output of Listing 4.23 would be a file of size $c^{m-1} = 546.75$ Gigabytes.

Listing 4.23. Writing all tops to a file.

```
void writeAllTops(number problemSize, range classSize) {                1
    char fileName[NAME_SIZE];                                           2
    sprintf(fileName, TOPS_FILE, problemSize, classSize);               3
    FILE *file = fopen(fileName, "wb");                                 4
                                                                        5
    number *tmp = malloc(problemSize * sizeof(number));                 6
    allTopsRecursive(tmp, 0, problemSize, classSize, file);             7
                                                                        8
    free(tmp);                                                          9
    fclose(file);                                                       10
}                                                                       11
```

The same applies to Listing 4.24, whose details are omitted for being similar to Listing 4.16 above, including the way Listing 4.24 generates a new file if none has been already created.

Listing 4.24. Reading all tops from a file.

```
number *readAllTops(length *numTops, number problemSize, range classSize) {
    char fileName[NAME_SIZE];
    sprintf(fileName, TOPS_FILE, problemSize, classSize);

    number *allTops = readFile(fileName, numTops, problemSize);

    if (allTops == NULL) {
        printf("Writing top combos file...\n");
        writeAllTops(problemSize, classSize);
        allTops = readFile(fileName, numTops, problemSize);
    }

    return allTops;
}
```

4.3.2 Writing a solution as text

The main algorithm in this implementation writes to a file the combination matrices of self-derivation computed from a provided set of tops. It writes them to a provided file in a format that is readily useable for music composition, unless otherwise specified. It takes as input a significant amount of pre-computed data which can be shared between several threads of execution, as well many pointers to scratch memory addresses that must be allocated for each thread. Both common and thread-specific data are organized into data structures to reduce the number of parameters necessary for each call. These data structures, as well as the text formatting of the output will be discussed in this section.

Listing 4.25 below is a data structure where all members are immutable, so is safe to shares between different threads of execution. Some members are common to a particular row,

namely the memory address and size of the row class, the retrograde and retrograde inverse maps addresses, which will be discussed in the next section, the row size, and whether it is retrograde or retrograde inverse-invariant. All other members are common to all rows of a particular size n, given a problem size m. These members are namely the memory address and size of all partitions read from the partitions file, the memory address and size of all tops read from the tops file, and the problem size itself.

Listing 4.25. Defining a data structure to hold thread-common data.

```
typedef struct {
    number *rowClass, *rMap, *riMap, *allPartitions, *allTops;
    length numPartitions, numTops;
    range topSize, classSize;
    number problemSize, rowSize, isInvariant;
} common_data;
```

Listing 4.26 below, on the other hand, cannot be shared between different threads of execution, as it mostly contains allocated scratch memory space to be used in a call to Listing 4.30, which computes all solutions for a given top. The members are, in order, the scratch memory for the size vector, the current top being used as input, the memory address for a stack of tops that will be needed by Listing 4.30, the current sum of the semi-magic square columns, the scratch memory for the semi-magic square itself, and lastly a variable to store the number of solutions obtained by the top at hand.

Listing 4.26. Defining a data structure to hold thread-specific data.

```
typedef struct {
    number *tmpSide, *currentTop, *tmpTops, *currentSum;
    length *tmpSquare;
    length counter;
} thread_data;
```

Listing 4.27 is used in Listing 4.28 as a subroutine to write as text an entire solution line-by-line. The lines in the solution include the top line, as well as one line for each entry in the side vector.

Listing 4.27. Writing a single row from a combination matrix.

```
void writeRow(const number *row, number rowSize, range topSize, bool newLine,    1
    FILE *file) {
    for (range i = 0; i < topSize; ++i) {                                         2
        if (i > 0 && i % rowSize == 0)                                            3
            write(file, "| ");                                                    4
                                                                                  5
        if (row[i] < 0)                                                           6
            write(file, "   ");                                                  7
        else if (row[i] < 10)                                                     8
            write(file, "%i  ", row[i]);                                          9
        else                                                                      10
            write(file, "%i ", row[i]);                                           11
    }                                                                             12
                                                                                  13
    if (newLine)                                                                  14
        write(file, "\n");                                                        15
}                                                                                 16
```

1. The inputs are the combination matrix row at hand, which includes the top line. These rows have the same length as the top row. Rows that are derived from the top, however, will have nonnegative entries whenever those entries align with the top's current index, and minus one otherwise. The row size, which is used to add separators at each of the top's columns, the top size, a boolean to indicate whether a new line character should be included after the combination matrix, and the address of a C-style FILE data structure.

2. Line 2 iterates over the top's length and line 3 checks if the current index of iteration has reached the boundary of a column. If so, line 4 will write a separator character and a space.
6. Line 6 checks if the current row's entry is less than zero, in which case there is nothing to align with the top, hence line 7 writes three spaces. Line 8 checks if the current entry is less than ten, in which case line 9 writes a single-character number and two spaces. If line 10 is reached, then the current entry has two characters, so line 11 lines the two-digit number and a single space.
14. Line 14 checks if a new line character was requested and, if so, line 15 writes one to the file.

The algorithm for converting a solution into text is presented below in Listing 4.28. A text solution consists of the solution number within the thread-specific context, and all the rows appended using calls to Listing 4.27. If a single thread is used, then the solution number is unique within the set of all solutions, otherwise it is only unique within the text file created by the thread the processed the solution. The rows of the combination matrix are prepended by their side vector index, and appended by their square row index, which is an index into the array of all integer partitions of size n, with entries in the range $[0, n-1]$, as previously discussed.

Listing 4.28. Writing an entire solution.

```
void writeSolution(thread_data *d, common_data *cd, FILE *solutionsFile) {      1
    write(solutionsFile, "Solution %lu:\n\n\t\t\t\t\t", d->counter);             2
    writeRow(d->tmpTops, cd->rowSize, cd->topSize, true, solutionsFile);        3
                                                                                4
    for (number i = 0; i < cd->problemSize; ++i) {                              5
        range topIndex = 0;                                                     6
        number rowIndex = 0;                                                    7
        const number *p = &cd->allPartitions[d->tmpSquare[i] * cd->problemSize  8
            ];
```

```
        const number *r = &cd->rowClass[d->tmpSide[i] * cd->rowSize];
        memset(d->currentTop, -1, cd->topSize * sizeof(number));

        for (number j = 0; j < cd->problemSize; ++j) {
            number pj = p[j];

            for (range k = topIndex; k < topIndex + cd->rowSize; ++k) {
                if (pj == 0)
                    break;

                if (d->tmpTops[k] == r[rowIndex]) {
                    d->currentTop[k] = r[rowIndex];
                    rowIndex++;
                    pj--;
                }
            }

            topIndex += cd->rowSize;
        }

        if (d->tmpSide[i] < 10)
            write(solutionsFile, "%i  | ", d->tmpSide[i]);
        else
            write(solutionsFile, "%i | ", d->tmpSide[i]);

        writeRow(d->currentTop, cd->rowSize, cd->topSize, false, solutionsFile
            );
        write(solutionsFile, "| %lu\n", d->tmpSquare[i]);
    }

    write(solutionsFile, "\n");
}
```

