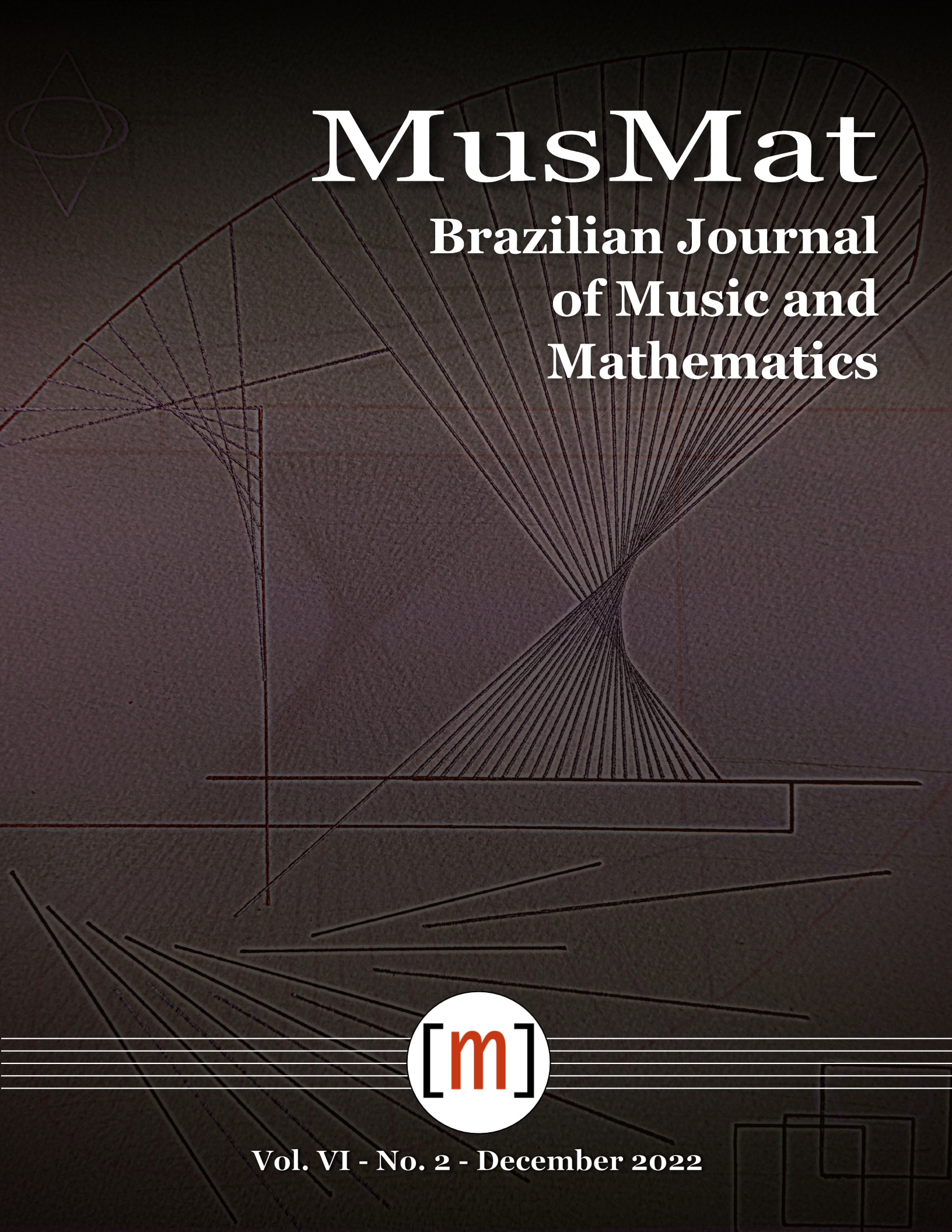


MusMat

Brazilian Journal of Music and Mathematics

Vol. VI - No. 2 - December 2022

MusMat

Brazilian Journal of Music and Mathematics

Vol. VI - No. 2

MusMat Research Group (Ed.)

