GENERALIZED SELF-DERIVATION

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 Maria Inês and Susana

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help, support and encouragement received over the years from Dr. James Paul Sain. Dr. Sain has helped me seeing the world in many different ways, motivating me to pursue areas of knowledge new to me both within and outside the real of music composition, always with immense affection, care, academic integrity and respect. My gratitude to Dr. Sain cannot be overstated, and his teachings and influence have changed my life forever. Thank you from the bottom of my heart.

I also extend my gratitude to Dr. Paul Richards, Dr. Silvio dos Santos and Dr. Kevin Keating, who have helped me with every aspect of this dissertation and beyond, as well as to Dr. Murali Rao, who introduced me to the world of mathematics and with whom I have had the most amazing conversations. I am specially thankful to Dr. Beverly Sanders for helping me become a computer scientist.

I am indebted to Dr. Baljinder Sekhon, Dr. Ciro Scotto and Dr. Robert Morris for showing me a view of music theory and composition that has completely changed the way I now hear the world, as well as contributed to the choice of self-derivation as the topic of this dissertation. I am equally grateful to Dr. David Kowalski for all the influence his work had on my understanding of self-derivation.

Without the constant and unconditional love and dedication from my wife Maria Inês and mother Susana, I would not have been able to withstand the difficulties, and obtain the achievements that I have over the years. These accomplishments are also theirs, as were the hardships. I love you forever.

TABLE OF CONTENTS

LIST OF FIGURES

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 SELF-DERIVATION

By

Luis Felipe Vieira Damiani

December 2022

Chair: 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 works of Daniel Starr and David Kowalsky in the field of twelve-tone theory, generalizing the procedure to arbitrary equal-temperament systems. An entirely new algorithm that computes combination matrices of self-derivation for arbitrary row and problem sizes is introduced, representing a major contribution to the fields of music composition and music theory. This algorithm is based on a proposed model for understanding self-derivation from the standpoint of semi-magic squares. A myriad of musical compositions are specially devised with the intent of illustrating the real-life use of the algorithm, and to provide criticism as to where such compositional practices situate within the realm of 21^{st}-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

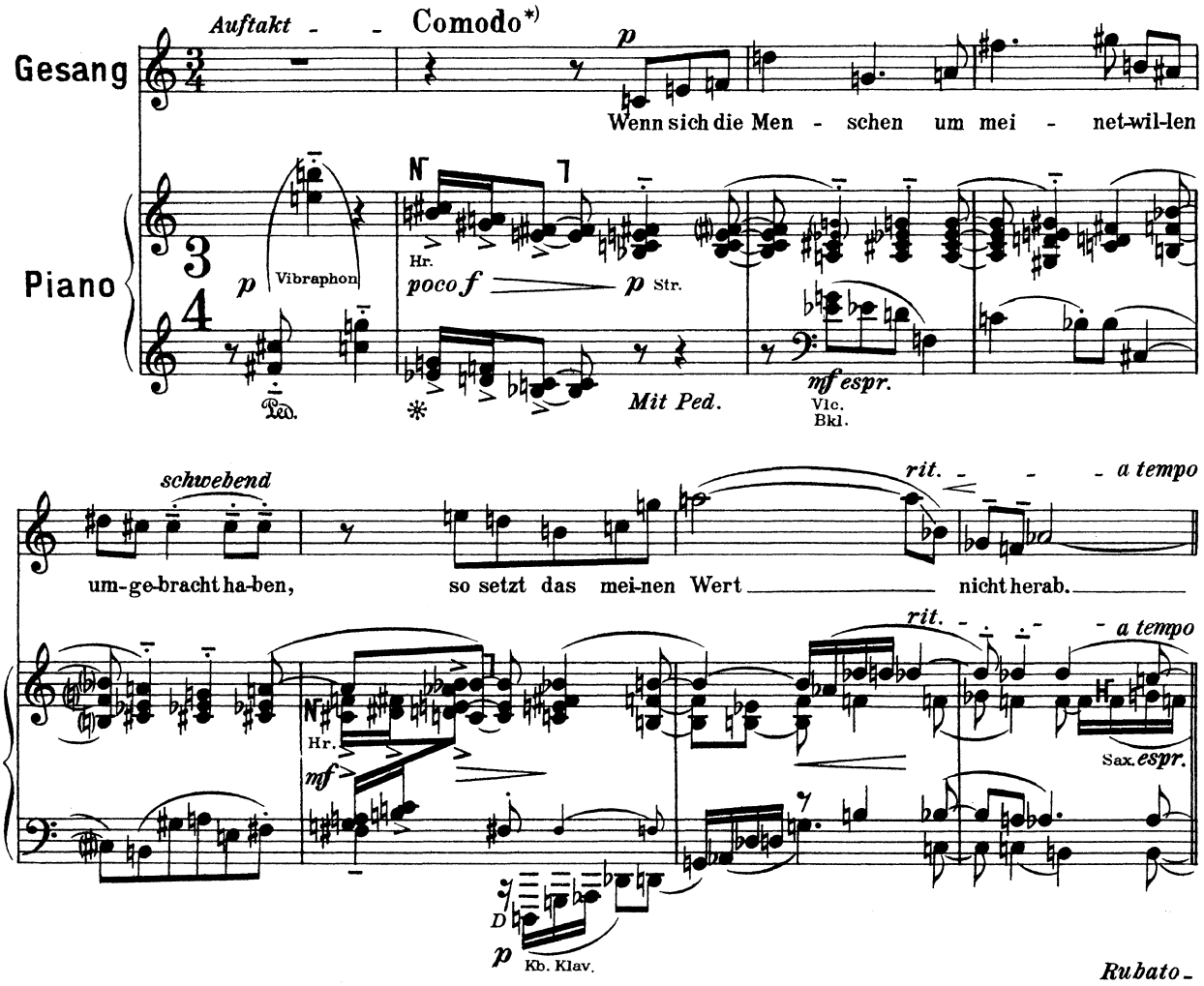
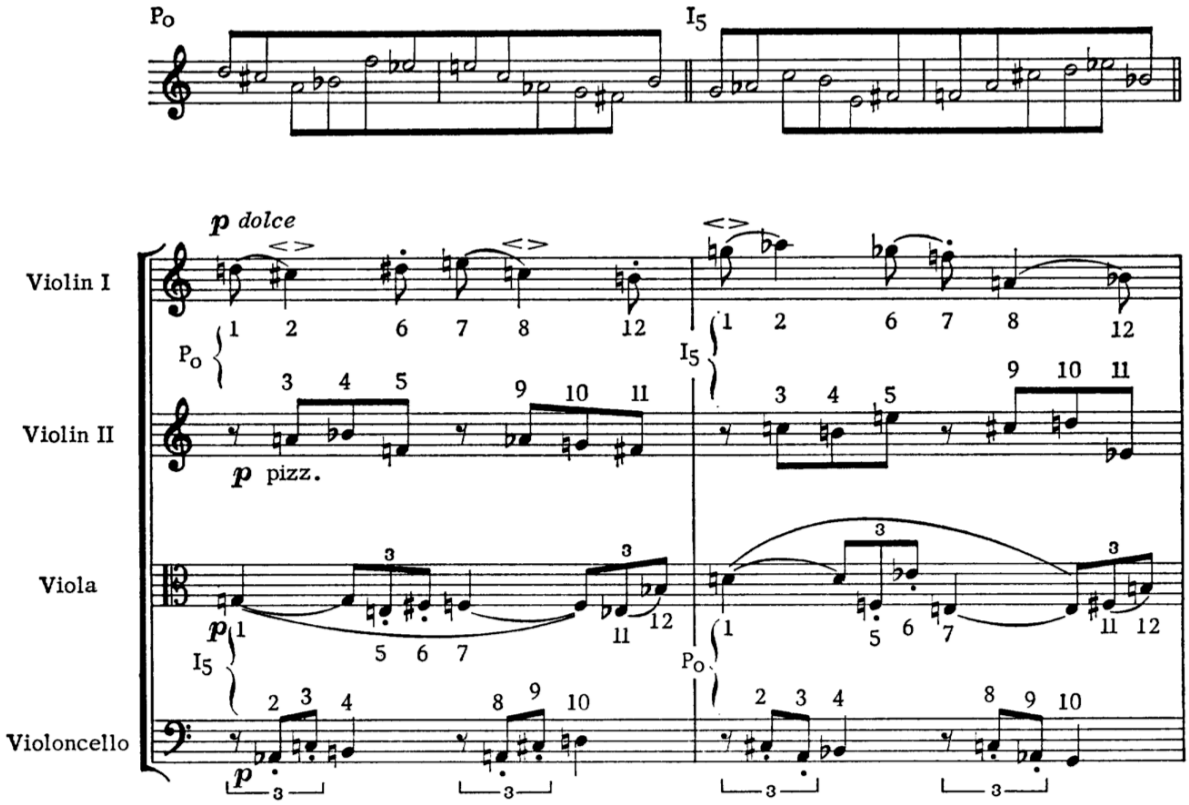


Figure 1-1. Alban Berg's *Lulu* [11, 182].

1.1 Basic Derivation Techniques

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. Fig. 1-2 shows an early example of this technique in Schoenberg's fourth string quartet [12, 100]. In this example, the basic row is stated by the violins in the first bar shown, while combined with its T_5I transform, which is presented by the viola and violoncello. In the second bar, this situation is reversed, and the violins present the T_5I transform, while viola and violoncello state the T_0 transform.



Ex. 11. Schoenberg: Fourth Quartet, First Mvt.

Figure 1-2. Derived row in Schoenberg's fourth string quartet [12, 100].

While separating the row between the two violins, two other rows are constructed, namely the row that is stated by the first violin between the two bars shown, and similarly the row that is stated by the second violin. These two rows are constructed from ordered

segments extracted from the basic row, and so are said to be derived from it. Neither the first violin row, nor the second violin row are twelve-tone transforms of the original row or of eachother. The same procedure of picking a fixed pattern of order numbers to separate the $[T_0 \mid T_5I]$ statement of the row between the two violins is used to split the $[T_5I \mid T_0]$ combined statement between viola and violoncello. While picking a fixed pattern of order numbers between the two lower instruments, however, the resulting linear statements for these instruments do not result in derived rows.

Many structured approaches to derivation exist that guarantee that the extracted segments, when combined, will produce derived rows. These approaches often involve constructing a combination matrix of rows. The rows combined in the matrix are usually twelve-tone transforms of the same basic row. The rows that are derived from the segments of these transforms, on the other hand, may or may not be themselves transforms of the basic row.

Example 1.1.1. *The basic row in Berg's* Lulu *[11, 182] may be used as motivation for a basic derivation procedure. The first step is to create a* 2×24 *array where the first row* S *is followed by* $\mathsf{R}(S)$*, 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, an arbitrary segment is chosen, separated from the the top row, and placed 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* $\mathsf{R}(V)$, *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 Ex. 1.1.1. One way would be to find an operation that makes the chosen segment invariant. In particular, it is easily checked that $S_1 = \{10, 0, 5, 7\} = \mathsf{R}\,\mathsf{T}_5\,\mathsf{I}(S_1)$. Ex. 1.1.1 can then be followed by the combination matrix $[\mathsf{R}\,\mathsf{T}_5\,\mathsf{I}(S) \mid \mathsf{T}_5\,\mathsf{I}(S_1)]$. In this second matrix, the segment S_1 could be preserved, but the row derived from $\mathsf{R}\,\mathsf{T}_5\,\mathsf{I}(S)$ by preserving the segment S_1 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 newly derived row would produce different harmonic regions under the same parsing. This procedure can be very pertinent compositionally, as it 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. A new row would then be derived from V, say Q, and eventually V would be followed by Q. Repeating this procedure *ad libitum* could generate many contrasting harmonic regions. In particular, this type of procedure is seen in Donald Martino's *Notturno* of 1974, a composition that won the Pulitzer Prize in the following year [11, 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.

1.2 Literature Review

One of the earliest references in the academic literature of a procedure that may be construed as a derivation technique, illustrated below in Ex. 1.2.1, is seen in [6], a paper whose main purpose is another, namely to generalize the construction of aggregate formations. The techniques described by Martino are influenced by Babbitt's ideas on combinatoriality [6, 224], and make extensive use of tables. In the interest of generalizing the construction of aggregate formations, Martino provides, in addition to hexachordal, also trichordal, tetrachordal, and even pentachordal combinatoriality tables, as well as somewhat brief discussions on oblique combinations [6, 241], which are further discussed in the context of derivation in [11, 216], and on uneven partitions of a row and their combinatoriality implications [6, 267]. Martino acknowledges the aggregates formed by combinatoriality are not necessarily ordered, rather seeing this characteristic as capable of bringing harmonic diversity [6, 228, 230]. In mentioning the derivation of new harmonic sets, this process is juxtaposed to another, namely the fragmentation of the original series. Both procedures are deemed as essentially the same, since set derivation via aggregate formations can be seen as a fragmentation of the set obtained vertically [6, 230, 231].

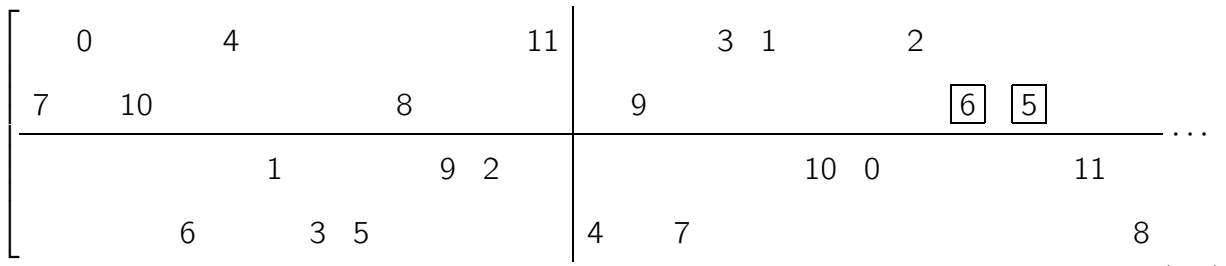
Example 1.2.1. *[6, 231] Let* $S = \{0, 4, 11, 3, 1, 2, 5, 6, 9, 8, 10, 7\}$ *and consider the combination matrix* $\hat{A}$*.*

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

In particular, the hexachords given by the $\hat{A}_i$ *can be combined with their transforms under* T_1I*, yielding the combination matrix* A*.*

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

The row $Q = \{7, 0, 10, 6, 4, 1, 3, 5, 8, 9, 2, 11\}$ *can then be derived from the first column of* A*, but* $T_9(Q)$ *can only partially be derived from the second column of* A*, as pitch-classes 6 and 5 get swapped.*



(1-5)

Even though the term derivation is not itself quite present in it, [12] introduces a mature conception of the problems involved with the technique of derivation. Two basic strategies for achieving polyphony from an essentially linear construct, namely partitioning and combining, are proposed in [12, 95]. Partitioning, which is illustrated in Fig. 1-2, consists of splitting a row into separate lines in a very Schenkerian fashion, with a bias toward partitioning the row in ways that preserve some of its intervallic structure [12, 100]. Combining, on the other hand, departs from combination matrices, similarly to the technique demonstrated in Ex. 1.2.1, and is truly remarkable in the sense that it already represents a technique for self-derivation [12, 101]. Ex. 1.2.2 demonstrates the technique of combining.