1. The inputs are the address of a thread-specific data structure, the address of a thread-common structure, and a pointer to a C-style FILE structure.
2. Line 2 writes the solution number within the current thread context. It also writes an additioinal line break, and five spaces, since the next line to be written is the top vector, which is not prepended by a side vector entry like the other derived rows. Line 3 then calls Listing 4.27 to write the top vector, using the fact that the latter is stored as the bottommost element of the *tmpTops* stack, whose details are deferred until the next section.
5. Line 5 iterates over the problem size, which is precisely the number of derived rows that combine to form a combination matrix. The thread-specific data structure provided as input contains a schematic solution, given by indices into various arrays found in the thread-common data structure. Therefore each derived row must be constructed before being written. Line 6 is then an index into the top vector, and line 7 is an index into the derived row. Line 8 is the address of the semi-magic square row within all the integer partitions, and line 9 is the address of the derived row within the row class. The memory address used to compute the spelled-out derived row is a reused scratch space, hence it is re-initialized at every iteration of the for-loop in line 5. Line 10 fills this scratch space with minus ones, as negative entries are skipped in Listing 4.27.
12. Line 12 too iterates over the problem size, only this time regarding each column of the semi-magic square. Line 13 simply stores the integer partition value for the current iteration in a variable for convenience. This value is effectively the cell of the semi-magic square at row i and column j.
15. The innermost for-loop in line 15 starts from the last top index and iterates n times, where n is the row size, and its body is responsible for matching the next pj entries in the derived row against the entire j^{th} column in the top vector, which has size n. Line 16 checks if all pj entries have been matched, in which case line 17 breaks the for-loop.

18. Line 19 checks if the derived row's entry at the current row index aligns with the current top column at the index of iteration given in line 15. If so, line 20 updates the scratch memory that will be used to write the derived row, line 21 increments the row index, and line 22 decrements *pj*, which at each iteration represents the number of elements in the current square cell that still need to be matched.
25. Line 25 increments the current top index by n at each iteration of the for-loop in line 12, needed in order to traverse the top vector column-by-column.
27. Line 27 checks if the current side vector index is less than ten, that is, if it has a single digit and, if so, line 28 prepends to the derived row line the side index, followed by two spaces and a separator. If the side index requires two digits, then line 30 prepends the side index followed by a single spaces and a separator.
32. Line 32 calls Listing 4.27 to write the computed derived row without a line break. Line 33 then prepends to the line the current square row index and a new line character.
36. Line 36 simply separates this solution from the next with an additional line break.

Example 4.3.5. *Below is a sample output of Listing 4.28. In it, the row* $r = \{0, 1, 4, 2, 5, 3\}$ *is a 6-tone row that is not retrograde or retrograde inverse-invariant. The top row is the vector* $\{T_0(r), RT_3I(r), T_0I(r)\}$. *The side vector is equal to* $\{21, 16, 19\} = \{RT_3I(r), RT_1(r), RT_4(r)\}$. *The semi-magic square is the vector* $\{8, 13, 17\}$. *These are indices into the array of all partitions of six, that can be inferred from the combination matrix to be the square*

$$\left[\begin{array}{c|c|c} 1 & 3 & 2 \\ \hline 2 & 2 & 2 \\ \hline 3 & 1 & 2 \end{array}\right]. \tag{4–2}$$

```
Solution 610:

     0  1  4  2  5  3  | 0  4  1  5  2  3  | 0  5  2  4  1  3
21 | 0                 |    4  1  5        |       2        3  | 8
16 |    1           3  | 0           2     |    5     4        | 13
```

```
19 |        4  2  5      |                  3  | 0            1       | 17
```

4.3.3 Computing all possible solutions for a given combination matrix

The algorithm presented in Listing 4.30 is the main procedure in this implementation for computing solutions to the problem of self-derivation. Like other algorithms described in the previous sections, it is a recursive backtracking algorithm. It utilizes Listing 4.29 as a subroutine to test if the a given representative of a row class can be matched with the top row. Listing 4.29 is very similar to the first inner for-loop in Listing 4.28 in that it tries to match a transform of the row r against a top vector, and that matching is done column-by-column, using the partition scheme given by the current square row. The main difference between Listing 4.28 and Listing 4.29 lies in the fact that the top row is not immutable. Rather, it is some scratch memory space that gets changed every time Listing 4.29 is called. Another key difference is the fact that Listing 4.29 can fail to produce a match, whereas Listing 4.28 is only called once a solution already exists for all derived rows.

Listing 4.29. Matching a representative of a row class with a given top vector.

```
bool matchRowWithTop(number *top, const number *row, const number *partitions,   1
    number problemSize, number rowSize) {
    range topIndex = 0;                                                           2
    number rowIndex = 0;                                                          3
                                                                                  4
    for (number i = 0; i < problemSize; ++i) {                                    5
        number p = partitions[i];                                                 6
                                                                                  7
        for (range j = topIndex; j < topIndex + rowSize; ++j) {                   8
            if (p == 0)                                                           9
                break;                                                            10
                                                                                  11
            if (top[j] == row[rowIndex]) {                                        12
                top[j] = -1;                                                      13
                rowIndex++;                                                       14
```

```
                p--;                                                    15
            }                                                           16
        }                                                               17
                                                                        18
        if (p > 0)                                                      19
            return false;                                               20
                                                                        21
        topIndex += rowSize;                                            22
    }                                                                   23
                                                                        24
    return rowIndex == rowSize;                                         25
}                                                                       26
```

1. The inputs are a memory address that holds some scratch space with size equal to the top size, the address of the transform of the row which the algorithm will try to match with the top, the address of the integer partition to be used, that is, a row from the semi-magic square, the problem size, and the row size.
2. Like lines 6 and 7 in Listing 4.28, lines 2 and 3 here are indices into the top vector and row transform.
5. Line 5 iterates over the problem size, that is, over the number of columns in the top vector, similarly to line 12 in Listing 4.28. Line 6 stores in a variable the current square cell value, and the use of variable p here is identical to the use of variable p in Listing 4.28.
8. The inner for-loop in line 8, as well as the check in lines 9 and 10 are identical to lines 15 to 17 in Listing 4.28.
12. Line 12 tests that the j^{th} entry in the top vector matches the row transform at the current row index. Unlike Listing 4.28, the top vector may not be unaltered. If the j^{th} entry in the top vector was already matched to a different transform of the row r in a previous call to Listing 4.29, then the j^{th} entry in the top vector will have been set to minus one, and Listing 4.29 will fail. Otherwise, line 13 will set the j^{th} entry in the

top vector to minus one, which differs from Listing 4.28 in the sense that the latter sets a scratch memory space that had previously initialized with minus ones to the the corresponding entry in the row transform. Line 14 increases the row index, and line 15 decreases p, similarly to what Listing 4.28 does in lines 21 and 22.