Graduate Program in Music
Federal University of Rio de Janeiro (UFRJ)
Rio de Janeiro - December 2022

MusMat • Brazilian Journal of Music and Mathematics

MusMat - Brazilian Journal of Music and Mathematics. MusMat Research Group (Ed.). Rio de Janeiro: Federal University of Rio de Janeiro (UFRJ), School of Music, Graduate Program in Music, 2022. Vol. VI, No. 2.

DOI: 10.46926/musmat.2022v6n2

Contents

Articles

Foreword

The MusMat Research Group is pleased to announce the publication of this new issue of the *MusMat • Brazilian Journal of Music and Mathematics*. The second number of our sixth volume is composed of four papers. The edition opens with "Contextual Dyadic Transformations, and the Opening of Brahms's Op. 5 as a Variation of the Opening of Schumann's Op. 14", a paper by Scott Murphy where he discusses, from a group-theoretical perspective, Schumann's influence on Brahms's compositional style, by showing that the latter's opening of his Op. 5 is a variation of the former's opening of his respective Op. 14. Following, we have the paper "Python Scripts for Rhythmic Partitioning Analysis", by Marcos da Silva Sampaio and Pauxy Gentil-Nunes, in which they present a Python implementation of the software *Parsemat*, initially implemented in MatLab/Octave programming languages and extremely helpful to perform rhythmic partitioning analysis. Moreover, the authors also expand *Parsemat* by adding access to measure indications of each partition, annotation of texture information into digital scores, and many other features. Next, we have a paper on a topic also discussed in the MusMat 2022 Conference, "On Xenakis's Games of Musical Strategy", by Stefano Kalonaris. This work discusses Xenakis's pieces based on zero-sum games (*Duel*, *Stratégie*, and *Linaia-Agon*), and reveals axiomatic inconsistencies between the game-theoretical model and the musical implementations biased toward the composer's aesthetic preference. Finally, we close this issue with a paper by Charles Ames, "Building a Knowledge Base of Rhythms", that proposes a software to create, store, and access rhythmic patterns formed by *rests*, *attacks*, and *ties* for compositional purposes. He also demonstrates his program in several examples and creates parallels with several of his previous pioneering tools on computer-assisted composition. Enjoy the reading!

MusMat Group

December 2022

Contextual Dyadic Transformations, and the Opening of Brahms's Op. 5 as a Variation of the Opening of Schumann's Op. 14

Scott Murphy
University of Kansas
smurphy@ku.edu
Orcid: 0000-0001-7766-0777
DOI: 10.46926/musmat.2022v6n2.1-16

Abstract: *A contextual transformation that acts on the diatonic content of outer-voice dyadic harmony provides a way to demonstrate another correspondence between the openings of the F-minor piano sonatas by Robert Schumann and Johannes Brahms beyond those already recognized in current research. This contextual transformation also models some variation procedures within other music of Brahms and his eighteenth-century predecessors, suggesting an understanding of any influence of Schumann's opening on Brahms's compositional procedures within the framework of variation.*

Keywords*: Robert Schumann. Johannes Brahms. Sonata. Variation. Contextual transformation. Influence.*

I. Historical Context and Article Outline

30th September 1853 marks a significant date in the history of canonic western art music: on this Friday, the twenty-year-old Johannes Brahms first visited the home of Robert and Clara Schumann in Düsseldorf, Germany. With him, Brahms carried two completed slow movements—what would become the second and fourth movements—for his third piano sonata, but its fast movements were not yet composed, including the first movement ([3, p. 371]). During the four weeks of Brahms's stay in Düsseldorf, Clara played for him Robert's third piano sonata in F minor, op. 14, which begins as shown in Figure 1a. On 26 December 1853, Brahms sent off for publication as his fifth opus his completed third piano sonata in F minor in five movements, constituting the "most solid and impressive piece he had yet written" ([17, p. 126]). This sonata begins as shown in Figure 1b.