Example 1.2.2. *[12, 102] Let* $S = \{2, 1, 9, 10, 5, 3, 4, 0, 8, 7, 6, 11\}$ *be a 12-tone row and consider the* 12×12 *matrix* A*.*

$$A = \begin{bmatrix} 2 & 1 & 9 & 10 & 5 & 3 & 4 & 0 & 8 & 7 & 6 & 11 \\ 1 & 0 & 8 & 9 & 4 & 2 & 3 & 11 & 7 & 6 & 5 & 10 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 3 & 2 & 10 & 11 & 6 & 4 & 5 & 1 & 9 & 8 & 7 & 0 \end{bmatrix}. \tag{1-6}$$

Then $\tilde{A}$ is a 12×144 combination matrix of self-derivation in which every column is a transform of S.

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

The columns of $\tilde{A}$ are obtained by simply arranging the contents of each corresponding column of A. In particular, $\tilde{A} = [T_0(S) \mid T_5 I(S) \mid \cdots$. Observing that 5 precedes 4 in the 10^{th} row of $\tilde{A}$, and similarly that 4 precedes 3 in the 11^{th} row of $\tilde{A}$, many alternative rearrangements of $\tilde{A}$ are possible, such as $\tilde{B}$.

$$\tilde{B} = \left[\begin{array}{c|c|c} \begin{array}{cccccccccccc} 2 & & & & & & & & & & & \\ & 1 & & & & & & & & & & \\ & & \vdots & \vdots & & & & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & & 5 & & \boxed{4} & & & & \\ & & & & & & & & & & & \\ & & & & & & 3 & & & & & \end{array} & \begin{array}{cccccccccccc} & & & & & & & & 1 & & & \\ & & & & 0 & & & & & & & \\ \vdots & \vdots & & \vdots & & & \vdots & \vdots & \vdots & & & \vdots \\ & & & & & & \boxed{4} & & & & 3 & \\ & & & & & & & & & 2 & & \end{array} & \cdots \end{array}\right] . \tag{1-8}$$

Combination matrices such as $A = [T_0 \mid \cdots \mid T_1]^T$ *may be constructed in alternative ways where inverse and retrograde transforms of the row* S *can be present. One possible sequence of transforms is*

$$\{T_0, T_{10}, T_8, T_6, T_4, T_2, T_5I, T_3I, T_1I, T_{11}I, T_9I, T_7I\} \ . \tag{1-9}$$

Another is

$$\{T_0, T_8, T_4, T_{11}I, T_7I, T_3I, RT_9, RT_5, RT_1, RT_{10}I, RT_6I, RT_2I\} \ . \tag{1-10}$$

Self-derivation in 12×144 combination matrices is straightforward, since every column is capable of producing arbitrary transforms of any row. However, self-derivation in combination matrices of four rows or less is deemed more difficult to achieved in [12, 108]. Although a systematic way to understand self-derivation in smaller combination matrices is not proposed in [12], the advancements in the field are extremely substantial and arguably one strong motivating force for the studies in self-derivation that follow in the subsequent decades.

In one of the seminal academic works in the field of twelve-tone theory, [11] utilizes a mostly set-theoretic framework to understand and categorize rows and procedures involved in producing combination matrices of derivation self-derivation. This approach revolves around the idea of looking at sets 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; each column of the matrix A in Ex. 1.2.2 is a free aggregate; a maximally constrained set corresponds to 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 here too contributes to a Schenkerian view of the technique of derivation [11, 183, 184]. Mathematically, the ideas in [11] 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. Bitwise operations can applied 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, 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 [11, 184, 185]. Often, pitch-class relations will contain many redundancies due to transitivity. In order to express these relations as oriented graphs, such redundancies must first be removed, or pruned [11, 186]. This process can be reversed and a pitch-class relation can be extended to the point of its transitive closure [11, 190]. It is also common for a pitch-class relation to be absent of any order constraint involving both $\{x, y\}$, in which case x and y are said to be 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 still remains 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 from some partial order is called its total order class [11, 188]. In a completely analogous manner, a pitch-class

relation can be can verticalized by removing constraints, and again minding that the result is still transitive and symmetric. 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 [11, 192, 193]. The following theorems provide more details on covering, as well as unions and intersections of pitch-class relations. In particular, Th. 1.2.4 plays a crucial role in the construction of combination matrices of self-derivation from aggregate realizations [11, 222]. Th. 1.2.6, which is stated without proof, discusses aspects of applying twelve-tone operations to pitch-class relations.

Theorem 1.2.3. *[11, 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 1.2.4. *[11, 194] Let A and B be partial orders and denote by $\mathsf{Toc}(A)$ and $\mathsf{Toc}(B)$ their respective total order classes. Then*

$$\mathsf{Toc}(A) \cap \mathsf{Toc}(B) = \mathsf{Toc}(\mathsf{Ext}(A \cup B)) \ , \tag{1-11}$$

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

$$\bigcap_{i=0}^{n} \mathsf{Toc}(A_i) = \mathsf{Toc}\left[\mathsf{Ext}\left(\bigcup_{i=0}^{n} A_i\right)\right] \ . \tag{1-12}$$

Theorem 1.2.5. *[11, 194] The intersection of two partial orders is again a partial order.*

Theorem 1.2.6. *[11, 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 $\mathsf{R}(C)$ is the retrograde of C, then $\{a, b\} \in \mathsf{R}(C)$ if and only if $\{b, a\} \in C$.*

ii. *If C is totally ordered, then $\mathsf{R}(C) = (S \setminus D) \cup F$, 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 $F\,\mathsf{R}$-invariant, then all cycles in F have length two.*

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 this total order, and lead to a classification of partial orders, as well as to many musical applications. One interesting compositional application of aggregate realizations is that of projecting a total order as a middle-ground entity. 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 [11, 197]. Whereas aggregate realizations correspond to a totally ordered sequence of disjoint subsets of the free aggregate, a columnar aggregate 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, a columnar aggregate must contain

the free aggregate as a subset, so that all pitch classes belonging to a given base are included in every column. 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 by the above [11, 201, 210]. A row segment is a pitch-class relation that is reflexive, transitive, and antisymmetric, with the additional requirement that some subset of its order constraints also satisfy trichotomy. Any partial order that covers some row segment is said to be an embedded segment of that 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 of its various embedded segments [11, 198]. Row segments may be concatenated in a natural way. Let $X = \{0, 1, 2\}$ and $Y = \{3, 4, 5\}$ be row segments such that $X \cap Y = \emptyset$ when seeing X and Y as partial orders. The concatenation $X|Y$ will then be the partial order $X \cup Y \cup \{\{a, b\} : a \in X, b \in Y\}$. It follows both X and Y are embedded row segments of $X|Y$. The 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, intercalation of segments is not allowed when concatenating. Finding embedded segments that are common to different partial orders is straightforward. Let $X = \{0, 1, 7, 2, 10, 9, 11, 4, 8, 5, 3, 6\}$ and $Y = \{1, 2, 8, 3, 11, 10, 0, 5, 9, 6, 4, 7\}$ be total orders. In particular, $Y = \mathsf{T}_1(X)$. Pruning the intersection $X \cap Y$ 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 then the longest embedded segments of both X and Y [11, 200].

Pruning an intersection of partial orders is also a method for computing common tones between sets that not only produces embedded segments, but can also be applied to the intersection of an arbitrary number of sets. When the sets are related to one another

by a twelve-tone operation, finding common tones between them can be accomplished by expressing the operation as a permutation. Let $S = \{0, 1, 4, 5, 8, 9\}$ and consider the cycle decomposition of T_3I = (0 3) (1 2) (4 11) (5 10) (6 9) (7 8). It follows every pitch-class in S maps to the twelve-tone complement of S under $\mathsf{T}_3\,\mathsf{I}$. If the operation is T_9 = (0 9 6 3) (1 10 7 4) (2 11 8 5), then S shares three common tones with $\mathsf{T}_9(S)$, namely $0 \mapsto 9$, $4 \mapsto 1$, and $8 \mapsto 5$. A simple formula for computing the number of common tones between two sets under some T_nI transform is also given in [9, 10]. Yet another technique for finding common tones between two sets consists of computing a table or matrix. This procedure has the advantage of not only giving all indices of transposition under which a set shares common tones with another, but of making it possible to determine whether these common tones will preserve their ordering after the transformation by examining the table's diagonals. It has the disadvantage of only being able to produce common tones between two sets at a time. 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 a row be compared 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, 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 being counted 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 $\bar{X}$ [8, 49]. This matrix for finding common tones under transposition has the interesting property that it becomes a symmetric matrix when $A = \bar{A}$ and its elements are taken 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 whether its elements are taken as interval classes. Multiplicative operations, however, will often lack multiplicative inverses in twelve tones. For p-TET systems where p is prime, multiplicative operations yield symmetric matrices as well, but those will sometimes require a proper definition of multiplicative interval classes.

The procedure described in Ex. 1.1.1 represents 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 one of the rows is retrograded. The construction is part of a more general procedure, detailed in in [11, 211], in which a row is always matched with a retrograded transform of itself. In the particular case of Ex. 1.1.1, the transform was $F = \mathsf{T}_0$ for simplicity, but arbitrary twelve-tone operations may be used. Denote the given series by S and define 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 that is singled out, remain invariant under F, which will force V_2 to also be invariant under F. In Ex. 1.1.1 this requirement is satisfied trivially. It is usually the case in this type of derivation procedure that the invariance of V under F, or under $\mathsf{R} \circ F$ for that matter, does not preserve any ordering, as illustrated in Ex. 1.2.8. On the other hand, Ex. 1.2.9 illustrates a derivation procedure similar to that in Ex. 1.1.1 where the operation F is not trivial and the singled-out segments are preserved as partial orders. That is accomplished by applying the technique described in [8, 49] for finding common row segments. 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 [11, 211]. More can be said about such combination matrices. Prop. 1.2.7 summarizes the general procedure under set-theoretic terms, and Eq. 1-13 provides the schematic representation of a derivation procedure involving the retrograde and an arbitrary operation F [11, 212].

Proposition 1.2.7. *[11, 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 $C_1 = V_1 \cup \mathsf{R} \circ \mathsf{F}(V_2)$, and similarly the second column is the partial order $C_2 = V_2 \cup \mathsf{R} \circ \mathsf{F}(V_1)$, such that $C_2 = \mathsf{R} \circ F(C_1)$. If D is a partial order that covers C_1, then $\mathsf{R} \circ F(D)$ 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 $\mathsf{R} \circ F(D)$ is in the total order class of C_2. Finally, 2-row derivations of this type exist for arbitrary rows.*

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

(1-13)

Example 1.2.8. *[11, 212] Let $S = \{0, 1, 7, 2, 10, 9, 11, 4, 8, 5, 3, 6\}$ and consider the operation $\mathsf{T}_2\mathsf{I} = (0\ 2)\ (3\ 11)\ (4\ 10)\ (5\ 9)\ (6\ 8)$. Inspecting the cycle decomposition of $\mathsf{T}_2\mathsf{I}$ reveals that the segment $S_1 = \{0, 1, 7, 2\}$ remains invariant under this operation. If a combination matrix involving $\mathsf{R}\,\mathsf{T}_2\mathsf{I}$ is considered, however, then the S_1, seen as a partial order, shall not be preserved, since both (1) and (7) are fixed points. The same would happen to any $\mathsf{T}_k\mathsf{I}$ 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.*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} 0 & . & . & 1 & . & 7 & . & . & . & 2 & . & . & 10 & 9 & . & 11 & 4 & 8 & . & 5 & . & 3 & 6 & . \\ . & 8 & 11 & . & 9 & . & 6 & 10 & 3 & . & 5 & 4 & . & . & 0 & . & . & . & 7 & . & 1 & . & . & 2 \end{array}\right] . \tag{1-14}$$

Example 1.2.9. *Let $S = \{10, 2, 3, 0, 5, 7, 4, 6, 9, 8, 1, 11\}$, the row in Berg's* Lulu*, and consider the segment $\vec{s} = [10\ 0\ 5\ 7]^T$ as a column vector. Now let $A^T = [\ \vec{s} \mid \vec{s} \mid \vec{s} \mid \vec{s}\ ]$ be*

the square matrix whose every column is equal to $\vec{s}$*. Then*

$$A + A^T = \begin{bmatrix} 2 & 10 & 3 & 5 \\ 10 & 0 & 5 & 7 \\ 3 & 5 & 10 & 0 \\ 5 & 7 & 0 & 2 \end{bmatrix} \pmod{12} \ . \tag{1-15}$$

In particular, the row segment $\{10, 0, 5, 7\}$ *is invariant under* $\mathsf{RT}_5\mathsf{I}$*, since* $A + A^T$ *displays an anti-diagonal of fives. Thus, by matching* S *with* $\mathsf{RT}_5\mathsf{I}(S)$*, the row segment* $\{10, 0, 5, 7\}$ *may be obtained 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* $V = \{2, 3, 4, 6, 9, 8, 1, 11, 10, 0, 5, 7\}$*.*

$$\left[\begin{array}{cccccccccccc|cccccccccccc} . & 2 & 3 & . & . & . & 4 & 6 & 9 & 8 & 1 & 11 & . & . & . & . & . & . & 10 & 0 & 5 & . & . & 7 \\ 10 & . & . & 0 & 5 & 7 & . & . & . & . & . & . & 6 & 4 & 9 & 8 & 11 & 1 & . & . & . & 2 & 3 & . \end{array}\right] . \tag{1-16}$$

A distinction is made in [11, 211, 214] between derivation and polyphonyzation. The former departs from a combination matrix of rows to infer a concatenation of rows that constitute the linearized result of flattening the combination matrix. The latter is the opposite concept, that is, given a concatenation of rows, a combination matrix is constructed such that each column breaks down the row at the top into disjoint ordered segments. Although differentiating between derivation and polyphonization is an important concept in the set-theoretic view of the problem, this paper shall not make that distinction. The term derivation will be used indiscriminately to both the breaking down of rows into contrapuntal combination matrices, and the flattening thereof.

A related technique involving greater than 2-row counterpoint is achieved by folding a combination matrix. The process is similar to first constructing a matrix of hexachordal combinatoriality in the traditional sense, 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 $V_1 \cup \mathsf{R} \circ \mathsf{F}(V_2)$. Then putting S in counterpoint with $G(S)$, as well as deriving V from $S|\,\mathsf{R} \circ F(S)$, yields the schematic representation in Eq. 1-17. It is argued without proof that vertically satisfying G while horizontally satisfying F implies F and G must commute [11, 215]. Ex. **??** shows an application of folded derivation.