19. Line 19 checks that all p elements from the row transform starting at row index have been matched with the current top vector column. If not, line 20 returns false.
22. If line 22 is reached, then the top index is increased by the row size, like line 25 in Listing 4.28, and the algorithm moves to the next top vector column.
25. If line 25 is reached, then the row transform has been matched successfully to all top vector columns. Still, the algorithm performs a sanity check and confirms that all row elements have been indeed matched by comparing the current row index with the row size.

Listing 4.30 combines Listing 4.21 and Listing 4.20 to compute all possible solutions for a given top. The general idea is to, for each potential row of a semi-magic square, try to match a transform of the row r with the current top using Listing 4.29. If that succeeds, Listing 4.30 recurses then backtracks, thus covering all of the row class and all of the partitions. Side vectors are not computed directly, but rather as a side effect of matching transforms of r with the top. The side vector is still maintained as the algorithm recurses, as it is crucial for avoiding permutations of the rows, as previously discussed, and is also used in Listing 4.28 to label each derived row in the text output of a solution.

Listing 4.30. Recursively computing all solutions for a top vector of a certain size.

```
void allSolutionsRecursive(thread_data *d, common_data *cd, FILE *                1
    solutionsFile, number currentSize) {
    if (currentSize == cd->problemSize) {                                         2
        d->counter++;                                                             3
        writeSolution(d, cd, solutionsFile);                                      4
        return;                                                                   5
    }                                                                             6
```

```
                                                                    7
length start = 0;                                                   8
                                                                    9
if (currentSize > 0) {                                              10
    start = d->tmpSquare[currentSize - 1];                          11
    memcpy(&d->tmpTops[currentSize * cd->topSize], d->currentTop, cd->  12
        topSize * sizeof(number));
}                                                                   13
                                                                    14
for (length i = start; i < cd->numPartitions; ++i) {                15
    const number *p = &cd->allPartitions[i * cd->problemSize];      16
    plus(d->currentSum, p, cd->problemSize);                        17
                                                                    18
    if (validate(d->currentSum, cd->problemSize, cd->rowSize)) {    19
        d->tmpSquare[currentSize] = i;                              20
        number rowStart = 0;                                        21
                                                                    22
        if (currentSize > 0 && start == i)                          23
            rowStart = d->tmpSide[currentSize - 1] + 1;             24
                                                                    25
        for (range j = rowStart; j < cd->classSize; ++j) {          26
            const number *r = &cd->rowClass[j * cd->rowSize];       27
                                                                    28
            if (matchRowWithTop(d->currentTop, r, p, cd->problemSize, cd->  29
                rowSize)) {
                d->tmpSide[currentSize++] = j;                      30
                allSolutionsRecursive(d, cd, solutionsFile, currentSize);   31
                currentSize--;                                      32
            }                                                       33
                                                                    34
            memcpy(d->currentTop, &d->tmpTops[currentSize * cd->topSize],   35
                cd->topSize * sizeof(number));
        }                                                           36
```

```
        }                                                                    37
                                                                             38
        minus(d->currentSum, p, cd->problemSize);                            39
    }                                                                        40
}                                                                            41
```

1. The inputs are the memory address of the thread-specific data, the memory address of the thread-common data, a pointer to a C-style FILE data structure, and the current size of the call stack.
2. Line 2 deals with the base case of the recursion by testing whether the current size has reached the problem size. If so, line 3 increases the thread-specific solution counter, line 4 calls Listing 4.28 to write the solution to the provided FILE, and line 5 returns.
8. Line 8 initializes a variable for where iteration of integer partition rows should start, similarly to line 7 in Listing 4.20.
10. Line 10 tests if the current size is greater than zero and, if so, line 11 sets the start value to the last row appended to the current semi-magic square, similarly to what line 12 in Listing 4.20 does, and line 12 here pushes onto the top history stack the current top. As derived rows are added to form a solution, the state of the current sum of square columns is always easy to backtrack by subtracting from the current sum array the last integer partition row that was appended to the square. Backtracking the current top, on the other hand, is more difficult. Every time a derived row is successfully matched against the current top, the entries where they align according to the square row at hand will be set to minus one in the current top. If all derived rows are matched and a solution is found, the entire current top scratch memory will contain only minus ones. Therefore backtracking the current top requires maintaining its history every time Listing 4.29 succeeds and a new call to Listing 4.30 is pushed onto the call stack.
15. Lines 15 to 17 do exactly what lines 14 to 16 do in Listing 4.20.
19. Similarly, lines 19 and 20 are identical to lines 18 and 19 in Listing 4.20. At this point, the current sum of square columns have been validated and a new integer partition row

has been appended to the square. With an integer partition row and a current top in place, the algorithm now searches for a representative of the row class that will fit under the current top given the scheme provided by the integer partition. Line 21 then declares a variable to represent what the start index of iteration over the representatives of the row class should be.

23. The rows of the square, seen as indices into the array of all integer partition rows, form a nondecreasing sequence. That is to avoid permutations of the square's rows, as by Cor. 4.3.3 two squares with permuted rows are equivalent. Nonetheless, squares may have several repeated rows. One simple example is the square in which all entries are equal to n/m, where n is the row size, m is the problem size, and m divides n. Line 23 tests for the case where the current square row is the same as the previous one, which can only be true if the current size is greater than zero. If that is the case, then the sequence of representatives of the row class that fall under the subsequence of repeated integer partition rows can be allowed to be nondecreasing, as well, to avoid permutations that would produce equivalent solutions. In fact, a stronger condition applies, and the sequence of representatives falling under repeated square rows can be strictly increasing. That is because the same representative of a row class cannot fit under the top row twice using the same integer partition row scheme. Line 24 then changes the start index of iteration over the representatives of the row class accordingly.

26. Line 26 iterates over the representatives of a row class, according to the row start index declared in line 21, and line 27 stores in a variable the memory address of the representative being considered.

29. Line 29 is a call to Listing 4.29. If it succeeds, line 30 sets the next row representative index in the side vector and increases the current size. Line 31 then pushes another call to Listing 4.30 onto the stack, and line 32 backtracks the current size.

35. Line 35 backtracks the current top to its previous state, using the top history stored in the top stack in line 12. Every call to Listing 4.29 is matched with the backtracking procedure in line 35.

39. Finally, line 39 backtracks the current sum of square columns with a balancing call to Listing 4.18, similarly to what Listing 4.20 does in its line 24.