Some scholars recognize similarities between the openings of the two composers' sonatas (e.g. [7]). In arguably the most exhaustive study of Brahms's three piano sonatas, Gero Ehlert

Received: June 17th, 2022
Approved: December 21st, 2022

([6, p. 325]) proposes three likenesses, shown with colored shaded regions in Figure 1. The blue shading indicates an initiating three-note treble motive descending stepwise from A flat to F, the red shading indicates the bass's chromatic descent—sometimes called a "lament bass"—from F down to C, and the green shading indicates both an aggressive cadential arrival on a downbeat dominant triad with E in the treble and a trochaic ebb onto lower pitches on the second beat with C in the treble.[1] Nonetheless, Ehlert recommends against inferring too much from these similarities, concentrating instead on the considerable contrasts between their openings, such as the differences in their melodies.

In this article, I propose that these melodies, particularly the notes enclosed in Figure 1, closely correspond, lending additional support to the hypothesis that Brahms's opening is a variation of Schumann's. This proposed correspondence recruits a new type of contextual transformation **G** that expands an ordered pair of pitches while preserving its interval class. Section II of this article defines and demonstrates **G**, Section III shows how **G** serves as the basis for variation in three passages from Brahms's oeuvre composed before and after 1853, and Section IV returns to Schumann's and Brahms's sonatas, enlisting **G** to draw a connection between them.

II. The Contextual Dyadic Transformation **G**

A contextual transformation, relative to a usual (noncontextual) transformation, requires some additional knowledge—the "context"—about the input in order to determine the output. David Lewin ([14], [16]) cultivated this concept, which was further explored by Philip Lambert [13], Joseph Straus [20], and others. For an example, the transformation "invert around C" is a noncontextual transformation, because no knowledge is needed beyond the content of the pitches being transformed to calculate the result of the transformation. If the inputted pitch-class set contains a B, then the output of this "invert around C" transformation will contain a C sharp, regardless of the rest of the input. However, the transformation "invert around the major third," applied exclusively to a major or minor triad, is a contextual transformation, because it requires an extra assessment of the input to know the output.[2] If the input contains a B, then the output of this "invert around the major third" transformation will contain a C sharp if and only if the entire input is an E-major triad.[3]

To compare the openings of Schumann's and Brahms's F-minor piano sonatas, I present a contextual transformation that I call **G** (for Grow) that acts on ordered duples of pitches. For unequal pitches x and y, the transformation **G** changes <x, y> into <n, y> (and **G'** changes <y, x> into <y, n>) such that (1) the unordered sets {x, y} and {n, y} are pitch-class transpositions of one another, (2) x is between and unequal to both n and y, and (3) the difference between x and n is minimal. This transformation resembles Straus's generalized contextual inversions [20], in that they maintain both set class and one or more common tones (and, ordering aside, a transposition of a dyad could as easily be considered as an inversion). However, whereas Straus [20] concerns trichords, tetrachords, and pentachords, he does not address dyads. This transformation undergirds something resembling Tymoczko's dyadic form of circular voice-leading space [21], but Tymoczko privileges conjunct voice-leading distance—therefore, he favors the nearly even interval class 5 (ic5)—and he eschews transformational methodology.

[1] The first edition of Schumann's sonata has a D flat in the treble instead of the C at this point (m. 7).

[2] The "invert around the major third" contextual transformation, applied particularly to a major or minor triad, is called **REL** in Lewin [15], and abbreviated to **R** in neo-Riemannian theory ([12], [14]).

[3] To invert around a pitch or set of pitches is to invert such that the pitch or set inverts into itself, and the other pitches invert likewise. Inverting around a major third is to invert the two pitches of this interval into one another. The major third in an E-major triad is between E and G sharp. I_0 inverts these two pitches into each other; it also inverts B into C sharp.