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

(1-17)

Example 1.2.10. *[11, 215] Let* $S = \{0, 1, 11, 3, 8, 10, 4, 9, 7, 6, 2, 5\}$ *and consider the cycle decomposition of* $\mathsf{T}_6 = (0\ 6)\ (1\ 7)\ (2\ 8)\ (3\ 9)\ (4\ 10)\ (5\ 11)$. *Let* $V_1 = \{1, 3, 8, 9, 7, 2\}$. *Since the unordered set* V_1 *is* T_6*-invariant, it follows* $V_2 = \{11, 0, 10, 4, 5, 6\}$ *and the following combination matrix is obtained:*

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

Consider further the cycle decomposition of $\mathsf{T}_5\mathsf{I} = (0\ 5)\ (1\ 4)\ (2\ 3)\ (6\ 11)\ (7\ 10)\ (8\ 9)$. *Then* $S_1 = \{0, 1, 11, 3, 8, 10\}$ *maps to its complement under* $\mathsf{T}_5\mathsf{I}$*, so* S *can be used in a combination matrix where* S *is matched with its transform under* $\mathsf{T}_5\mathsf{I}$ *in the usual way, that is, in a matrix of hexachordal combinatoriality. Moreover, the row* $\mathsf{T}_5\mathsf{I}(V)$ *can be derived from* $\mathsf{T}_5\mathsf{I}(S)$. *Since* $\mathsf{T}_5\mathsf{I}$ *commutes with* T_6*, as any pitch-class operation does, the*

following folded derivation is possible:

$$\left[\begin{array}{cccccccccccc|cccccccccccc}
 & 1 & & 3 & 8 & & & 9 & 7 & & & 2 & 11 & & 0 & & & 10 & 4 & & & 5 & & 6 \\
0 & & 11 & & & 10 & 4 & & & 6 & & 5 & & 8 & & 1 & 3 & & & 2 & 9 & & 7 & \\
\hline
5 & & 6 & & & 7 & 1 & & & 11 & & 0 & & 9 & & 4 & 2 & & & 3 & 8 & & 10 & \\
 & 4 & & 2 & 9 & & & 8 & 10 & & & 3 & 6 & & 5 & & & 7 & 1 & & & 0 & & 11
\end{array}\right] . \tag{1-19}$$

An analogue to the technique of oblique combination described in [6, 241, 267] is given in [11] in the context of folded derivation, and denoted skewed polyphonization. The procedure consists of shifting horizontally one of the two rows of the derivation matrix. Skewed polyphonization may be equivalent to folding combination matrices that involve the retrograde, and there is no requirement that the combinatoriality be hexachordal. Eq. 1-20 provides a schematics description of a skewed polyphonization procedure involving the retrograde. The asterisk indicates there is no requirement that the same row class be derived in both foldings, in which case V would be replaced by some other row V^* and G could be the identity. An interesting compositional application consists of matching altogether different foldings of a row, thus effectively bridging different derivations of the same generative material. Ex. 1.2.11 shows an application of skewed polyphonization where combinatoriality is not hexachordal, but both foldings derive different transforms of the same row [11, 216].

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

(1-20)

Example 1.2.11. *[11, 216] Consider the row* $S = \{0, 1, 7, 2, 10, 9, 11, 4, 8, 5, 3, 6\}$ *and the combination matrix given by* $\mathsf{R}\,\mathsf{T}_{10}\,\mathsf{I}(S)$ *against* $\mathsf{T}_{11}(S)$*:*

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

Deriving the row $V = V_1|V_2 = \{1, 7, 2, 9, 8, 3\}|\{4, 5, 6, 11, 0, 10\}$ *from* $S|\,\mathsf{R}\,\mathsf{T}_{10}\,\mathsf{I}(S)$*, and following the scheme given in Eq. 1-20, yields the following shifted derivation:*

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

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

Self-derivation may be seen both from the set-theoretic perspective, and from the standpoint of aggregate realizations is discussed in [11, 217, 226]. The only difference between self-derivation and the general case is that in combination matrices of self-derivation all row forms belong to the same row class, that is, all rows are RT_nI-transforms of the original row. The same applies to folded and skewed combination matrices of self-derivation. The general scheme for a two-row combination matrix of self-derivation involving the retrograde is given in Eq. 1-23 [11, 219]. It is stated without proof in [11, 217] that the operations F and G in Eq. 1-23 must commute. For this type of self-derivation on two-row counterpoint, however, simple inspection of the examples that follow such statement

already affords a counter-proof, as described in Ex. 1.2.12. Similarly to the general derivation case, two-row self-derivations can also 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. 1.2.13. The folded rows can subsequently be folded, and this procedure can generate many levels of self-derivation [11, 221]. Ex. 1.2.14 provides a musical application of folded self-derivation matrices that constitute the main compositional procedure in Ciro Scotto's *Tetralogy*.

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

(1-23)

Example 1.2.12. *[11, 218] Let* $S = S_1|S_2 = \{3,8,1,0,9,6\}|\{4,7,10,5,2,11\}$ *and consider the cycle decomposition of* $\mathsf{T}_9\mathsf{I} = (0\ 9)\ (1\ 8)\ (2\ 7)\ (3\ 6)\ (4\ 5)\ (10\ 11)$*. In particular, both* S_1 *and* S_2 *are invariant under* $\mathsf{T}_9\mathsf{I}$*. It is not obvious, however, that* S *and* $\mathsf{R}\,\mathsf{T}_9\mathsf{I}(S)$ *can be derived from a combination matrix* A *where the first column is* $\mathsf{T}_7(S)$ *and the second is* $\mathsf{R}\,\mathsf{T}_2\mathsf{I}(S) = \mathsf{R}\,\mathsf{T}_9\mathsf{I}\circ\mathsf{T}_7(S)$*. Here* $F = \mathsf{T}_9\mathsf{I}$ *and* $G = \mathsf{T}_7$*. It is not the case that* F *and* G *commute, as* $\mathsf{T}_2\mathsf{I} = F\circ G \neq G\circ F = \mathsf{T}_4\mathsf{I}$*.*

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

Example 1.2.13. *[11, 221] Let* $S = \{0,11,5,10,4,2,7,9,8,3,6,1\}$ *and consider the following combination matrix given by* $T_2(S)|\,\mathsf{R}\,\mathsf{T}_2(S)$*, whose derived rows are* S *and* $\mathsf{R}(S)$*:*

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

Subjecting the entire matrix to T_1 yields:

$$\left[\begin{array}{cccccccc|cccccccc} & 1 & & 0 & 6 & & 11 & & 5 & 3 & & 8 & & 10 & & 9 & 4 & & 7 & 2 \\ 2 & & 7 & & & 4 & 9 & & 10 & & 8 & & 3 & & 5 & & & 11 & & \\ & & & & & & & & & & & & & & & & 6 & 0 & 1 & \end{array}\right]. \tag{1-26}$$

The entire matrix in Eq. 1-25 can then be pulled from the first row of Eq. 1-26:

$$\left[\begin{array}{cccccccc|cccccccc|c} & & & 0 & & & 11 & & 5 & & & & 10 & & 4 & & & & 2 & \\ & 1 & & & 6 & & & & & & 3 & 8 & & & & 9 & & 7 & & 2 \cdots \\ 2 & & 7 & & & 4 & 9 & & 10 & & 8 & 3 & & 5 & & 11 & & 6 & 0 & 1 & \end{array}\right]. \tag{1-27}$$

Example 1.2.14. *[10, 180] Let $S = \{0, 4, 7, 3, 11, 2, 10, 1, 6, 8, 9, 5\}$ and consider the self-derivation array in Fig. 1-3.*

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 1-3. Self-derived row in Scotto's *Tetralogy*.

Similarly to Ex. 1.2.13, the above matrix can be folded in order to obtain subsequent levels of derivation. Unlike Ex. 1.2.13, however, Scotto derives a rotated transform of S from $\mathsf{R}\,\mathsf{T}_7(S)$. The rows in Fig. 1-4 are thus $\mathsf{T}_2(S), \mathsf{R}\,\mathsf{T}_2(S), \rho_6\,\mathsf{R}\,\mathsf{T}_2(S)$ and $\rho_6\,\mathsf{T}_2(S)$, where ρ is the cyclical rotation operator.

[037]	[014]	[037]	[014]	[014]	[037]	[014]	[037]	[037]	[014]	[037]	[014]	[014]	[037]	[014	[037
	2	6	9	5	1		4	0	3		8		t	e	7
7	e	t		8		3	0	4		1	5	9	6	2	
4		1	5	9	6	2		7	e	t		8		3	0
0	3		8		t	e	7		2	6	9	5	1		4
T_0				T_0R				T_0				T_0R			

Figure 1-4. 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* $\mathsf{T}_2(S)$ *now corresponds to the alto register, as seen in Fig. 1-5. One of the foreground realizations of the Schenkerian graph in Fig. 1-5 is given in Fig. 1-6. Rather than a strict serial composition,* Tetralogy *employs a variety of prolongation procedures which are in line with its Schenkerian orientation.*

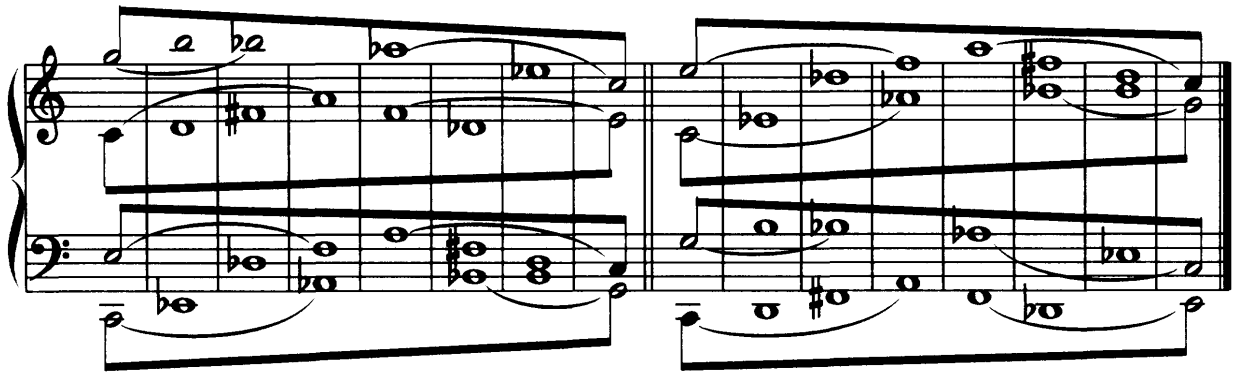


Figure 1-5. Schenkerian middle-ground structure in Scotto's *Tetralogy.*

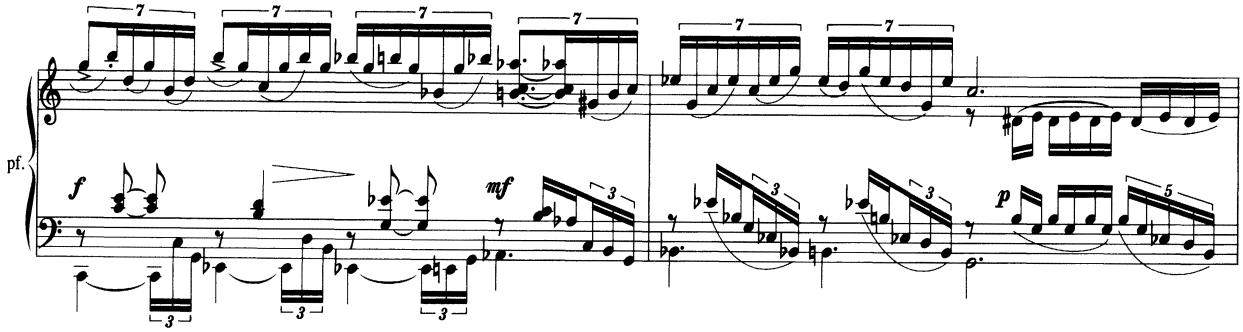


Figure 1-6. Musical realization of the middle-ground in Scotto's *Tetralogy.*

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 background of the piece. The prolongation of the first bar in Fig. 1-5 is depicted in Fig. 1-7, and a musical realization thereof is displayed in Fig. 1-8.*

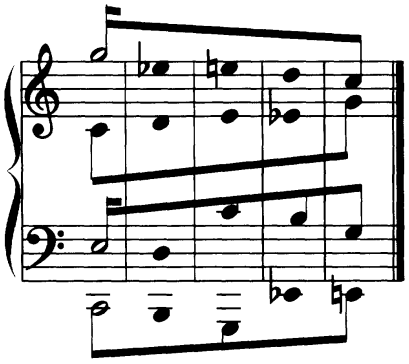


Figure 1-7. Prolongation of the middle-ground structure in Scotto's *Tetralogy*.

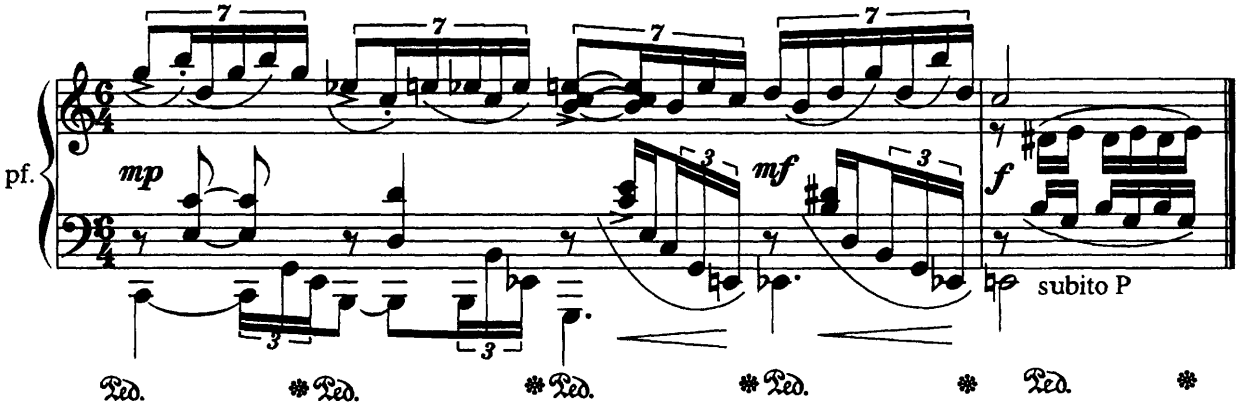


Figure 1-8. Musical realization of the prolonged middle-ground in Scotto's *Tetralogy*.

Three algorithms for computing combination matrices of self-derivation are presented in [11]. The first is an algorithm for computing 2×24 matrices that consists of finding a segment at the beginning of an ordered set that can become an embedded segment in some transform of the set, then constructing a combination matrix involving the retrograde by manually resolving any symmetries. There is no prior knowledge of the entire row, thus the row is actually composed by adding order constraints to an initially unconstrained set [11, 217]. The second algorithm utilizes Th. 1.2.4 to construct self-derivation from general combinatoriality matrices by finding operations that resolve symmetries between the columns of the matrix. Although there is no requirement that the matrix be retrograde-invariant, the retrograde makes it arguably easier to work out the algorithm by hand. Ex. 1.2.15 illustrates the second algorithm in which, similarly

to the first, the entire row is discovered as a result of the procedure [11, 222]. The third algorithm proposed makes use of columnar aggregates to compose combination matrices of self-derivation, and also utilizes Th. 1.2.4. Ex. 1.2.16 illustrates how this third algorithm works by explicitly building the columnar aggregates in the matrix from the cycles of some operation [11, 226, 227], and Ex. 1.2.17 shows how a 6×72 matrix of self-derivation may be used in a musical composition. This matrix was constructed by exploring the symmetries of an aggregate realization, that is, by using the third algorithm described in [11].