4.4 Creating work data for use in concurrent threads

This section outlines the procedures used for allocating the data structures to be used as inputs to Listing 4.30. Pre-allocating and reusing scratch memory locations is absolutely critical for the performance of all algorithms described above. Unnecessary calls to allocate and deallocate memory can significantly increase the time it takes for Listing 4.30 to execute as row size and problem size grow. Listing 4.31 describes how thread-common data is allocated, and every call to Listing 4.31 should be balanced with a call to Listing 4.32 when resources are no longer needed. Listing 4.31 allocates two arrays with size equal to the row class size. These are namely the retrograde and the retrograde inverse maps. As the name suggests, theses are mappings between indices within the row class. The retrograde map takes an element into its retrograde, and the retrograde inverse map takes an element into its retrograde inverse. These maps will be used in conjunction with Listing 4.40, thus further details are deferred until the next section.

Listing 4.31. Allocating thread-common data.

```
void createCommonData(common_data *cd, int argc, char *argv[]) {          1
    number problemSize = (number) strtol(argv[1], NULL, 10);              2
    number rowSize = argc - 2;                                            3
    number *row = malloc(rowSize * sizeof(number));                       4
                                                                          5
    for (number i = 0; i < rowSize; ++i)                                  6
        row[i] = (number) strtol(argv[i + 2], NULL, 10);                  7
                                                                          8
    cd->problemSize = problemSize;                                        9
```

```
cd->rowSize = rowSize;                                                    10
cd->topSize = problemSize * rowSize;                                      11
                                                                          12
bool rInvariant = isRetrogradeInvariant(row, rowSize);                    13
bool riInvariant = isRetrogradeInverseInvariant(row, rowSize);            14
                                                                          15
cd->isInvariant = rInvariant || riInvariant;                              16
cd->rowClass = getRowClass(&cd->classSize, row, rowSize, cd->isInvariant); 17
cd->rMap = malloc(cd->classSize * sizeof(number));                        18
cd->riMap = malloc(cd->classSize * sizeof(number));                       19
cd->allPartitions = readAllPartitions(&cd->numPartitions, problemSize,    20
    rowSize);
cd->allTops = readAllTops(&cd->numTops, problemSize, cd->classSize);      21
                                                                          22
if (rInvariant) {                                                         23
    for (int i = 0; i < rowSize; ++i)                                     24
        cd->rMap[i] = cd->rowClass[(i + 1) * rowSize - 1];                25
                                                                          26
    for (int i = rowSize; i < cd->classSize; ++i)                         27
        cd->rMap[i] = cd->rowClass[(i + 1) * rowSize - 1] + rowSize;      28
} else {                                                                  29
    for (int i = 0; i < rowSize; ++i)                                     30
        cd->rMap[i] = cd->rowClass[(i + 1) * rowSize - 1] + rowSize;      31
                                                                          32
    for (int i = rowSize; i < cd->classSize; ++i)                         33
        cd->rMap[i] = cd->rowClass[(i + 1) * rowSize - 1];                34
}                                                                         35
                                                                          36
for (int i = 0; i < cd->classSize; ++i)                                   37
    cd->riMap[i] = (cd->rMap[i] == 0 || cd->rMap[i] == rowSize)           38
                 ? modulo(cd->rMap[i] + rowSize, cd->classSize)           39
                 : modulo(-cd->rMap[i], cd->classSize);                   40
                                                                          41
```

```
    free(row);                                                    42
}                                                                 43
```

1. The inputs are the memory address of a thread-common data structure, and the number of arguments provided in the command line, and the array of such arguments. Listing 4.31 allocates the internal pointers in a thread-common data structure, but also parses the command line inputs, which are a problem size and a row, separated by spaces. The row does not need to be in its canonical form.
2. Line 2 retrieves the problem size from the first element in arguments array. Note that the zeroth element is the application name. Line 3 computes then the row size, which is just the number of arguments minus two, namely the row size and the application name. Line 4 allocates some memory space which will be used to parse the row provided in the remaining arguments.
6. Line 6 iterates through the row size and line 7 parses each row entry provided, storing them in the memory allocated in line 4.
9. Line 9 updates the provided pointer to a thread-common data structure with the problem size, line 10 updates the row size, and line 11 computes and updates the top size.
13. Line 13 is a call to Listing 4.9 to determine if the row is retrograde-invariant, and line 14 is similarly a call to Listing 4.10 to determine if the row is retrograde inverse-invariant. Note that the row does not need to be in its canonical form when determining retrograde or retrograde inverse-invariance.
16. Line 16 updates the data pointer with information whether the row is retrograde or retrograde inverse-invariance. The specific information whether retrograde invariance occurs under inversion or not will be needed to compute retrograde and retrograde inverse maps in Listing 4.31, but will not be needed in the algorithms that depend on a thread-common data structure. Line 17 is a call to Listing 4.11, which allocates and returns the memory address of the row class array. Line 18 allocates the memory that will hold the retrograde map, and line 19 allocates the memory that will hold

the retrograde inverse map. Lines 20 and 21 are calls to respectively Listing 4.16 and Listing 4.24, both returning allocated memory addresses.

23. Line 23 tests if the row is retrograde-invariant and, if so, line 24 iterates over the size of the row, filling the first half of the retrograde map in line 25. Note that, since the row is retrograde-invariant, the size of the row class is twice the size of the row.

27. Line 27 continues iterating over the class size, filling the second half of the retrograde map in line 28.

30. If the row is retrograde inverse-invariant, line 30 iterates over the size of the row, filling the first half of the retrograde map in line 31. Note that here the first and second halves of the retrograde map are reversed in regard to the retrograde-invariant case, that is, line 31 does the same computation that line 28 does.

33. Line 33 continues iterating over the class size, filling the second half of the retrograde map in line 34, with the same computation in line 25.

37. Line 37 iterates over the entire size of the row class to set the values in the retrograde inverse map. Line 38 tests if the corresponding retrograde map entry contains the T_0 or the T_0I transforms of the row, in which case line 39 assigns these indices such that $T_0 \mapsto T_0I$ and $T_0I \mapsto T_0$. If not, then line 40 assigns the other indices such that $T_x \mapsto T_{-x}$.

42. Finally, line 42 deallocates the space created in line 4, as the original row can now be retrieved from the row class array.