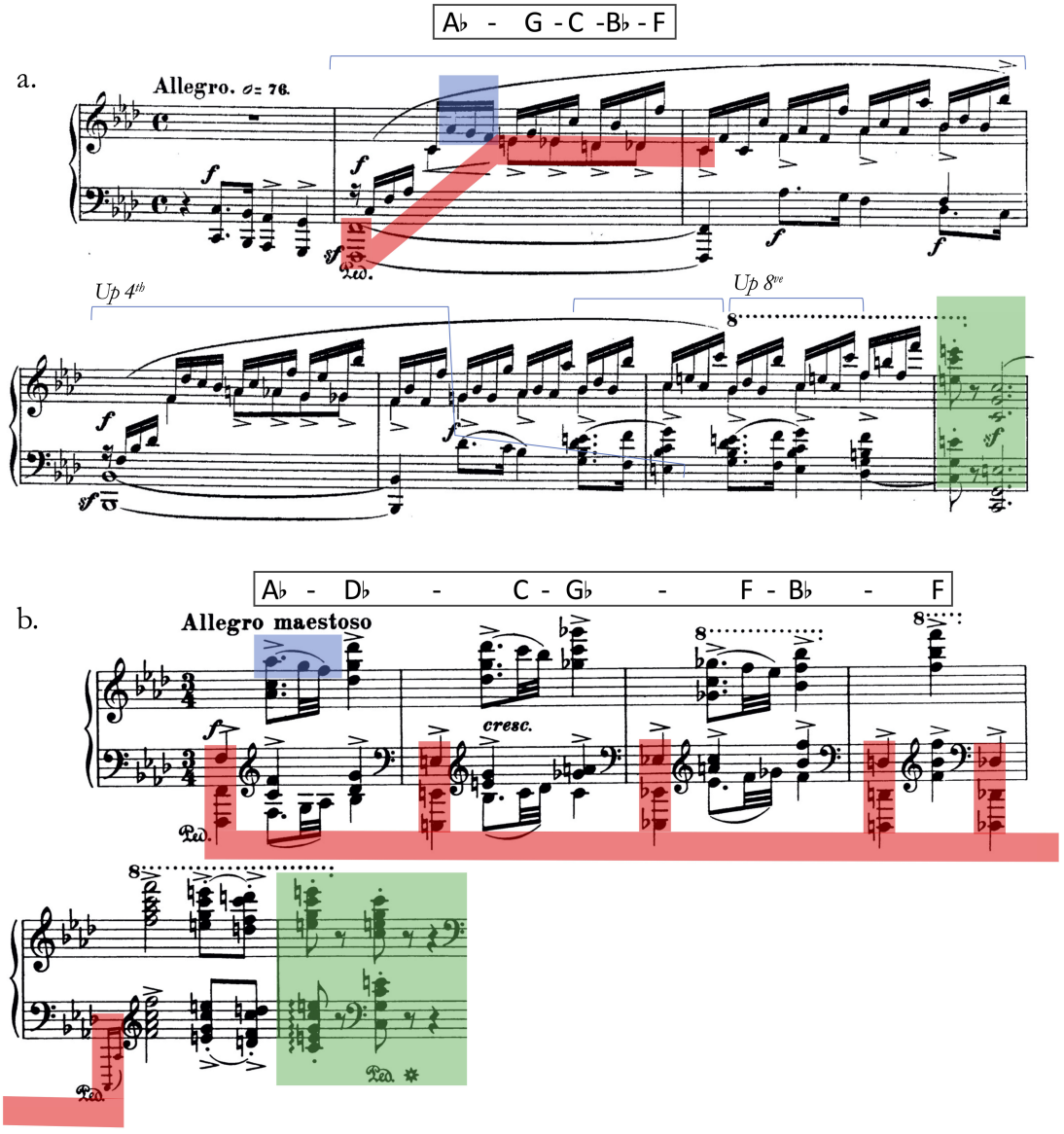


Figure 1: *(a) Robert Schumann, Piano Sonata No. 3 in F Minor, op. 14, mm. 1–7 (b) Johannes Brahms, Piano Sonata No. 3 in F Minor, op. 5, mm. 1–6 (coloration indicates correspondences put forward in [6]).*