Example 1.2.15. *[11, 224] Let* $S = \{0, 1, 7, 2\}|\{10, 9\}|\{11, 4, 8, 5\}|\{3, 6\}$. *Given that the complement of a row's first hexachord is always obtained from its second hexachord, a trivial case of hexachordal combinatoriality results when a row is matched with its retrograde. It is then possible to partition* S *such that the combination matrix will have four columns:*

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

Now consider the cycles of $\mathsf{T}_{11}\mathsf{I} = (0\ 11)\ (1\ 10)\ (2\ 9)\ (3\ 8)\ (4\ 7)\ (5\ 6)$. *In particular, the first column of* A *will comprise the partial order* $S_1 \cup \mathsf{R}(S_4)$, *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, hexachordal combinatoriality under* $\mathsf{T}_{11}\mathsf{I}$ *is obtained 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. The same result is not obtained 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 yields the matrix $\hat{A}$.

$$\hat{A} = \left[\begin{array}{c|c|c|c} \{0,1,7,2\} & \{10,9\} & \{11,4,8,5\} & \{3,6\} \\ \{6,3\} & \{5,8,4,11\} & \{9,10\} & \{2,7,1,0\} \\ \{11,10,4,9\} & \{1,2\} & \{0,7,3,6\} & \{8,5\} \\ \{5,8\} & \{6,3,7,0\} & \{2,1\} & \{9,4,10,11\} \end{array}\right] . \tag{1-29}$$

The above can be construed as a sequence of four columnar aggregates. It may be possible to derive members of the same row class from each column. If the total order class of the intersection of all columns is not empty, that is, if it contains the free aggregate and does not contain any symmetry, then Th. 1.2.4 guarantees that representatives of this row class can be derived 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 $T \in \mathsf{Toc}\{\mathsf{Ext}[C_1 \cup \mathsf{R}\,\mathsf{T}_1\,\mathsf{I}(C_2) \cup \mathsf{T}_1\,\mathsf{I}(C_3) \cup \mathsf{R}(C_4)]\}$. Therefore $[V|\,\mathsf{R}\,\mathsf{T}_1\,\mathsf{I}(V)|\,\mathsf{T}_1\,\mathsf{I}(V)|\,\mathsf{R}(V)]$ can be derived from the columns of $\hat{A}$.

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

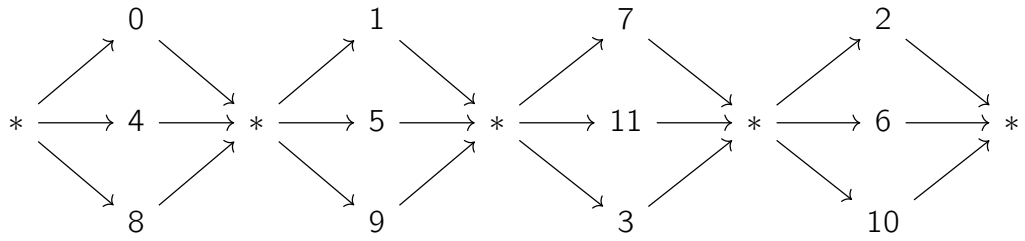
$$\cdots \left[\begin{array}{c|c|c|c} \begin{array}{cccccccc} & & & & & & & \\ & & & & 11 & 4 & 8 & 5 \\ & & & 9 & & & & 10 \\ \hline & & 0 & 7 & 3 & & 6 & \\ 2 & 2 & 1 & & & & & \end{array} & \begin{array}{cccccccc} & & & & & & & \\ 3 & & & & & & & 6 \\ & & & & & 2 & 7 & \\ \hline 8 & & 5 & & & & &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

(1-30)

Example 1.2.16. *[11, 226, 227] Let* $S = S_1|S_2|S_3 = \{0,1,7,2\}|\{10,9,11,4\}|\{8,5,3,6\}$ *and consider the matrix* $A = [S|\,\mathsf{T}_4(S)|\,\mathsf{T}_8(S)]^T$.

$$A = \left[\begin{array}{cccc|cccc|cccc} 0 & 1 & 7 & 2 & 10 & 9 & 11 & 4 & 8 & 5 & 3 & 6 \\ 4 & 5 & 11 & 6 & 2 & 1 & 3 & 8 & 0 & 9 & 7 & 10 \\ 8 & 9 & 3 & 10 & 6 & 5 & 7 & 0 & 4 & 1 & 11 & 2 \end{array}\right] . \tag{1-31}$$

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.

$A_1 =$ 

(1-32)

When the first column of A *is regarded as an aggregate realization, it becomes clearer that any* $V \in \mathsf{Toc}(A_1)$ *must be a succession of augmented triads. One possibility would be* $V = \{4,8,0,9,1,5,11,7,3,2,10,6\}$. *It is easy to see that linearizing* V *from* A_1 *is possible, but it is not obvious whether some transform of* V *can be linearized 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*

$$S_1 \pmod 4 = \{0,1,3,2\} = \mathsf{T}_2\,\mathsf{I}(\{2,1,3,0\}) = \mathsf{T}_2\,\mathsf{I} \circ S_2 \pmod 4 \ . \tag{1-33}$$

The above method works because the elements in each of A_1*'s columns are incomparable, thus may be picked from each column in any order. In other words,* A_1 *could be flipped horizontally, say, and still have the same aggregate realization. Since all of* A*'s columns can be reduced* $\mod 4$ *by construction, it becomes enough to only consider each column's residue modulo four, and four-tone operations. Since* $S_1 \pmod 4 = S_3 \pmod 4$, V *itself*

can be derived from A_3, yielding the following derivation matrix, where the second column is $\mathsf{T}_2\mathsf{I}(V)$:

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

$$\left[\begin{array}{cccccccccc|cccccc}
 & & & & & & 11 & & 4 & & & 8 & & & 5 & & 3 & & 6 \\
 & \cdots & 3 & & & & & & & & 8 & & 0 & 9 & & & 7 & & 10 \\
 & & & & 7 & & 0 & & & & & 4 & & & 1 & 11 & & 2 &
\end{array}\right] . \quad \text{(1-34)}$$

It may be possible to derive other rows from A that are not based on the concatenation of cycles from an operation. Knowing that the columns of A are related as aggregate realizations by the operation tuple $\mathcal{A} = [\mathsf{T}_0 \ \ \mathsf{T}_2\mathsf{I} \ \ \mathsf{T}_0]$, and regarding the A_i as columnar aggregates, take $\hat{A} = \mathsf{Ext}[\bigcup_i(\mathcal{A}_i \circ A_i)]$. By Th. 1.2.4, any row that can be linearized from $\hat{A}$, will be in $\bigcap_i \mathsf{Toc}(A_i)$, and thus its $\mathcal{A}_i$-transform can be derived from each i-column of A. Below is the columnar aggregate $\hat{A}$.

$$\hat{A} = \quad \begin{array}{ccccccccccc}
 & & 0 & \longrightarrow & 1 & \longrightarrow & 7 & \longrightarrow & 2 & & \\
 & \nearrow & & \searrow\nearrow & & \searrow\nearrow & & \searrow\nearrow & & \searrow & \\
* & \longrightarrow & 4 & \longrightarrow & 5 & \longrightarrow & 11 & \longrightarrow & 6 & \longrightarrow & * \\
 & \searrow & & \nearrow\searrow & & \nearrow\searrow & & \nearrow\searrow & & \nearrow & \\
 & & 8 & \longrightarrow & 9 & \longrightarrow & 3 & \longrightarrow & 10 & &
\end{array} \quad \text{(1-35)}$$

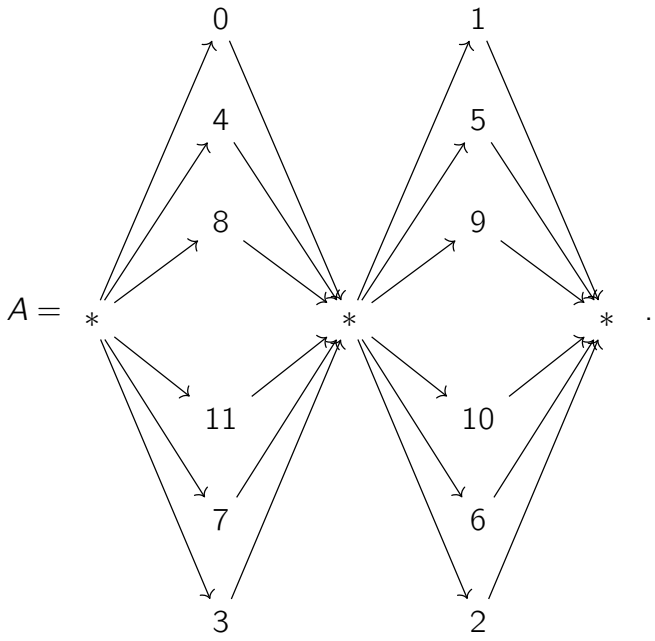
It follows $\hat{V} = \{0, 1, 4, 8, 9, 5, 7, 2, 11, 3, 10, 6\}$ *can be linearized from* $\hat{A}$*, yielding the derivation matrix below:*

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

$$\left[\begin{array}{cccccccccc|cccccc} & & 9 & & & 11 & & 4 & & & 8 & & & 5 & & 3 & 6 & \\ \cdots & & & 3 & & & 8 & & & & 0 & 9 & & 7 & & & & 10 \\ & & & & 0 & & & & & & & & 4 & 1 & & 11 & 2 & \end{array}\right] . \quad (1\text{-}36)$$

Example 1.2.17. *Consider the aggregate realization* A*.*

$A =$



. (1-37)

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* $\rho \in \mathsf{Toc}(A)$*, then also* $\Omega_i(\rho) \in \mathsf{Toc}(A)$*. 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* $\mathcal{A} = [\mathcal{A}_1 | \cdots | \mathcal{A}_6]$.

$$\mathcal{A} = \left[\begin{array}{cc|cc|cc|cc|cc|cc} 0 & 1 & 5 & 8 & 9 & 4 & 10 & 3 & 7 & 6 & 2 & 11 \\ 4 & 5 & 9 & 0 & 1 & 8 & 2 & 7 & 11 & 10 & 6 & 3 \\ 8 & 9 & 1 & 4 & 5 & 0 & 6 & 11 & 3 & 2 & 10 & 7 \\ 11 & 10 & 6 & 3 & 2 & 7 & 1 & 8 & 4 & 5 & 9 & 0 \\ 7 & 6 & 2 & 11 & 10 & 3 & 9 & 4 & 0 & 1 & 5 & 8 \\ 3 & 2 & 10 & 7 & 6 & 11 & 5 & 0 & 8 & 9 & 1 & 4 \end{array}\right] . \tag{1-38}$$

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 a transform of* S *can be derived from every* $\mathcal{A}_i$. *Since* $\mathsf{T}_7(S) \in \mathsf{Toc}(\mathcal{A}_1)$ *and* $\mathsf{R}\,\mathsf{T}_0\,\mathsf{I}(S) \in \mathsf{Toc}(\mathcal{A}_2)$, *the self-derivation matrix* X *is obtained.*

$$X = \left[\begin{array}{cccccc|cccccc} & & & 11 & & 10 & & & & & & \\ & & & & 4 & & 5 & & & & & \\ & & 3 & & & & & 2 & & & & \\ & 0 & & & & & & & 1 & & & \\ 8 & & & & & & & & & 9 & & \\ 7 & & & & & & & & & & & 6 \end{array}\right| \left.\begin{array}{cccccc} & & 6 & & & 3 \\ & & & & 9 & & & 0 \\ & 10 & & & & & 7 & \\ & & & 5 & & 8 & & \\ 1 & & & & & & 4 & \\ & & & & 2 & & & 11 \end{array}\right] . \tag{1-39}$$

It is important to point out that, although the matrix X *could be extended to a* 6×72 *derivation matrix wherein all columns of* $\mathcal{A}$ *are presented, this could not be done using arbitrary transforms of* S. *Upon inspection, it follows that the transforms of* S *that can be derived from* $\mathcal{A}_1$ *and* $\mathcal{A}_5$ *are in*

$$\{\mathsf{T}_3, \mathsf{T}_7, \mathsf{T}_{11}, \mathsf{R}\,\mathsf{T}_1, \mathsf{R}\,\mathsf{T}_5, \mathsf{R}\,\mathsf{T}_9, \mathsf{T}_0\,\mathsf{I}, \mathsf{T}_4\,\mathsf{I}, \mathsf{T}_8\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_2\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_6\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_{10}\,\mathsf{I}\} . \tag{1-40}$$

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

$$\{\mathsf{T}_1, \mathsf{T}_5, \mathsf{T}_9, \mathsf{R}\,\mathsf{T}_3, \mathsf{R}\,\mathsf{T}_7, \mathsf{R}\,\mathsf{T}_{11}, \mathsf{T}_2\,\mathsf{I}, \mathsf{T}_6\,\mathsf{I}, \mathsf{T}_{10}\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_0\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_4\,\mathsf{I}, \mathsf{R}\,\mathsf{T}_8\,\mathsf{I}\} \ . \tag{1-41}$$

Rather than a hinderance, this fact can be leveraged to explore contrasting harmonic regions. Fig. 1-9 shows a musical realization of X:

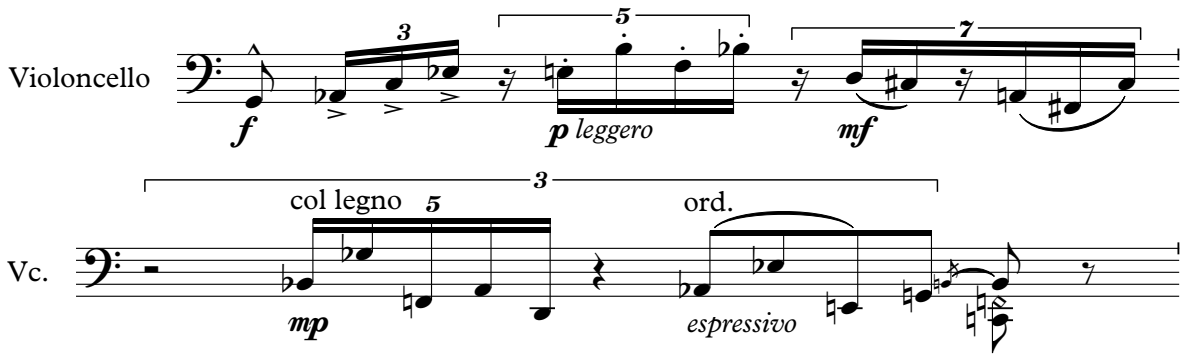


Figure 1-9. Self-derivation in Damiani's *Stingray*.

1.3 Final Remarks

The main objective of the present paper is to understand the construction of combination matrices of self-derivation from an algorithmic standpoint. The techniques proposed in [11] for self-derivation depart from row segments, aggregate realizations or columnar aggregates to construct a total order, but the row as a whole is not known before hand. The algorithm proposed in the next chapter will approach the problem of self-derivation from previous knowledge of row. In other words, given a row and a combination matrix size, what are all the possible combination matrices of self-derivation that can be constructed. A fundamental aspect of this algorithm is that it is conceived in all generality, that is, there are no restrictions to row or to problem sizes, hence it will be possible to compute combination matrices for arbitrary n-tone equal temperament systems, if the intention is to use the results to organize pitch classes. However, this generality also implies that self-derivation be used for musical dimensions other than pitch, for which the 12-TET paradigm might not apply. Another important concern is the reduction of the body of

solutions into representatives of equivalence classes. Reducing solutions is fundamental for improving execution time and space requirements. The last chapter of the present work will present musical applications of self-derivation as obtained by the proposed model and algorithm. The main relevance of this research is that it extends the field of self-derivation with a novel algorithm based on semi-magic squares. This procedure is simple enough to compute solutions by hand, and powerful enough to compute thousands of solutions in a few milliseconds using a computer. It also completely general with regard to row and matrix sizes. Moreover, it complements the existing techniques and algorithms in the literature by allowing the composer to use a pre-existing row as input, rather than to have a row composed as a byproduct of the procedure. An important contribution is the understanding of how the group RT_nI acts on the set of solutions produced by the algorithm, inducing orbits of different sizes depending on whether the input row is retrograde-invariant or not.