Example 4.4.1. *Let r be a retrograde-invariant row with size $n = 6$. Then the row class array computed by Listing 4.11 will contain a sequence of transforms of r equal to $C = \{T_0, \cdots, T_5, T_0I, \cdots, T_5I\}$. The retrograde map of the row class of r will be the sequence containing the last entry in each transform of r in C. This sequence will always be equal to $C_R = \{3, 4, 5, 0, 1, 2, 9, 10, 11, 6, 7, 8\}$. Since r is retrograde-invariant, the sequence $\{0, \cdots, 5\}$ of indices of r maps to itself under the retrograde operator, that is, R is the permutation* (0 5) (1 4) (2 3) *when seen as a permutation of indices. Since $r = RT_x(r)$ by*

assumption, Tx must be a transposition that, seen as a permutation of pitch-classes, comprises solely 2-cycles. This is only possible if $x = 3$. Since $T_3 = (0\ 3)\ (1\ 4)\ (2\ 5)$, the array given by C_R, when seen as a permutation of indices of the row class, is in fact the map

$$T_0 \mapsto T_3, \cdots, T_5 \mapsto T_2, T_3I \mapsto T_0I, \cdots, T_5I \mapsto T_2I \ . \tag{4–3}$$

Given an index in C, the retrograde map takes that index to another index in C such that a transform of r maps to itself under $RT_{n/2}$. Let $C_I = \{6, 11, 10, 9, 8, 7, 0, 5, 4, 3, 2, 1\}$ be the array that maps a transform of r in the row class array to its inverse. Seen as a permutation, $C_I = (0\ 6)\ (1\ 11)\ (2\ 10)\ (3\ 9)\ (4\ 8)\ (5\ 7)$. The retrograde inverse map is then the composition of C_R with C_I, that is

$$C_R \circ C_I = \{9, 8, 7, 6, 11, 10, 3, 2, 1, 0, 5, 4\} \ . \tag{4–4}$$

Example 4.4.2. *Let $r = \{0, 2, 3, 4, 5, 1\}$, with n, C and C_I as in Ex. 4.4.1. Then r is retrograde inverse-invariant. The sequence containing the last entry in each transform of r in C will depend on which x satisfies $r = RT_xI(r)$. Since T_xI, seen as a permutation, must comprise only 2-cycles, x is necessarily odd. In particular, $x = 1$ for this choice of r, but unlike retrograde maps, retrograde inverse maps depend on the row at hand. More precisely, on the transposition index in the operation that maps the row to itself. Here, $C_R = \{7, 8, 9, 10, 11, 6, 5, 0, 1, 2, 3, 4\}$ and thus*

$$C_R \circ C_I = \{5, 4, 3, 2, 1, 0, 7, 6, 11, 10, 9, 8\} \ . \tag{4–5}$$

Listing 4.32 below describes how the data allocated in Listing 4.31 is deallocated. It simply frees all the memory that had been previously allocated in the heap. Due to their simplicity, further details are omitted in the next three routines.

Listing 4.32. Deallocating thread-common data.

```
void destroyCommonData(common_data *cd) {                    1
    free(cd->rowClass);                                      2
```

```
    free(cd->rMap);                                                    3
    free(cd->riMap);                                                   4
    free(cd->allPartitions);                                           5
    free(cd->allTops);                                                 6
}                                                                      7
```

Listing 4.33 is a very simple procedure, as well. As it depends on a previous call to Listing 4.31, all thread-common data has already been allocated, and sizes computes. Listing 4.33 then allocates the necessary scratch areas that cannot be shared between different thread contexts, and sets the thread-specific solution counter to zero.

Listing 4.33. Allocating thread-specific data.

```
void createThreadData(thread_data *d, common_data *cd) {               1
    d->counter = 0;                                                    2
    d->tmpSide = malloc(cd->problemSize * sizeof(number));             3
    d->currentTop = malloc(cd->topSize * sizeof(number));              4
    d->tmpTops = malloc(cd->problemSize * cd->topSize * sizeof(number));5
    d->currentSum = malloc(cd->problemSize * sizeof(number));          6
    d->tmpSquare = malloc(cd->problemSize * sizeof(length));           7
}                                                                      8
```

Similarly to Listing 4.32, Listing 4.34 deallocates the heap memory that was allocated in Listing 4.33. Here too, every call to Listing 4.33 should be balanced with a call to Listing 4.34 when resources are no longer needed.

Listing 4.34. Deallocating thread-specific data.

```
void destroyThreadData(thread_data *d) {                               1
    free(d->tmpSide);                                                  2
    free(d->currentTop);                                               3
    free(d->tmpTops);                                                  4
    free(d->currentSum);                                               5
    free(d->tmpSquare);                                                6
}                                                                      7
```

4.5 Defining a top-level procedure for computing all solutions for a given row class

This section outlines Listing 4.42, a procedure that wraps Listing 4.30, providing it with the necessary scaffolding to output all solutions computed to a text file. In a single-threaded context, Listing 4.42 will be furnished with all possible tops for a given problem size. In a multi-threaded context, however, Listing 4.42 will be given a range of tops to be processed by each thread. Listing 4.22 above computes the minimal subset of tops necessary for the operation of Listing 4.42. As previously discussed, the number of possible tops becomes impractically large as the problem size grows. Many algorithms described in this section deal with skipping tops that can be achieved by retrograding a top that has already been seen. Listing 4.42 itself handles computing tops that cannot be achieved from the results of Listing 4.22. Listing 4.35 creates a file to which solutions will be written.

Listing 4.35. Creating a text file to hold all thread-specific solutions.

```
FILE *createSolutionsFile(number *row, number problemSize, number rowSize,      1
    length startTop, length endTop) {
    char fileName[64];                                                          2
    sprintf(fileName, "all_solutions_%i", problemSize);                         3
    char offset = problemSize < 10 ? 15 : 16;                                   4
                                                                                5
    for (number i = 0; i < rowSize; ++i) {                                      6
        sprintf(&fileName[offset], "_%i", row[i]);                              7
        offset += row[i] < 10 ? 2 : 3;                                          8
    }                                                                           9
                                                                                10
    sprintf(&fileName[offset], "_%lu_%lu.txt", startTop, endTop);               11
                                                                                12
    return fopen(fileName, "wt");                                               13
}                                                                               14
```

1. The inputs are the row in canonical form, the problem size, the row size, and the start and end indices into the all tops array. Since this file will contain solutions only for the

tops between theses indices, the indices are used in the name of the file, to distinguish it from other files written by other threads.

2. Line 2 creates a buffer to store the file name, and line 3 writes the first part of the name into the buffer. Line 4 computes the current offset from which the rest of the name should start, which depends on the problem size.
6. Line 6 iterates over the row size, appending in line 7 each row entry to the file name, and updating in line 8 the offset declared in line 4.
11. Line 11 appends to the file name the start and end top indices.
13. Line 13 returns a pointer to a C-style FILE.

The procedure for determining whether a top can be achieved from a previously processed top by some RT_nI transform differs if the row is retrograde or retrograde inverse-invariant or not. Listing 4.40 outlines the procedure for retrograde or retrograde inverse-invariant rows, whereas Listing 4.39 outlines the procedure for rows otherwise. The next three algorithms below are used as subroutines in Listing 4.39.