Despite the many unique contributions, there are certain areas within the field of self-derivation that will not be studied in the present work. In particular, folded and skewed combination matrices are not computed by the algorithm. Folding as described in Ex. 1.2.13 can be construed as an equivalence relation in the sense that combination matrices of different sizes may form an orbit of contractions and expansions. Understanding how solutions of different sizes relate is a fascinating subject for further research, but unfortunately not discussed here. Another important limitation is the fact that only the action of RT_nI is studied, hence multiplication and the cyclic rotation operator are not considered. The former operation is simpler than the latter if the present algorithm were to be improved. However, the latter is arguably more relevant. Fig. 1-4 presents an important example of how cyclic rotation may appear when folding combination matrices of self-derivation. Multiplication, on the other hand, is controversial for not preserving intervallic content. Magic squares in themselves have amazing compositional applications. One interesting example is the music of Zack Browning. This paper will discuss

semi-magic squares only in conjunction with self-derivation. Even within the realm of self-derivation, however, semi-magic squares present a potential for musical syntax, as one can go from one square to another via matrix operations. This potential is nevertheless left for future research. Derivation is often approached from the more general context of multiple order functions. These include not only the derivation techniques described above, but also row segments that can belong to more than one dimension of the musical discourse at a time. A prototypical example are the Mallalieu-type rows. Multiple order functions, including Mallalieu rows, are well-understood, and the latter do not necessarily yield interesting combination matrices of self-derivation which are the main subject of this research. Therefore, self-derivation shall not be addressed here from the perspective of multiple order functions. Having an efficient algorithm to compute self-deriving matrices is an important component in the algorithmic composition of electroacoustic music. Devising algorithms for both real-time and non-real-time generation of electroacoustic music, where self-derivation is applied to arbitrary parameters of a musical composition, is also not discussed in this paper, but represents a fundamental direction for future research.

The ideas presented throughout the theoretical framework in the next chapter assume a wealth of prior knowledge. A set-theoretic view of 20^{th}-century musical analysis, as taught in graduate-level courses, that is, familiarity with such basic concepts as contour, pitch, and pitch-class spaces, intervals and interval classes, set and row classes, twelve-tone operations, and associated group-theoretic notions is assumed. Likewise, solid knowledge of 12-tone theory, including the hexachordal combinatoriality seen in the works of Schoenberg, and the trichordal combinatoriality seen in the works of Babbitt, as well as the methods that afford constructing such combinatorialities will not be discussed, but assumed. Also necessary is a working knowledge of abstract and linear algebra, to the extend that many graduate texts in atonal music theory will require. C and C++ programming, and basic concepts of computer science are not included, but only a very superficial knowledge of computer science is necessary to understand the theoretical

framework. More involved correctness proofs and complexity analyses of the algorithms presented in the next chapter will be omitted in order to maintain the focus on the musical relevance of such constructs.

CHAPTER 2
THEORETICAL FRAMEWORK

2.1 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.

2.1.1 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 2.1, 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 2.1. 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 2.1 as a subroutine, and is outlined in Listing 2.2. The sole parameter to Listing 2.2 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 2.2. 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 / 2; ++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 2.1 defined above. As i increases, front traverses half of 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 2.5, which in turn depends on two small subroutines, namely Listing 2.3 and Listing 2.4. 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 2.3. 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 2.4 is a simple convenience method that is used to check whether $x \equiv X \pmod{n}$, which is also needed in Listing 2.5.

Listing 2.4. 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 2.4.

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 2.5 is to have previously called Listing 2.2, the result thereof being used as one of the inputs to Listing 2.5.

Listing 2.5. 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 2.8, which in turn does transform the input row into its canonical form. This transformation happens in Listing 2.7, and the latter relies on a straightforward method to in-place swap row entries, described below in Listing 2.6.

Listing 2.6. 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 2.6 is very common-place, so the details are omitted. Listing 2.7 is also straightforward and represents the main subroutine in Listing 2.8.

Listing 2.7. 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 2.6.
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 2.7 is used below in Listing 2.8, 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 2.8 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 2.8.

Listing 2.8. 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 2.2 and Listing 2.5 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 2.7. The third argument is the additive inverse of the row's start element. It varies, naturally, if the row is inverted.

2.1.2 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 2.11. It depends on two subroutines, namely Listing 2.9 and Listing 2.10 that are described below.

Lemma 2.1.1. *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 2.1.2. *A row with odd size greater than* 3 *cannot be retrograde-invariant.*

Proof. □

Listing 2.9. Determining whether a row is retrograde-invariant.

```
bool isRetrogradeInvariant(const number *row, number rowSize) {
    if (rowSize > 3 rowSize % 2 == 1)
        return false;

    number half = rowSize / 2;

    for (number i = 0; i < half; ++i) {
        if (modulo(row[rowSize - i - 1] - row[i], rowSize) != half)
            return false;
    }
```

```
                                                                    11
    return true;                                                    12
}                                                                   13
```

1. The only inputs are a row and its size.
2. Line 2 utilizes Th. 2.1.2 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. 2.1.1. 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 2.10 outlines the procedure in all generality.

Lemma 2.1.3. *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 2.10. 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 2.9, 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 2.11. 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 2.7 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 2.11.

Listing 2.11. Computing and storing a row class in memory.

```
number *getRowClass(range *classSize, number *row, number rowSize, bool      1
    isInvariant) {
    getCanonicalForm(row, rowSize, isInvariant);                           2
    *classSize = isInvariant ? rowSize * 2 : rowSize * 4;                  3
                                                                           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];
        memcpy(rep, rowClass, rowSize * sizeof(number));
        getTransformInPlace(rep, rowSize, i, true, false);
    }

    if (*classSize == rowSize * 2)
        return rowClass;

    for (number i = rowSize * 2; i < rowSize * 3; ++i) {
        number *rep = &rowClass[i * rowSize];
        memcpy(rep, rowClass, rowSize * sizeof(number));
        getTransformInPlace(rep, rowSize, i, false, true);
    }

    for (number i = rowSize * 3; i < rowSize * 4; ++i) {
        number *rep = &rowClass[i * rowSize];
        memcpy(rep, rowClass, rowSize * sizeof(number));
        getTransformInPlace(rep, rowSize, i, true, true);
    }

    return rowClass;
}
```

1. The inputs are a pointer to the size of the row class, which will be computed by Listing 2.11 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 2.11, utilizing the aforementioned procedures Listing 2.9 and Listing 2.10. 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 2.7 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_nI transforms of the row.

35. Line 36 then simply returns the memory location allocated in line 5.

2.2 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.

2.2.1 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 2.14 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 2.14, a couple subroutines for reading and writing files are outlined.

Listing 2.12. 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 2.12 is a completely common-place procedure in the C-language, and its details are omitted. It is used as a subroutine in Listing 2.13, which is described next.

Listing 2.13. 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 2.12 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 2.15 effectively writes all integer partitions of n with a certain size to a file. It depends on Listing 2.14 to compute all said partitions recursively, which is described next. One important feature of Listing 2.14 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 2.14. Recursively computing all partitions of a number n and permutations thereof with a certain size.

```
void allPartitionsRecursive(number *tmp, number currentSize, number                1
    problemSize, number currentSum, number rowSize, FILE *file) {
    if (currentSize == problemSize - 1) {                                          2
        if (currentSum > 0) { // skip the [0, ..., 0, rowSize] partition           3
            tmp[currentSize] = rowSize - currentSum;                               4
            fwrite(tmp, sizeof(number), problemSize, file);                        5
        }                                                                          6
```

```
                                                                    7
        return;                                                     8
    }                                                               9
                                                                    10
    for (number i = 0; i < rowSize; ++i) {                          11
        if (i + currentSum <= rowSize) {                            12
            tmp[currentSize++] = i;                                 13
            allPartitionsRecursive(tmp, currentSize, problemSize, i + 14
                currentSum, rowSize, file);
            currentSize--;                                          15
        }                                                           16
    }                                                               17
}                                                                   18
```

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 2.14.

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

Listing 2.15 described below is essentially a wrapper around Listing 2.14. 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 2.14.

Listing 2.15. 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 2.14 in the following line. Being that Line 7 is the bottommost call on the call stack for Listing 2.14, 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 2.14, and line 10 closes and releases the file created in line 4.

The counterpart to Listing 2.15 is Listing 2.16, 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 2.16. Reading all partitions from a file.

```
number *readAllPartitions(length *numPartitions, number problemSize, number rowSize) {
    char fileName[32];
    sprintf(fileName, "all_partitions_%i_%i.dat", problemSize, rowSize);

    number *allPartitions = readFile(fileName, numPartitions, problemSize);

    if (allPartitions == NULL) {
        printf("Writing partitions file...\n");
        writeAllPartitions(problemSize, rowSize);
        allPartitions = readFile(fileName, numPartitions, problemSize);
    }

    return allPartitions;
}
```

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 2.15.
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 2.15, 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.

2.2.2 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 2.14, 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 2.17.

Listing 2.17. 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 2.17 is Listing 2.18. 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 2.17 is Listing 2.18 do to great effect. This option is not explored in this implementation, however. Details are omitted for Listing 2.18, as is essentially the same as Listing 2.17, only with the inverse operation.

Listing 2.18. 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 2.19 is also used as subroutine in Listing 2.20 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 2.19. 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 2.20 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 2.30, 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 2.20 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 2.20. 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 2.14, 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 2.20, the backtracking nature of the algorithm will push onto the call stack many other calls to Listing 2.20 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 2.17 to add this integer partition row to the current sum scratch memory.

18. Line 18 calls Listing 2.19 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 2.20 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 2.17 was called. Line 24 accomplishes that with a balancing call to Listing 2.18.

2.3 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 2.3.1. *Let* $S = \{3, 8, 1, 0, 9, 6, 4, 7, 10, 5, 2, 11\}$ *as in Ex. 1.2.12 and consider the combination matrix of self-derivation in Eq. 1-24. 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*

$$\left[\begin{array}{c|c} 6 & 6 \\ \hline 6 & 6 \end{array}\right] \quad . \tag{2-1}$$

The row S *is not in its canonical form. An application of Listing 2.8 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 2.3.2. *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 2.3.3. *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. 2.3.2 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 2.3.4. *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. □

2.3.1 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 2.3.3, 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 2.20, the algorithm presented in Listing 2.21 below also facilitates understanding of Listing 2.30, but it is not used in computing combination matrices of self-derivation. The algorithm described in Listing 2.22, on the other hand, is not only used in this implementation, but will also be the subject of future research.

Listing 2.21. Recursively computing all side vectors of a certain size.

```
void allSidesRecursive(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
    number start = currentSize == 0 ? 0 : tmp[currentSize - 1];                   7
                                                                                  8
    for (range i = start; i < classSize; ++i) {                                   9
        tmp[currentSize++] = i;                                                   10
        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 2.20. 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 2.30 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 2.21, 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 2.22. 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 2.22 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 2.15, Listing 2.23 wraps around Listing 2.22 to save a file in the same location where the the main executable is built, so the details are omitted. Unlike Listing 2.15, however, the output of Listing 2.23 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 2.23 would be a file of size $c^{m-1} = 546.75$ Gigabytes.

Listing 2.23. 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 2.24, whose details are omitted for being similar to Listing 2.16 above, including the way Listing 2.24 generates a new file if none has been already created.

Listing 2.24. Reading all tops from a file.

```
number *readAllTops(length *numTops, number problemSize, range classSize) {   1
    char fileName[NAME_SIZE];                                                  2
    sprintf(fileName, TOPS_FILE, problemSize, classSize);                      3
                                                                               4
```

```
    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;
}
```

2.3.2 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 2.25 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 2.25. 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 2.26 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 2.30, 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 2.30, 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 2.26. Defining a data structure to hold thread-specific data.

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

Listing 2.27 is used in Listing 2.28 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 2.27. Writing a single row from a combination matrix.

```
void writeRow(const number *row, number rowSize, range topSize, bool newLine,
    FILE *file) {
    for (range i = 0; i < topSize; ++i) {
        if (i > 0 && i % rowSize == 0)
            write(file, "| ");
```

```
                                                                    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 2.28. A text solution consists of the solution number within the thread-specific context, and all the rows appended using calls to Listing 2.27. 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 2.28. Writing an entire solution.

```
void writeSolution(thread_data *d, common_data *cd, FILE *solutionsFile) {
    write(solutionsFile, "Solution %lu:\n\n\n\n\n", d->counter);
    writeRow(d->tmpTops, cd->rowSize, cd->topSize, true, solutionsFile);

    for (number i = 0; i < cd->problemSize; ++i) {
        range topIndex = 0;
        number rowIndex = 0;
        const number *p = &cd->allPartitions[d->tmpSquare[i] * cd->problemSize
            ];
        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++;                                                    21
                    pj--;                                                          22
                }                                                                  23
            }                                                                      24
                                                                                   25
            topIndex += cd->rowSize;                                               26
        }                                                                          27
                                                                                   28
        if (d->tmpSide[i] < 10)                                                    29
            write(solutionsFile, "%i  |", d->tmpSide[i]);                          30
        else                                                                       31
            write(solutionsFile, "%i |", d->tmpSide[i]);                           32
                                                                                   33
        writeRow(d->currentTop, cd->rowSize, cd->topSize, false, solutionsFile     34
            );
        write(solutionsFile, "| %lu\n", d->tmpSquare[i]);                         35
    }                                                                              36
                                                                                   37
    write(solutionsFile, "\n");                                                    38
}                                                                                  39
```

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 2.27 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 2.27.

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 2.27 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 2.3.5. *Below is a sample output of Listing 2.28. 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*

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

```
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
```

2.3.3 Computing all possible solutions for a given combination matrix

The algorithm presented in Listing 2.30 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 2.29 as a subroutine to test if the a given representative of a row class can be matched with the top row. Listing 2.29 is very similar to the first inner for-loop in Listing 2.28 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 2.28 and Listing 2.29 lies in the fact that the top row is not immutable. Rather, it is some scratch memory space that gets changed every time Listing 2.29 is called. Another key difference is the fact that Listing 2.29 can fail to produce a match, whereas Listing 2.28 is only called once a solution already exists for all derived rows.

Listing 2.29. Matching a representative of a row class with a given top vector.

```
bool matchRowWithTop(number *top, const number *row, const number *partitions,
    number problemSize, number rowSize) {
    range topIndex = 0;
    number rowIndex = 0;

    for (number i = 0; i < problemSize; ++i) {
        number p = partitions[i];

        for (range j = topIndex; j < topIndex + rowSize; ++j) {
            if (p == 0)
                break;

            if (top[j] == row[rowIndex]) {
                top[j] = -1;
                rowIndex++;
                p--;
            }
        }

        if (p > 0)
            return false;

        topIndex += rowSize;
```

```
    }                                                                    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 2.28, 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 2.28. 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 2.28.
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 2.28.
12. Line 12 tests that the j^{th} entry in the top vector matches the row transform at the current row index. Unlike Listing 2.28, 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 2.29, then the j^{th} entry in the top vector will have been set to minus one, and Listing 2.29 will fail. Otherwise, line 13 will set the j^{th} entry in the top vector to minus one, which differs from Listing 2.28 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 2.28 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 2.28, 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 2.30 combines Listing 2.21 and Listing 2.20 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 2.29. If that succeeds, Listing 2.30 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 2.28 to label each derived row in the text output of a solution.

Listing 2.30. 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 2.28 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 2.20.
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 2.20 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 2.29 succeeds and a new call to Listing 2.30 is pushed onto the call stack.
15. Lines 15 to 17 do exactly what lines 14 to 16 do in Listing 2.20.
19. Similarly, lines 19 and 20 are identical to lines 18 and 19 in Listing 2.20. 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. 2.3.3 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 2.29. 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 2.30 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 2.29 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 2.18, similarly to what Listing 2.20 does in its line 24.

2.4 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 2.30. 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 2.30 to execute as row size and problem size grow. Listing 2.31 describes how thread-common data is allocated, and every call to Listing 2.31 should be balanced with a call to Listing 2.32 when resources are no longer needed. Listing 2.31 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 2.40, thus further details are deferred until the next section.

Listing 2.31. Allocating thread-common data.