Listing 4.36. Computing the index of the inverse row within a row class.

```
range getInverse(range num, number rowSize) {                    1
    if (num == 0 || num == rowSize * 2)                          2
        return num + rowSize;                                    3
                                                                 4
    if (num == rowSize || num == rowSize * 3)                    5
        return num - rowSize;                                    6
                                                                 7
    if (num < rowSize * 2)                                       8
        return rowSize * 2 - num;                                9
                                                                 10
    return rowSize * 6 - num;                                    11
}                                                                12
```

1. The inputs are the index of the row within the row class, and the row size.

2. Line 2 tests if the given transform is either T_0 or RT_0, in which case line 3 returns respectively $T_0 I$ or $RT_0 I$.
5. Line 5 does the opposite of line 2, that is, it tests if the given transform is either $T_0 I$ or $RT_0 I$, in which case line 6 returns respectively T_0 or RT_0.
8. Line 8 tests if the given transform is not retrograded, in which case line 9 returns an index within the first half of the row class size.
11. If line 11 is reached, then the given transform is retrograded, so the the index returned lies within the second half of the row class size.

Listing 4.37 is a lot simpler than Listing 4.36. It simply takes an index in the first half of the row class size and returns the corresponding index in the second half, and vice-versa.

Listing 4.37. Computing the index of the retrograde row within a row class.

```
range getReverse(range num, number rowSize) {                    1
    return (num + rowSize * 2) % (rowSize * 4);                  2
}                                                                3
```

Listing 4.38 is even simpler and just combines calls to Listing 4.36 and Listing 4.37 to return the retrograde inverse index of the provided input index.

Listing 4.38. Computing the index of the retrograde inverse row within a row class.

```
range getReverseInverse(range num, number rowSize) {             1
    return getInverse(getReverse(num, rowSize), rowSize);        2
}                                                                3
```

Listing 4.39 is designed to filter tops that can be achieved by transforming some previously seen top. It relies on the order in which Listing 4.22 outputs tops to filter out top invariances under the retrograde operation.

Listing 4.39. Determining whether a top has already been seen.

```
bool seen(const number *combo, common_data *cd) {                1
    number last = cd->problemSize - 1;                           2
    bool I = (combo[last] % (cd->rowSize * 2)) > (cd->rowSize - 1);  3
```

```
    number T = combo[last] % cd->rowSize;                                    4
    range back, transform, offset;                                           5
                                                                             6
    for (int i = 0; i <= last; ++i) {                                        7
        if (I) {                                                             8
            back = getReverseInverse(combo[last - i], cd->rowSize);          9
            transform = modulo(back + T, cd->rowSize);                       10
        } else {                                                             11
            back = getReverse(combo[last - i], cd->rowSize);                 12
            transform = modulo(back - T, cd->rowSize);                       13
        }                                                                    14
                                                                             15
        offset = (back / cd->rowSize) * cd->rowSize;                         16
                                                                             17
        if (combo[i] < transform + offset)                                   18
            return false;                                                    19
        else if (combo[i] > transform + offset)                              20
            return true;                                                     21
    }                                                                        22
                                                                             23
    return false;                                                            24
}                                                                            25
```

1. The inputs are the memory addresses of a top row, and of a thread-common data structure. The output is a boolean representing whether the top has already been seen.
2. Line 2 stores in a variable the index of the last row in the top. Line 3 determines if the last top entry is inverted, that is, if it lies in the second or fourth quarters of the row class size. Line 4 determines if the last top entry's index of transposition. Line 5 declares three variables that will be needed in the for-loop in line 7.
7. Line 7 iterates through all entries in the top, retrograding or retrograde inverting it. Line 8 checks if the top is inverted, and if so, line 9 calls Listing 4.38 and stores the result.

Line 10 uses the index of transposition computed in line 4 to determine the transposition index of the top entry at hand, modulo the row size.

12. Line 12 is the equivalent to line 9 when the top is not inverted, that is, it stores the result of a call to Listing 4.37 instead. Line 13 is similarly the equivalent to line 10, and accordingly subtracts modulo the row size the value computed in line 4 instead of adding.

16. Line 16 essentially computes in which quarter of the row class size the current retrograded or retrograde-inverted top entry lies. If the row size is n, it stores a value $k \in \{0, n, 2n, 3n\}$. These values correspond respectively to the start indices of the transpositions, inversions, retrogrades and retrograde inversions within the row class.

18. Line 18 checks whether the current top entry, after being retrograded or retrograde inverted, and transposed so that the last (now first) top entry starts with zero, is less than the top entry given at the current index of iteration, that is the non-transformed top entry read form left to right. If the former is less than the latter, then this top or some transform thereof has not been processed yet, as tops are traversed in lexicographic order. In that case, line 19 returns false.

20. Line 20 tests the opposite case, where the transformed top entry seen from right to left is greater than the non-transformed entry seen form left to right. If that is the case, this top or some transform that leads to it has already been processed and thus line 21 returns true.

24. If line 24 is reached, then every transformed top entry seen from right to left is equal to the corresponding non-transformed entry seen form left to right, which indicates that the top is retrograde or retrograde inverse-invariant. The algorithm then returns false. Since the first element of every top is always zero, line 24 can only be reached if the top at hand is being seen for the first time. Note that there are exceptions to the rule that every top's first element is zero, which shall be discussed in conjunction with Listing 4.42.

The purpose of Listing 4.40 is also to filter out tops that can be achieved from a previously seen top through some transformation, only that Listing 4.40 handles the cases where the row is retrograde or retrograde inverse-invariant. The general idea is the same: compute the retrograde or retrograde inverse-invariant transform of the top, transposing it so it starts with zero. Then compare with the original top, read from left to right, and return true is there is an entry in the transformed top greater than the corresponding entry in the original top, or if there are no entries in the transformed top less than the corresponding entry in the original top, implying the top is retrograde or retrograde inverse-invariant. Listing 4.40 utilizes retrograde and retrograde inverse maps contained in the thread-common data structure, as filtering out tops may depend on the row at hand if the row is retrograde or retrograde inverse-invariant.

Listing 4.40. Determining whether a top has not yet been seen for retrograde or retrograde inverse-invariant rows.

```
bool unseen(const number *top, common_data *cd) {                    1
    number last = cd->problemSize - 1;                               2
    number offset = cd->rMap[top[last]];                             3
    bool invert = offset >= cd->rowSize;                             4
                                                                     5
    if (invert)                                                      6
        offset = cd->riMap[top[last]];                               7
                                                                     8
    for (number i = 1; i < cd->problemSize; ++i) {                   9
        number front = top[i];                                       10
        number back = invert ? cd->riMap[top[last - i]] : cd->rMap[top[last -  11
            i]];
                                                                     12
        if (back < cd->rowSize)                                      13
            back = modulo(back - offset, cd->rowSize);               14
        else {                                                       15
            back = modulo(back - offset - cd->rowSize, cd->rowSize); 16
            back += cd->rowSize;                                     17
```

```
        }                                                                    18
                                                                             19
        if (back > front)                                                    20
            return true;                                                     21
        else if (back < front)                                               22
            return false;                                                    23
    }                                                                        24
                                                                             25
    return true;                                                             26
}                                                                            27
```

1. The inputs are the memory addresses of a top row, and of a thread-common data structure. Unlike Listing 4.40, the output is a boolean representing whether the top has not yet been seen.
2. Line 2 stores in a variable the index of the last row in the top. Line 3 uses the retrograde map to compute the transposition index of the top's last entry, assuming that the retrograde of the top is not inverted. Line 4 checks if the top's last entry is inverted and stores the result.
6. Line 6 uses the result of the test computed in line 4, and if the retrograde of the top is indeed inverted, line 7 corrects the wrong assumption made in line 3, using this time the retrograde inverse map to update the transposition index of the top's last entry.
9. Line 9 iterates through the entries in the top, skipping the comparison between its first and last entries. Since the row class does not contain any retrograde transforms, the result of the comparison between the first and last entries here would be trivial. Line 10 stores the current leftmost entry, and line 11 stores the current rightmost entry, using the value computed in line 4 to determine if the retrograde or the retrograde inverse map should be used to retrieve the rightmost entry.
13. Line 13 tests if the current rightmost entry is not inverted, in which case line 14 transposes it using the previously computed transposition index.

16. If the current rightmost entry is in fact inverted, lines 16 and 17 use the previously computed transposition index to transpose and invert the current rightmost entry.
20. Line 20 simply tests if the current rightmost entry is greater than the current leftmost entry and, if so, line 21 returns true.
22. Line 22 tests the opposite scenario, and line 23 returns false if the current rightmost entry is less than the current leftmost entry.
26. If line 26 is reached, then the top is is retrograde or retrograde inverse-invariant, so the algorithm returns true.

Listing 4.41 is used as a subroutine in Listing 4.42 copy the top to be processed by Listing 4.30 into the thread-specific data structure's scratch area for the top vector. A top, as computed in Listing 4.22, is an array of indices into the row class array. Listing 4.41 iterates through the problem size, and copies the appropriate row transform from the row class array into the current column offset of the top row scratch area.

Listing 4.41. Refreshing the top row scratch area.

```
void fillTop(thread_data *d, common_data *cd, const number *combo) {
    for (number i = 0; i < cd->problemSize; ++i)
        memcpy(&d->currentTop[i * cd->rowSize],
               &cd->rowClass[combo[i] * cd->rowSize],
               cd->rowSize * sizeof(number));
}
```

Listing 4.42 processes a range of tops and writes all solutions found into a text file. It relies on thread-specific data, so its execution should be constrained to a single thread. Multiple parallel calls to Listing 4.42 can be made from different thread contexts, provided that each thread maintains their own thread-specific data structures. Listing 4.42 also expands the provided range of tops when the row at hand is not retrograde or retrograde inverse-invariant. That is accomplished by processing the range of tops twice, the second time replacing the first entry in the top, which is always zero, with half the row class size, which is the index of the RT_0 transform. The reason is because some tops cannot be achieved by transforming the

results of Listing 4.22 alone. As a simple example, consider the case where r is a row of size n that is not retrograde or retrograde inverse-invariant, such that the size of its row class is $4n$. Let $T = \{0, 24\} = \{T_0, RT_0\}$ be a top. In particular, T is one of the tops in the output of Listing 4.22. However, the retrograde of T is the top

$$R(T) = \{R(RT_0), R(T_0)\} = \{T_0, RT_0\} = \{0, 24\} = T \ . \tag{4–6}$$

This shows that, in order to achieve some tops in which the first entry is a retrograde transform, such as $\tilde{T} = \{24, 0\}$, it is necessary that the top's first entry be a retrograde transform and not zero. It is sufficient that this first entry be $2n = RT_0$, since other tops in which the first entry is a retrograde transform can then be achieved by transformation under transposition, inversion or both. To see this, let m be the problem size and let

$$G = \underbrace{RT_xI \times \cdots \times RT_xI}_{m \text{ times}} \ . \tag{4–7}$$

The action of RT_xI on G by right multiplication induces orbits of two sizes, depending on whether $g \in G$, which is a top, is retrograde or retrograde inverse-invariant. If so, the orbit has size $2n$, otherwise the orbit has size $4n$. In the latter case, all orbits have a representative in which the first entry is zero. In the former case, all orbits have either a representative in which the first entry is zero, or one in which the first entry is $2n$.

Listing 4.42. Writing a text file with all solutions for a row class within a range of tops.

```
length writeSolutions(thread_data *d, common_data *cd, length startTop, length endTop) {
    length topCounter = 0;
    FILE *solutionsFile = createSolutionsFile(&cd->rowClass[0], cd->problemSize, cd->rowSize, startTop, endTop);

    for (range first = 0; first < cd->classSize; first += cd->classSize / 2) {
        for (length i = startTop; i < endTop; ++i) {
            number *combo = &cd->allTops[i * cd->problemSize];
```

```
            combo[0] = first;                                                    8
                                                                                 9
            if (cd->isInvariant && first >= cd->classSize / 2)                   10
                break;                                                           11
                                                                                 12
            if (cd->isInvariant && !unseen(combo, cd))                           13
                continue;                                                        14
                                                                                 15
            if (!cd->isInvariant && seen(combo, cd))                             16
                continue;                                                        17
                                                                                 18
            fillTop(d, cd, combo);                                               19
            memcpy(d->tmpTops, d->currentTop, cd->topSize * sizeof(number));     20
            memset(d->currentSum, 0, cd->problemSize * sizeof(number));          21
            allSolutionsRecursive(d, cd, solutionsFile, 0);                      22
            topCounter++;                                                        23
        }                                                                        24
    }                                                                            25
                                                                                 26
    fclose(solutionsFile);                                                       27
    return topCounter;                                                           28
}                                                                                29
```

1. The inputs are the memory addresses of a thread-common and a thread-specific data structures, as well as the start and end top indices to be covered. The output is the actual number of tops that were processed.
2. Line 2 declares a variable to hold the total number of processed tops. Line 3 is a call to Listing 4.35 to create and store the pointer to the C-style FILE where to which the computed solutions will be written.
5. Line 5 iterates twice over the row class size to change the top's first entry from zero to half the row size in the second iteration, as discussed above. Line 6 iterates over

the provided range of tops. Line 7 stores the memory address of the current top being processed, and line 8 changes the top's first entry.

10. Line 10 checks if the row is retrograde or retrograde inverse-invariant and the current top's first entry is not zero, in which case line 11 breaks the for-loop.