```
void createCommonData(common_data *cd, int argc, char *argv[]) {
    number problemSize = (number) strtol(argv[1], NULL, 10);
    number rowSize = argc - 2;
    number *row = malloc(rowSize * sizeof(number));

    for (number i = 0; i < rowSize; ++i)
        row[i] = (number) strtol(argv[i + 2], NULL, 10);

    cd->problemSize = problemSize;
    cd->rowSize = rowSize;
    cd->topSize = problemSize * rowSize;

    bool rInvariant = isRetrogradeInvariant(row, rowSize);
```

```
    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 2.31 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 2.9 to determine if the row is retrograde-invariant, and line 14 is similarly a call to Listing 2.10 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 2.31, but will not be needed in the algorithms that depend on a thread-common data structure. Line 17 is a call to Listing 2.11, 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 2.16 and Listing 2.24, 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 2.4.1. *Let r be a retrograde-invariant row with size $n = 6$. Then the row class array computed by Listing 2.11 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*

$$T_0 \mapsto T_3, \cdots, T_5 \mapsto T_2, T_3I \mapsto T_0I, \cdots, T_5I \mapsto T_2I \ . \tag{2-3}$$

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*

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

Example 2.4.2. *Let* $r = \{0, 2, 3, 4, 5, 1\}$, *with* n, C *and* C_I *as in Ex. 2.4.1. 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*

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

Listing 2.32 below describes how the data allocated in Listing 2.31 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 2.32. Deallocating thread-common data.

```
void destroyCommonData(common_data *cd) {                    1
```

```
    free(cd->rowClass);
    free(cd->rMap);
    free(cd->riMap);
    free(cd->allPartitions);
    free(cd->allTops);
}
```

Listing 2.33 is a very simple procedure, as well. As it depends on a previous call to Listing 2.31, all thread-common data has already been allocated, and sizes computes. Listing 2.33 then allocates the necessary scratch areas that cannot be shared between different thread contexts, and sets the thread-specific solution counter to zero.

Listing 2.33. Allocating thread-specific data.

```
void createThreadData(thread_data *d, common_data *cd) {
    d->counter = 0;
    d->tmpSide = malloc(cd->problemSize * sizeof(number));
    d->currentTop = malloc(cd->topSize * sizeof(number));
    d->tmpTops = malloc(cd->problemSize * cd->topSize * sizeof(number));
    d->currentSum = malloc(cd->problemSize * sizeof(number));
    d->tmpSquare = malloc(cd->problemSize * sizeof(length));
}
```

Similarly to Listing 2.32, Listing 2.34 deallocates the heap memory that was allocated in Listing 2.33. Here too, every call to Listing 2.33 should be balanced with a call to Listing 2.34 when resources are no longer needed.

Listing 2.34. Deallocating thread-specific data.

```
void destroyThreadData(thread_data *d) {
    free(d->tmpSide);
    free(d->currentTop);
    free(d->tmpTops);
    free(d->currentSum);
    free(d->tmpSquare);
```

```
}                                                                    7
```

2.5 Computing all solutions for a given row class

This section outlines Listing 2.42, a procedure that wraps Listing 2.30, providing it with the necessary scaffolding to output all solutions computed to a text file. In a single-threaded context, Listing 2.42 will be furnished with all possible tops for a given problem size. In a multi-threaded context, however, Listing 2.42 will be given a range of tops to be processed by each thread. Listing 2.22 above computes the minimal subset of tops necessary for the operation of Listing 2.42. 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 2.42 itself handles computing tops that cannot be achieved from the results of Listing 2.22. Listing 2.35 creates a file to which solutions will be written.

Listing 2.35. 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 2.40 outlines the procedure for retrograde or retrograde inverse-invariant rows, whereas Listing 2.39 outlines the procedure for rows otherwise. The next three algorithms below are used as subroutines in Listing 2.39.

Listing 2.36. 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_0I or RT_0I.
5. Line 5 does the opposite of line 2, that is, it tests if the given transform is either T_0I or RT_0I, 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 2.37 is a lot simpler than Listing 2.36. 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 2.37. 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 2.38 is even simpler and just combines calls to Listing 2.36 and Listing 2.37 to return the retrograde inverse index of the provided input index.

Listing 2.38. 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 2.39 is designed to filter tops that can be achieved by transforming some previously seen top. It relies on the order in which Listing 2.22 outputs tops to filter out top invariances under the retrograde operation.

Listing 2.39. 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 2.38 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 2.37 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 2.42.

The purpose of Listing 2.40 is also to filter out tops that can be achieved from a previously seen top through some transformation, only that Listing 2.40 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 2.40 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 2.40. 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 2.40, 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 2.41 is used as a subroutine in Listing 2.42 copy the top to be processed by Listing 2.30 into the thread-specific data structure's scratch area for the top vector. A top, as computed in Listing 2.22, is an array of indices into the row class array. Listing 2.41 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 2.41. 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 2.42 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 2.42 can be made from different thread contexts, provided that each thread maintains their own thread-specific data structures. Listing 2.42 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 2.22 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 2.22. However, the retrograde of T is the top

$$R(T) = \{R(RT_0), R(T_0)\} = \{T_0, RT_0\} = \{0, 24\} = T \ . \tag{2-6}$$

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

$$G = \underbrace{RT_xI \times \cdots \times RT_xI}_{m \text{ times}} \ . \tag{2-7}$$

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 2.42. 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;

            if (cd->isInvariant && first >= cd->classSize / 2)
                break;

            if (cd->isInvariant && !unseen(combo, cd))
                continue;

            if (!cd->isInvariant && seen(combo, cd))
                continue;

            fillTop(d, cd, combo);
            memcpy(d->tmpTops, d->currentTop, cd->topSize * sizeof(number));
            memset(d->currentSum, 0, cd->problemSize * sizeof(number));
            allSolutionsRecursive(d, cd, solutionsFile, 0);
            topCounter++;
        }
    }

    fclose(solutionsFile);
    return topCounter;
}
```

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 2.35 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 2.40, 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 2.39, in which case line 17 skips this iteration.
19. Line 19 is a call to Listing 2.41 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 2.30 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.

CHAPTER 3
APPLICATIONS AND CONCLUSION

3.1 Single and multi-threaded command line tools

This section exemplifies the use of the framework developed in the previous chapter by defining two command line tools for computing combination matrices of self-derivation. The first of them computes all solutions for a given row and problem size using a single thread of execution, while the second utilizes several threads. The single-threaded application is reasonably fast for small row and problem sizes. As input sizes grow, the number of solutions grows dramatically, and a single thread of execution becomes insufficient.

Listing 3.1. A single-threaded command line tool.

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        printf("Enter a problem size and a row!\n");
        return -1;
    }

    common_data commonData;
    createCommonData(&commonData, argc, argv);

    thread_data data;
    createThreadData(&data, &commonData);

    clock_t start = clock();
    writeSolutions(&data, &commonData, 0, commonData.numTops);

    printf("%f seconds elapsed.\n", ((double) (clock() - start)) /
       CLOCKS_PER_SEC);
    printf("%lu solutions found!\n", data.counter);

    destroyThreadData(&data);
    destroyCommonData(&commonData);
```

```
                                                                    21
    return 0;                                                       22
}                                                                   23
```

1. The inputs are the standard command line inputs, that is, the number of arguments, and the array of strings containing them.
2. Line 2 checks that at least three arguments were passed and, if not, line 3 prints usage instructions to the console and line 4 returns an error.
7. Line 7 creates an empty thread-common data structure and line 8 calls Listing 2.31 to fill it.
10. Line 10 creates an empty thread-specific data structure and line 11 calls Listing 2.33 to fill it.
13. Line 13 computes the current time and stores it in a variable. Line 14 calls Listing 2.42 to write solutions for all tops to a single text file.
16. Line 16 computes the time after the call to Listing 2.42 in line 14, subtracts from it the start time computed in line 13, converts the result to seconds and prints it to the console. Line 17 prints to the console the number of solutions found.
19. Lines 19 and 20 free all allocated memory by calling respectively Listing 2.34 and Listing 2.32.
22. Line 22 simply exits the program normally.

Listing 3.2 below describes a data structure that is used in Listing 3.3 to manage multiple threads of execution. The framework described in the previous chapter was designed using only platform-independent components of the C programming language. For multi-threaded applications, the C programming language does not provide such platform-independent components. Listing 3.3 therefore is written in C++, which provides a standard library for managing threads in a platform-independent way, as well as an object-oriented paradigm which the creation, dispatching, and destruction of threads, thus greatly simplifying their use.

Listing 3.2. A class to manage concurrency.

```
class thread_manager {
public:
    thread_manager(int argc, char *argv[]) : threadData(std::thread::
        hardware_concurrency()) {
        createCommonData(&commonData, argc, argv);

        for (auto &data: threadData)
            createThreadData(&data, &commonData);
    }

    ~thread_manager() {
        for (auto &data: threadData)
            destroyThreadData(&data);

        destroyCommonData(&commonData);
    }

    long long runThreads() {
        auto numThreads = threadData.size();

        auto doWork = [this, &numThreads](int i, length startTop, length
            endTop) {
            writeSolutions(&threadData[i], &commonData, startTop, endTop);
            std::lock_guard<std::mutex> lock(mutex);
            numThreads--;
            condition.notify_one();
        };

        length startTop = 0;
        length offset = commonData.numTops / numThreads;
        auto maxThreads = numThreads - 1;
```

```
        auto startTime = std::chrono::high_resolution_clock::now();
        for (auto i = 0; i < maxThreads; ++i) {
            std::thread(doWork, i, startTop, startTop + offset).detach();
            startTop += offset;
        }
        doWork(maxThreads, startTop, commonData.numTops);
        std::unique_lock<std::mutex> lock(mutex);
        condition.wait(lock, [&numThreads] { return numThreads == 0; });
        auto stopTime = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(
            stopTime - startTime);
        return duration.count();
    }
    length getNumSolutions() const {
        length numSolutions = 0;
        for (auto &data: threadData)
            numSolutions += data.counter;
        return numSolutions;
    }
private:
    std::vector<thread_data> threadData;
    common_data commonData;
    std::mutex mutex;
    std::condition_variable condition;
```

};

62

3. The constructor of a thread manager in line 3 takes as inputs the same arguments passed on the command line, initializing the vector of thread-specific data structures declared in line 57 with the number of hardware cores in the CPU. Using more threads than the number of CPU cores does not necessarily improve performance. Line 4 then creates the thread-common data with a call to Listing 2.31, storing it to the member variable declared in line 58.

6. The for loop in line 6 iterates through the vector of thread-specific data structures and allocates memory for each one of them in line 7 with calls to Listing 2.33.

11. The destructor of a thread manager iterates through the vector of thread-specific data structures and freeing allocated memory in line 12 for each one of them with calls to Listing 2.34.

14. Line 14 frees the allocated thread-specific data in the destructor of a thread manager with a call to Listing 2.32.

17. Line 17 declares the*runThreads* method, which takes no inputs and outputs a big number. Line 18 stores in a variable the number of threads that will be used to compute the solutions.

20. Line 20 declares a lambda that will be executed by each thread. The lambda captures the instance of the thread manager, as well as a reference to the variable declared in line 18. Its inputs are the thread index, the top start index, and the top end index. Line 21 calls Listing 2.42 from each thread, using the thread index to provide each thread with a pointer to a thread-specific data structure. When the call returns, line 22 locks the mutex declared in line 59 to protect access to the *numThreads* variable. Line 23 decreases *numThreads*, and line 24 notifies the condition variable declared in line 60.

27. Line 27 creates a local variable inside *runThreads* to store the start top that will be given to each thread, and line 28 computes how many tops each thread will be asked

to compute. Line 29 stores the maximum number of threads minus one, and line 30 stores the current time.

32. Line 32 iterates over the number of threads minus one, creating and detatching a new thread in line 33. Line 34 then updates the start top index for each thread.

37. Line 37 calls the lambda declared in line 20 in the current thread, without dispatching a new one, thus utilizing all CPU cores available.

39. Line 39 uses the mutex declared in line 59 to lock the current thread. Line 40 then uses the lock and the condition variable declared in line 60 to block the current thread until *numThreads*, which is decreased by every thread, reaches zero, indicating that no more threads are running.

42. Line 42 computes the current time, and line 43 computes the execution time for all threads in milliseconds.

45. Line 45 simply returns the execution time computed in line 43 as a number.

48. Line 48 declares the*getNumSolutions* method, which returns a number. Line 49 initializes a local variable to zero, which will be used to return the total number of solutions for all threads.

51. Line 51 iterates through all thread-specific data structures and accumulates in line 52 the number of thread-specific solutions.

54. Line 54 simply retuns the value of the variable declared in line 49.

Listing 3.3 is the main entry point in a multi-threaded application that computes all possible combination matrices of self-derivation given a problem size and a row. To do so, it creates an instance of a thread manager, and prints to the console the results of calls to *runThreads* and to *getNumSolutions*. Due to its simplicity and similarity with Listing 3.1, further details are omitted.

Listing 3.3. A multi-threaded command line tool.

```
int main(int argc, char *argv[]) {                                    1
    if (argc < 3) {                                                   2
```

```
        printf("Enter a problem size and a row!\n");
        return -1;
    }

    thread_manager threadManager(argc, argv);
    std::cout << threadManager.runThreads() << " milliseconds elapsed." << std
        ::endl;
    std::cout << threadManager.getNumSolutions() << " solutions found!" << std
        ::endl;

    return 0;
}
```

3.2 Musical applications

This section shows how various outputs of Listing 2.42 may be used to compose music. Most of the musical examples in literature around self-derivation approach it from the standpoint of 12-tone practice. There is, however, no requirement that theses constructs be focused on the pitch domain, nor that they be restricted to base 12. For this reason, the examples presented here attempt to explore the practical use of self-derivation with sizes different than 12, as well as to dimensions other than pitch. The first musical application below describes the use of self-derivation in a composition for solo percussion. The instrument setup consists five tom-toms, five cowbells, and five woodblocks, all of different sizes. The tom-toms are placed at the lowest height, and the woodblocks at the highest, with the cowbells in the middle. All three rows of instruments are assembled from left to right according to pitch. Alternatively, tom-toms may be replaced with rototoms, cowbells with Almglocken, and woodblocks with temple blocks. The point of departure for the musical passage shown in Fig. 3-1 is a freely-composed, four-bar rhythmic phrase, repeated three times. The four 3×15 combination matrices of self-derivation described in Listing 3.4 are used to dictate instrument changes in this passage. Each matrix row

corresponds to a row in the instrument setup, that is, the bottom matrix row corresponds to the tom-toms, the top row to the woodblocks, and the center row to the cowbells.

Listing 3.4. Output of Listing 2.42 for problem size three and row $\{0, 2, 3, 1, 4\}$.

```
Solution 37:

     0  2  3  1  4  | 4  2  1  3  0  | 0  3  1  2  4
15 |          1     | 4  2     3  0  |                  | 8
2  |    2        4  |                | 0  3  1        | 9
19 | 0     3        |       1        |          2  4  | 10

Solution 38:

     0  2  3  1  4  | 4  2  1  3  0  | 0  3  1  2  4
15 |          1     | 4  2     3  0  |                  | 8
19 | 0     3        |                |       1  2  4  | 9
13 |    2        4  |       1        | 0  3           | 10

Solution 39:

     0  2  3  1  4  | 4  1  3  2  0  | 0  2  3  1  4
0  |                |             0  |    2  3  1  4  | 0
10 |             4  |    1  3  2     | 0              | 7
0  | 0  2  3  1     | 4              |                | 17

Solution 40:

     0  2  3  1  4  | 4  1  3  2  0  | 0  2  3  1  4
14 |                |       3        | 0  2     1  4  | 0
4  |             4  |    1     2  0  |       3        | 7
0  | 0  2  3  1     | 4              |                | 17
```

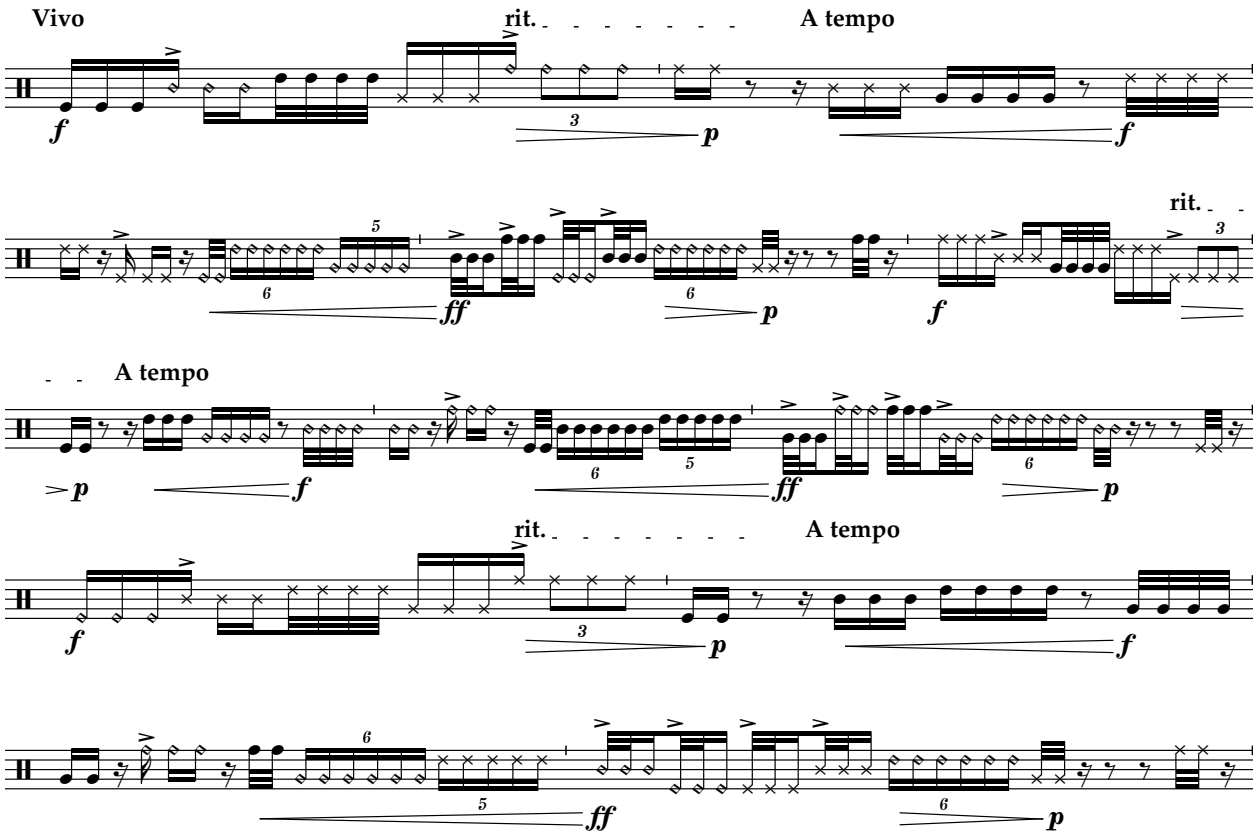


Figure 3-1. Using self-derivation to change instruments in a percussion setup.

Each of the five instruments from each family is notated in Fig. 3-1 in one of the staff lines, from bottom to top, according to ascending pitch. To discern between the instrument families, tom-toms are notated with a normal note head, cowbells with a diamond, and woodblocks with a cross. The base row $\{0, 2, 3, 1, 4\}$ in Listing 3.4 was chosen specifically not to be retrograde invariant. The set of solutions $\{37, 38, 39, 40\}$, on the other hand, was chosen to fit a particular musical structure. The semi-magic squares in solutions 37 and 38 are the same, and so are the semi-magic squares in solutions 39 and 40. The side vectors in solutions 37 and 38 differ by one row, whereas the side vectors in solutions 39 and 40 differ by two. Moreover, the side vectors in solutions 37 and 38 contain no representatives of the row class contained in the side vectors in solutions 39 and 40, and vice-versa. Therefore the entire passage can be broken into two statements. The motivation for this choice is to make the three repetitions of the basic rhythmic

phrase as diverse as possible, avoiding any alignment between the repetitions and the solutions. This is accomplished by the fact that three is relatively prime to four and to two. Dynamics and tempo makings, nevertheless, are aligned with each repeated statement of the rhythmic phrase, since these two musical dimensions belong to the intuitive sphere of this musical composition.

The next musical application is a passage for string quartet where four 4×32 combination matrices of self-derivation are used to determine pitch. The four instruments are associated with the four rows of the combination matrix in the canonical score order. An 8-tone row is used as the indices of an octatonic scale. The basic index row is $S = \{0, 2, 5, 6, 3, 1, 7, 4\}$. Listing 3.5 displays the output of a modified version of Listing 2.42 where the base row is $RT_4(S) = \{0, 3, 5, 7, 2, 1, 6, 4\}$. The octatonic row is $\mathcal{O} = \{0, 2, 4, 5, 6, 8, 10, 11\}$, so that $S \mapsto \mathcal{O}$ is the injective mapping

$$0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 4, 3 \mapsto 5, 4 \mapsto 6, 5 \mapsto 8, 6 \mapsto 10, 7 \mapsto 11 \ . \tag{3-1}$$

Listing 3.5. Output of Listing 2.42 for problem size four and row $\{0, 3, 5, 7, 2, 1, 6, 4\}$.

```
Solution 86:

     0  3  5  7  2  1  6  4  | 0  3  5  7  2  1  6  4  |
2  |              2           |        5  7              |
28 | 0                    6   |     3        2           | ...
23 |    3  5                 | 0                 1  6  4  |
19 |          7     1     4  |                          |

   | 2  5  7  1  4  3  0  6  | 7  1  4  5  2  0  6  3
   |          1  4  3  0  6  |                          | 62
   |    5  7                 |     1  4                 | 93
   | 2                       | 7                        | 101
   |                         |           5  2  0  6  3  | 106
```

Solution 116:

```
     0  3  5  7  2  1  6  4  | 0  3  5  7  2  1  6  4  |
14 |                    6     |    3                    |
30 |             2           | 0     5              4  | ...
19 |          7     1     4  |                         |
0  | 0  3  5                 |          7  2  1  6     |

   | 4  7  1  3  6  5  2  0  | 6  3  1  7  4  5  0  2
   |                         |       1  7  4  5  0  2  | 50
   |    7  1  3  6           |                         | 67
   |                5  2  0  | 6  3                    | 109
   | 4                       |                         | 125
```

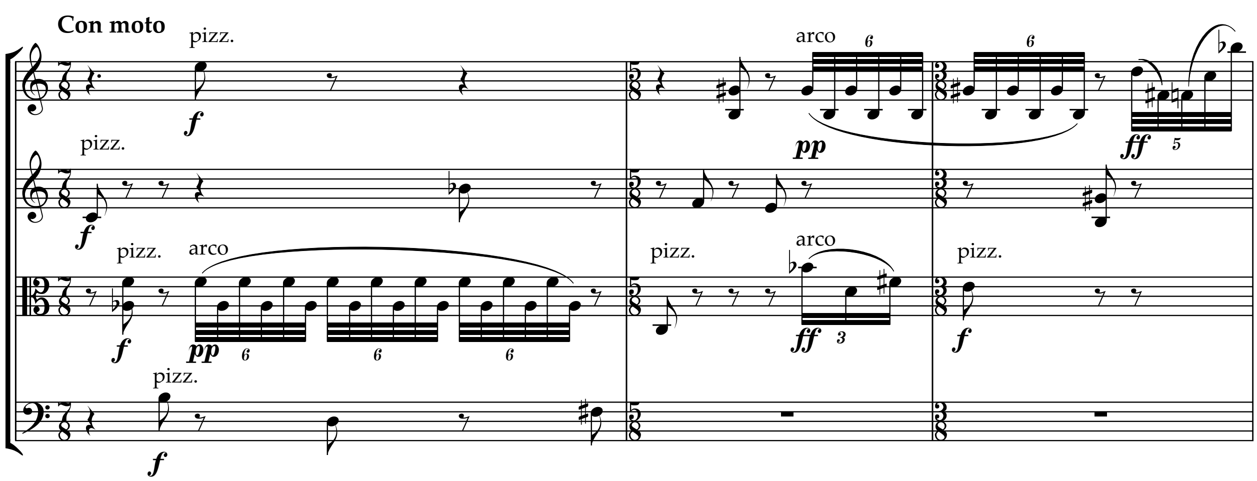
Solution 161:

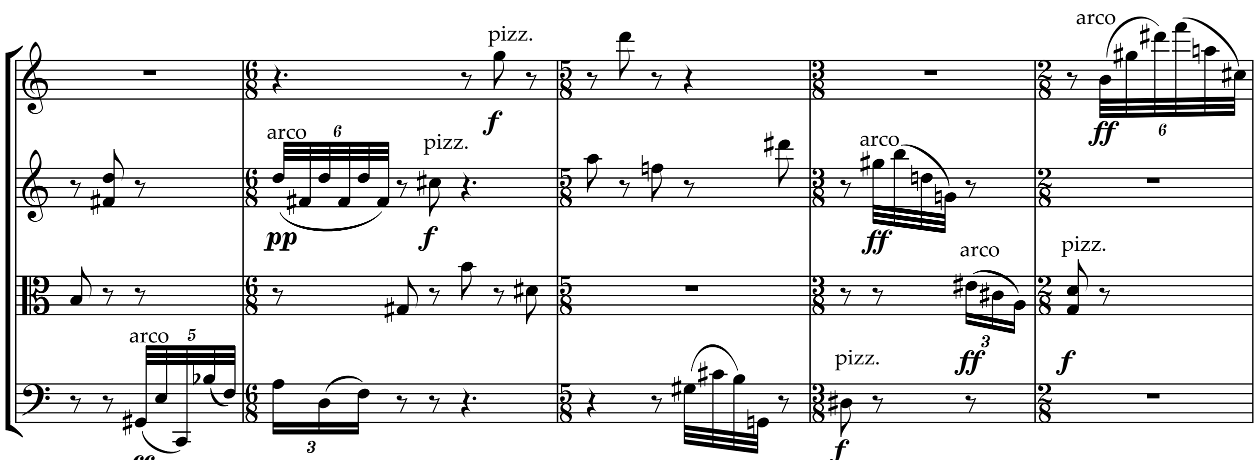
```
     0  3  5  7  2  1  6  4  | 0  3  5  7  2  1  6  4  |
0  |                         | 0  3  5                 |
7  |                         |          7  2        4  | ...
26 |                    6  4 |                1        |
0  | 0  3  5  7  2  1        |                   6     |

   | 7  2  4  6  1  0  5  3  | 0  3  5  7  2  1  6  4
   | 7  2                    |                1  6  4  | 24
   |          6  1  0  5  3  |                         | 27
   |                         | 0  3  5  7  2           | 85
   |       4                 |                         | 156
```

Solution 175:

```
     0  3  5  7  2  1  6  4  | 0  3  5  7  2  1  6  4  |
0  |                         | 0  3  5                 |
9  |                1  6     |                      4  | ...
```

```
22 |                 2           4 |             7                 |
0  | 0   3   5   7                 |                 2   1   6     |

   | 0   5   3   1   6   7   2   4 | 1   6   4   2   7   0   3   5
   |                     7   2     | 1   6   4                     | 24
   |                               |             2   7   0   3   5 | 85
   | 0   5   3   1   6             |                               | 90
   |                             4 |                               | 140
```





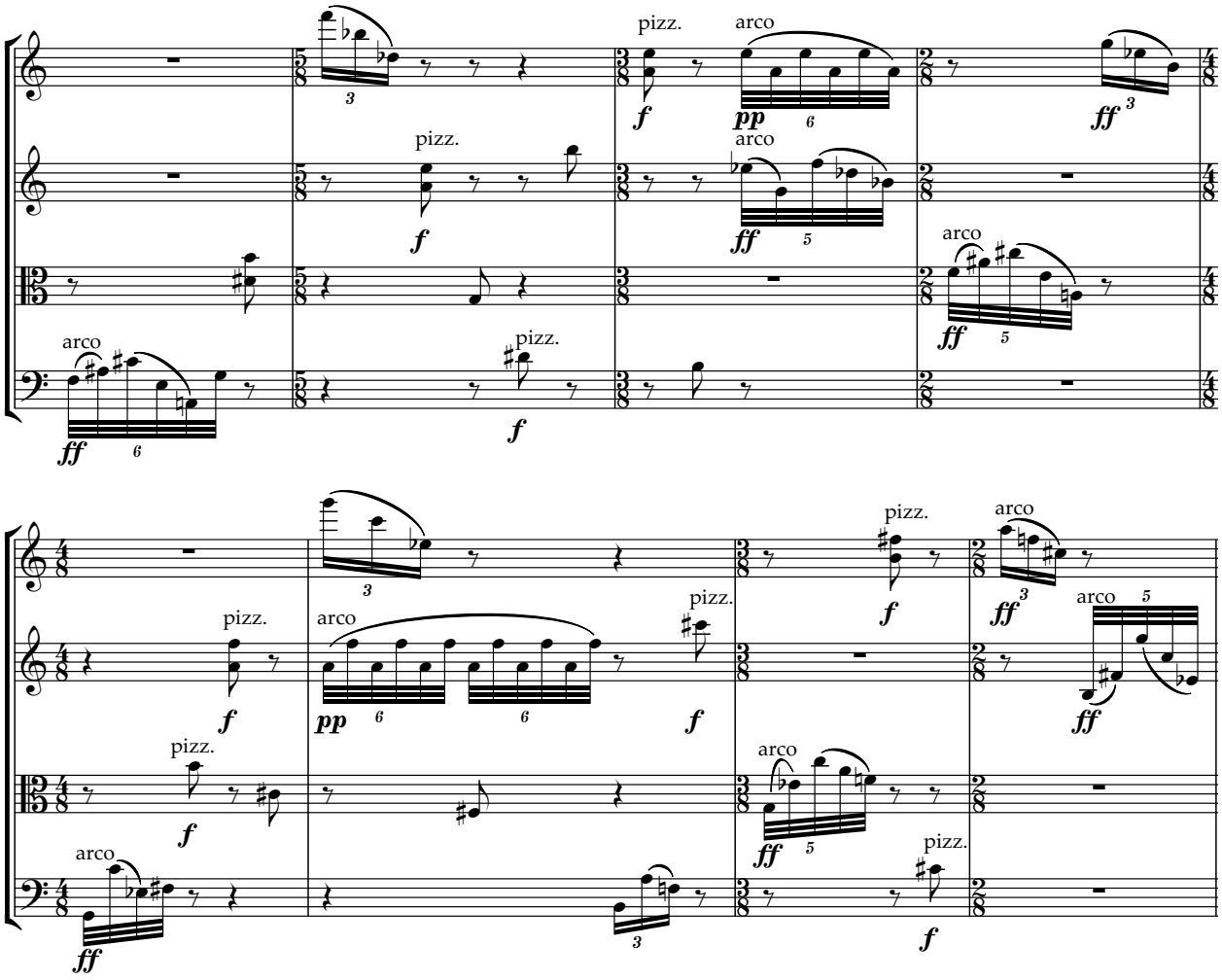


Figure 3-2. Using self-derivation to map an octatonic scale to an 8-tone row.