13. Line 13 checks if the row is retrograde or retrograde inverse-invariant and the current top can be achieved by transforming another top already seen by calling Listing 4.40, in which case line 14 skips this iteration.

16. Line 16 checks if the row is not retrograde or retrograde inverse-invariant and the current top can be achieved by transforming another top already seen by calling Listing 4.39, in which case line 17 skips this iteration.

19. Line 19 is a call to Listing 4.41 to refresh the top row's scratch area. Line 20 copies the current top into the bottom of the top stack, and line 21 sets all entries in the scratch area for square column sums to zero. Line 22 then calls Listing 4.30 to compute and output solutions, and line 23 increases the top counter.

27. Line 27 closes the solutions file and line 28 returns the number of tops actually processed.

4.5.1 Directions for the future

Vectorize vector addition and subtraction.

APPENDIX A

A.1 ELEMENTARY TWELVE-TONE THEORY

Definition A.1.1. *Let x and y be arbitrary pitches. The* **ordered pitch interval** *between x and y is given by*

$$i(x, y) = y - x \ . \tag{A–1}$$

The **ordered pitch-class interval** *between x and y is given by*

$$i(\bar{x}, \bar{y}) = \bar{y} - \bar{x} \ . \tag{A–2}$$

The **unordered pitch interval** *between x and y is given by*

$$\bar{i}(x, y) = |x - y| \ . \tag{A–3}$$

The **unordered pitch-class interval**, *or simply interval class, between x and y is given by*

$$\bar{i}(\bar{x}, \bar{y}) = \min\{i(\bar{x}, \bar{y}), i(\bar{y}, \bar{x})\} \ . \tag{A–4}$$

Example A.1.2. *Put $x = 43$ and $y = -13$. Then*

$$i(x, y) = -56 \tag{A–5}$$

$$i(\bar{x}, \bar{y}) = \overline{11} - \bar{7} = \bar{4} \tag{A–6}$$

$$\bar{i}(x, y) = 56 \tag{A–7}$$

$$\bar{i}(\bar{x}, \bar{y}) = \bar{4} \tag{A–8}$$

Whenever the context is clear, we shall drop quotient notation and subscripts. In most situations, we are interested in the interval class between x and y, in which case we will simply write $i(7, -1) = 4$. Interval classes can also be seen graph-theoretically as the edge connecting two members of a pitch-class set, displayed clockwise.

APPENDIX B
ELEMENTARY GROUP THEORY

REFERENCES

[1] D. Starr, *Perspectives of New Music* **23**, 180 (1984). Available from: http://www.jstor.org/stable/832915.

[2] J. Rahn, *In Theory Only* **1**, 10 (1975). Available from: http://quod.lib.umich.edu/g/genpub/0641601.0001.002.

[3] R. Morris, *Composition with Pitch-classes: A Theory of Compositional Design* (Yale University Press, 1987).

[4] C. Scotto, *Perspectives of New Music* **38**, 169 (2000). Available from: http://www.jstor.org/stable/833592.

[5] D. Kowalski, *In Theory Only* **9**, 27 (1987). Available from: http://name.umdl.umich.edu/0641601.0009.005.

[6] D. Lewin, *Journal of Music Theory* **10**, 276 (1966). Available from: http://www.jstor.org/stable/843244.

[7] R. Morris, *In Theory Only* **7**, 15 (1976). Available from: http://name.umdl.umich.edu/0641601.0002.007.

[8] D. Lewin, *In Theory Only* **7**, 8 (1976). Available from: http://name.umdl.umich.edu/0641601.0002.007.

[9] M. Babbitt, *In Theory Only* **7**, 9 (1976). Available from: http://name.umdl.umich.edu/0641601.0002.007.

[10] H. Fripertinger, P. Lackner, *Journal of Mathematics and Music* **9**, 111 (2015). Available from: http://dx.doi.org/10.1080/17459737.2015.1070088.

[11] A. Mead, *Perspectives of New Music* **27**, 180 (1989). Available from: http://www.jstor.org/stable/833268.

[12] D. Martino, *Journal of Music Theory* **5**, 224 (1961). Available from: http://www.jstor.org/stable/843226.

[13] J. J. Rotman, *Advanced Modern Algebra* (Prentice Hall, 2002).

[14] D. S. Dummit, R. M. Foote, *Abstract Algebra* (John Wiley & Sons, 2004).

[15] A. Tucker, *Mathematics Magazine* **47**, 248 (1974). Available from: http://www.jstor.org/stable/2688047.

[16] M. Aigner, *Discrete Mathematics* (American Mathematical Society, 2007).

[17] D. L. Reiner, *The American Mathematical Monthly* **92**, 51 (1985). Available from: http://www.jstor.org/stable/2322196.

BIOGRAPHICAL SKETCH

Luis F. Vieira Damiani earned in 2002 a Bachelor of Music degree in violin performance from Rio Grande do Sul's Federal University, under the orientation of Prof. Marcello Guerchfeld, a former student of Galamian. In 2003, Luis entered Porto Alegre Symphony, with which he performed as soloist in the 2005 season. In 2004, he was appointed Assistant Concertmaster of the Catholic University of Rio Grande do Sul's Philharmonic Orchestra, and from 2009 until 2011, Luis was a member of Minas Gerais Philharmonic, one of the top Brazilian groups of its kind. In Italy, he studied with Boris Belkin and Giuliano Carmignola at the Accademia Chigiana. Other teachers include Jennifer John at the Aspen Music Festival in 1996, and Kurt Sassmannshaus at University of Cincinnati's College-Conservatory of Music in 1999. As a performer, Luis researched, engraved and premiered works by 20th Century Southern Brazilian composers Bruno Kiefer and Armando Albuquerque, as well as recorded in 2006 of the *Duo for Violin and Cello* (2000) by composer Pablo Castellar, a recording that won the Açorianos Prize from the Municipal Office of Culture of Porto Alegre in 2008.

As a composer, Luis wrote in 2008 the original music for the motion picture *O Guri*. For television, his commissioned works include the soundtrack for the short film *O Sabiá*, aired in 2010. In the same year, he received the prestigious Classical Composition Award from the National Foundation of Arts in Brazil for his *Solo Violin Suite*, making him take a turn from a well-established orchestral career into pursuing graduate studies in the USA. Luis' awards since include Best Feature Soundtrack at the 6th Cinefantasy International Fantastic Film Festival in 2011 and the University of South Florida's 2012 Percussion Composition Prize, among numerous academic awards. In 2013, he received a Master of Music degree in music composition from the University of South Florida, under the orientation of Dr. Baljinder Sekhon. In the same year, Luis was awarded a fellowship to pursue Ph.D. studies in music composition at the University of Florida, where he currently works as a graduate teaching assistant, under the orientation of Dr. James Paul Sain.