The output of Listing 2.42 presented in Listing 3.5 is shown as a musical score in Fig. 3-2. The chosen set of solutions is $\{86, 116, 161, 175\}$, and these choices were based mostly on how each semi-magic square explored all instruments. The musical realization is straightforward in regards to how the rows of the combination matrices are mapped to the four instruments. Rhythmically, the contents of the combination matrices are realized depending on the contents of each semi-magic square. Square cells equal to one, two or zero are realized as 8^{th}-notes or rests. Square cells equal to three are realized as 16^{th}-note triplets, cells equal to four are realized as 32^{th}-notes, and cells equal to five and six are realized as 32^{th}-note quintuplets and sextuplets, respectively. Cells equal to two

are struck simultaneously as double-stops and prolonged as 32^{th}-note sextuplet tremolos. The tremolo order is the same as the square cell order, and the tremolos have a left and right spacing of one 8^{th}-note, that is, tremolos begin one 8^{th}-note after the double-stop, and end one 8^{th}-note before the next non-empty square cell for that instrument. Tremolos are altogether omitted when there is not enough time for them, hence there needs to be at least three 8^{th}-note rests between a double-stop and the next non-empty cell, so that the tremolo may be surrounded by rests. Besides the tremolos, the remainder of the passage is mostly pointillistic. To emphasize this characteristic, single and double-stop 8^{th}-notes are marked with pizzicati. The time signature changes in this realization are also influenced by the combination matrices. Given the aforementioned durations for each square cell size, every combination matrix column corresponds in Fig. 3-2 to a bar, such that the time signature of every bar is the sum of the durations of the column cell in the corresponding combination matrix.

3.3 Conclusion

The algorithms, applications and examples presented in this paper represent a major advancement in the study of combination matrices of self-derivation. Arguably the most notable contribution is an algorithm that is simple enough to allow computations by hand, while at the same time fast enough to generate thousands of solutions per second when run by a computer, and general enough to compute solutions for arbitrary problem and row sizes. The ideas in [11] are invaluable for an understanding and classification of various derivation techniques from a set-theoretic perspective. It also contributes enormously to understanding how a particular class of solutions for the self-derivation problem can be constructed by exploring the permutation cycles of RT_nI operations. What is missing in [11] is an efficient way to compute combination matrices of self-derivation in a general setting. This shortcoming is addressed in [3], which restricts its focus to the self-derivation problem and introduces an efficient algorithm to compute solutions for the 12-tone case. Even though the algorithm devised in [3] can be generalized

to other row sizes, it is not a simple algorithm to understand or apply. More importantly, it intentionally avoids computing all possible solutions, sacrificing the less interesting ones for speed. The void that Listing 2.30 intends to fill is exactly that of an algorithm that can produce a complete set of solutions given a row and a combination matrix size. The achieved result is straightforward to understand and implement. Devising this algorithm required understanding the problem of self-derivation from from a new point of view, namely that of a structure that can be broken down into three basic constituent parts: a linearized top, a semi-magic square, and an array of derived row transforms. In this formulation, a top, a square and a side are necessary and sufficient to characterize a solution, that is, every solution is uniquely determined by them according to Th. 2.3.2. Such analysis of the problem paves the way for a much deeper understanding of self-derivation, especially when combined with a simple and fast algorithm capable of unveiling a large amount of solutions within reasonable time. Once the self-derivation problem was broken into parts, each part had to be studied and understood in detail. This study resulted in reducing the body of solutions into equivalence classes under the action of RTn_I, the number and size of which depending on retrograde or retrograde inverse-invariance of the row. This body of solutions also groups into classes solutions which are equivalent under permutations of the side vector, leading to Cor. 2.3.3, and further into permutations of repeated semi-magic square rows. Although a significant advance, this paper is still a small toward a more complete understanding of self-derivation, falling short in many aspects. Some of theses aspects will be the subject of the rest of this chapter.

One major weakness of the present implementation is the way tops are pre-computed and stored as binary data. Although this method does improve execution speed, it is impractical, if not impossible, to handle large rows and problem sizes, as the number of possible tops becomes quickly unwieldy. One possibility to mitigate this problem would be to compute tops dynamically, compromising execution speed somewhat. In itself, this workaround would not help with the issue that Listing 2.30 can produce an unmanageable

amount of solutions, many if not most of them being rather uninteresting compositionally. There is a contradiction between a research goal in the present implementation, which is to devise an algorithm that produces a complete body of solutions, and the compositional goal of utilizing combination matrices to achieve meaningful counterpoint. Modifying the current implementation to focus on more interesting combination matrices is very easy. Besides the choice of row itself, the main contributor to uninteresting solutions is the semi-magic square itself. Therefore an implementation that restricts the search to interesting squares should produce equally interesting solutions according to the choice of row. Even though the current research is biased toward the understanding of self-derivation in all generality, the methodology described is still suitable for more restrictive applications. Restricting the range of the output of Listing 2.30 may also represent a practical way of studying self-derivation itself in cases where the number of solutions is too large. This is an open topic of discussion, however, as what is considered too large is a matter of context. In a distributed system with hundreds of thousands of machines, space requirements and computational times have indeed a completely different meaning. Another argument is that improving how solutions can be further reduced into equivalence classes may involve first being able to produce all solutions. There are several additional ways in which the body of solutions produced by Listing 2.30 may be amenable to reduction into representatives of larger and larger equivalence classes. One example is by allowing multiplication, that is, considering the action of the larger group RT_nMI. Multiplication in rows of arbitrary size generalizes to studying the group of automorphisms of $\mathbb{Z}/n\mathbb{Z}$, which in turn impacts how row classes computed and retrograde invariances are filtered. Another example is to allow cyclic rotations of the row. Grouping rows into equivalence classes that include their cyclic rotations is thoroughly described in [2]. The compositional use of combination matrices of self-derivation that include the cyclic rotated transforms of the row can also be seen in [10]. Ex. 1.2.14 shows how folding a 2×12 combination matrix of self-derivation into a 4×12 combination matrix can be

achieved by simply repeating the top, that is, extending the top $\{T_0, RT_0\}$ into the top $\{T_0, RT_0, T_0, RT_0\}$. In Ex. 1.2.14, the derived rows contain cyclic rotated transforms of the original row. Yet another example of how solutions may be reduced is then given by how solutions for *different* problem sizes can relate by the technique of folding, whether these relations contemplate cyclic rotated transforms of the row or not. A particularly interesting consequence of regarding combination matrices of self-derivation as a top, a square and a side is that the set of semi-magic squares have some algebraic structure. It may be possible to reduce the set of solutions output by Listing 2.30 further, but more important could be the potential for generating musical syntax, as any operation that transforms a semi-magic square into another can motivate transforming a combination matrix into another. The action of RT_nI on a combination matrix preserves the semi-magic square, therefore a transformation of the square itself may lead to more interesting musical syntax. Moreover, complete sets of solutions for a problem size and row can also help understanding how these solution sets relate for rows that are generated from the same aggregate realization, as described in [11]. Comparing sets of solutions, however, brings out a fundamental problem with Listing 2.30, which is the fact that solutions are not output in a standardized order. Given an orbit of solutions under the action of RT_nI, a representative for the orbit may be picked by selecting the element where the first derived row is in canonical form. That would ensure that solution sets for different but related rows be more easily compared.

Improving the present implementation may also be considered from the standpoint of execution speed and space requirements. The latter point is would involve outputting solutions in a minimized form, such as top, square and side indices, for example. A simple method similar to Listing 2.28 can then be used to reconstruct the solution as a combination matrix. The former point, that is, improving execution speed, is more involved. A simple optimization would be to use SIMD to vectorize vector addition and subtraction, which are fundamental components used in constructing and backtracking

semi-magic squares. A more complicated possibility would be to use general purpose computing on the GPU card to increase the number of threads. Although possible, this approach brings many complications. The most notable is that GPU programming is very suitable for applications that where loops broken down into into smaller ones and computed by multiple threads that load a single compute kernel. The recursive nature of Listing 2.30 goes against this paradigm, and even though recursion is possible on the GPU, it is platform-specific and arguably not the best use if GPU programming. In fact, although GPU programming can be platform independent to an extent, extracting every ounce of performance would likely involve getting into the implementation details of each platform. As mentioned previously, the use of distributed systems can greatly reduce computation times and make space constraints less problematic. It would also require less modifications to Listing 2.30. At the same time, the code necessary to distribute the execution of Listing 2.30 may be more platform-independent and not require the modifications necessary to extract compute kernels from Listing 2.30 and translate its recursive calls into iterative ones. The difficulty with distributed system is either the access to a data center, or the cost of building one of a decent size. The use of self-derivation in real-time applications for electroacoustic music represents another area to which the ideas presented here can be extended. In these applications, the current execution speed of Listing 2.30 may be adequate and no further optimizations be needed. By nature, combination matrices of self-derivation would fit best within a real-time music application in the sphere of event generation. That may or may not involve MIDI, but the more important observation is that generating events usually occurs at a much slower rate than the digital signal processing of audio samples. The modifications necessary to adapt Listing 2.30 to a real-time application might involve not only how many combination matrices are output at a time, but also how these matrices are formatted. One idea is to pair each entry of the top with a timestamp. This vector of timestamps could be algorithmically generated, based on a time-point system, or even triggered by human

interaction in live scenarios. Another idea is to pair each entry of each derived row with information regarding their register, if those are being used as pitch classes. Derived rows could also be marked with orchestration information, which in an electroacoustic context could mean the sound source triggered by an element of the derived row, with the derived row itself representing an entire track. For all these example applications, the changes to Listing 2.30 would be minimal, and likely only involve adding parameters to it to restrict the output, as well as changing the output to an schematic representation of a combination matrix.

APPENDIX A
THE MALLALIEU PROPERTY

Consider the $\mathsf{T}_n\mathsf{M}\mathsf{I}$ class of 12-tone rows with representative

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

This series has the remarkable property that, appending it with 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 A.0.1. *Let* $S^* = \{0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6, *\}$. *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* $\mathsf{T}_1(S)$. *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 [4, 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 $\mathsf{T}_n\mathsf{M}\mathsf{I}$.

Proposition A.0.2. *[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, the cases where $k \in \{5, 7, 11\}$ do not need to be considered, as $\mathsf{T}_5 = \mathsf{M} \circ \mathsf{T}_1$, $\mathsf{T}_7 = \mathsf{MI} \circ \mathsf{T}_1$, and $\mathsf{T}_{11} = \mathsf{I} \circ \mathsf{T}_1$. Hence, without loss, 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, 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 [4, 278]. It is easy to see that such an array has the same structure as the array S^* constructed above, if multiplication is considered 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 the extra symbol can be dropped by simply operating on the indices from 1 to $p - 1$. The row of order numbers shall still be referred to as S^*, however, the context making it clear whether it is being constructed 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 A.0.3. *Let* $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ *and* S^* *be as above. Then* $\mathsf{M}_3(S^*) = \{3, 6, 9, 12, 2, 5, 8, 11, 1, 4, 7, 10\}$*, 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 $\mathsf{T}_n\mathsf{MI}$ 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 relying on the isomorphism that constitutes the proof of A.0.4 is not possible in general. Whenever possible, however, the existence of mallalieu rows is easily verified.

Proposition A.0.4. *[4, 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 $|\operatorname{Aut}(\mathbb{Z}/(p-1)\mathbb{Z})| \geq 1$ for every prime p, every $(p-1)$-TET system is capable of producing a mallalieu row, as desired. □

In face of A.0.4, A.0.2 becomes just the special case where $p = 13$, as demonstrated in the next example.

Example A.0.5. *[5, 8] [1, 9] The number of isomorphisms $\mathbb{Z}/12\mathbb{Z} \to (\mathbb{Z}/13\mathbb{Z})^\times$ is equal to*

$$|\operatorname{Aut}\left((\mathbb{Z}/12\mathbb{Z})^\times\right)| = 4 \ . \tag{A-2}$$

These isomorphisms can be constructed by mapping a generator of $\mathbb{Z}/12\mathbb{Z}$, say $\bar{1}$, to the generators of $(\mathbb{Z}/13\mathbb{Z})^\times$, namely $\bar{2}, \bar{6}, \bar{7}$ and $\overline{11}$. Explicitly, the four maps $i \pmod{12} \mapsto 2^i \pmod{13}$, $i \pmod{12} \mapsto 6^i \pmod{13}$, $i \pmod{12} \mapsto 7^i \pmod{13}$ are obtained, and $i \pmod{12} \mapsto 11^i \pmod{13}$. The verification that these maps are well defined and bijective

is left to the reader. Denote the first map by φ*. Then*

$$\varphi(a+b) = 2^{a+b} = 2^a \cdot 2^b = \varphi(a) \cdot \varphi(b) \ , \tag{A-3}$$

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*

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

which by A.0.2 is one of the untransposed 12-tone rows that have the mallalieu property.

REFERENCES

[1] Babbitt, M. 1976. Letter: The Row 0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6. In Theory Only **7**:9. http://name.umdl.umich.edu/0641601.0002.007.

[2] Fripertinger, H., and P. Lackner. 2015. Tone rows and tropes. Journal of Mathematics and Music **9**:111–172. http://dx.doi.org/10.1080/17459737.2015.1070088.

[3] Kowalski, D. 1987. An Algorithm and a Computer Program for the Construction of Self-Deriving Arrays. In Theory Only **9**:27–49. http://name.umdl.umich.edu/0641601.0009.005.

[4] Lewin, D. 1966. On Certain Techniques of Re-Ordering in Serial Music. Journal of Music Theory **10**:276–287. http://www.jstor.org/stable/843244.

[5] Lewin, D. 1976. Letter: The Row 0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6. In Theory Only **7**:8. http://name.umdl.umich.edu/0641601.0002.007.

[6] Martino, D. 1961. The Source Set and Its Aggregate Formations. Journal of Music Theory **5**:224–273. http://www.jstor.org/stable/843226.

[7] Morris, R. 1976. Serial Forum: More on 0, 1, 4, 2, 9, 5, 11, 3, 8, 10, 7, 6. In Theory Only **7**:15–17. http://name.umdl.umich.edu/0641601.0002.007.

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

[9] Rahn, J. 1975. Gentle Reminder no. 1: Two Common-Tone Theorems. In Theory Only **1**:10–11. http://quod.lib.umich.edu/g/genpub/0641601.0001.002.

[10] Scotto, C. 2000. A Hybrid Compositional System: Pitch-Class Composition with Tonal Syntax. Perspectives of New Music **38**:169–222. http://www.jstor.org/stable/833592.

[11] Starr, D. 1984. Derivation and Polyphony. Perspectives of New Music **23**:180–257. http://www.jstor.org/stable/832915.

[12] Westergaard, P. 1966. Toward a Twelve-Tone Polyphony. Perspectives of New Music **4**:90–112. http://www.jstor.org/stable/832218.