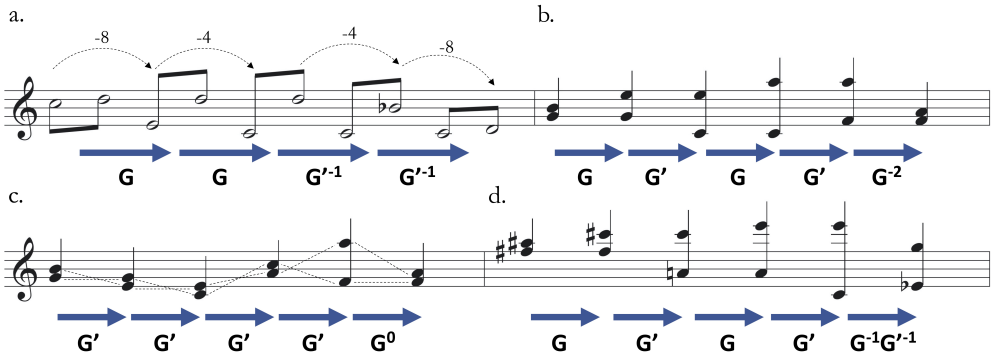


Figure 2: *Demonstrations of* **G***: (a) temporal ordering <first, second>, chromatic or* WT_0 *scale, registral pitch (b) registral ordering <top, bottom>, white-note scale, registral pitch (c) prime-form ordering <root, not root> (à la Straus [20]), white-note scale, pitch class (d) registral ordering <top, bottom>,* $OCT_{0,1}$ *scale, registral pitch.*

G can be defined for any scale and any kind of ordering, act on any interval class, and involve pitch classes rather than registral pitches. If the inputs and outputs are pitch classes, the second and third parts of the definition provided above are immaterial. Following convention, compounds of **G** and **G'** are indicated with positive superscripts (e.g. **G**•**G** = $\mathbf{G}^2$, **G'**•**G'**•**G'** = $\mathbf{G'}^3$, the zero superscript ($\mathbf{G}^0$) indicates the identity transformation, and the inverse of **G** or **G'** is indicated with negative unit superscripts: $\mathbf{G}^{-1}$ and $\mathbf{G'}^{-1}$. When they act on registral pitches, $\mathbf{G}^{-1}$ or $\mathbf{G'}^{-1}$ shrink the interval between them; the inverse does not exist if and only if **G** acts on registral pitches and these registral pitches span the minimal distance permitted by the interval class they span within the governing scale.

Figure 2 provides some examples of **G** in action. The first dyad in Figure 2a is <C5, D5>, ordered in time. The **G** transform of this dyad changes only the first pitch (**G'** would change the second pitch), and lowers it in particular so that the distance between the two pitches will increase. This first pitch is lowered precisely to E4, because this is the one pitch both below C5 and closest to C5 that, when combined with D5, forms an unordered dyad that is a pitch-class scalar transposition of C5, D5 within, say, the chromatic scale or the whole-tone scale with C (WT_0). Therefore, **G**<C5,D5> = <E4,D5>. If **G** were applied to the <C5, D5> dyad within, say, the 2- or 3-flat diatonic scale, or the octatonic scale with C and D ($OCT_{2,3}$) instead, then G<C5,D5> = <Eb4,D5>. Although <C5,D5> and <Eb4,D5> as unordered sets are not chromatic pitch-class transpositions of one another, they are nonetheless pitch-class transpositions of one another within these other scalar contexts.

Figure 2a continues by using the output <E4,D5> as the input for another **G** transformation, which yields <C4, D5>, again within the chromatic scale or WT_0. Curiously, the moving pitch drops only four semitones (E4 down to C4) under the second **G**, whereas it dropped eight semitones (C5 down to E4) under the first **G**. This curiosity, highlighted with curved arrows and well manifested in both the two back-to-back **G** transformations that open Figure 2a as well as the two back-to-back $\mathbf{G'}^{-1}$ transformations that follow, illustrates a special feature of contextual transformations: homogeneity of contextual transformation has the potential to be realized as heterogeny of noncontextual transformation. In published transformational theory, this feature is most evident in neo-Riemannian theory's contextual transpositions. For example, **LP** is a contextual transposition that transposes a major triad by $\mathbf{T_4}$ or "up a major third" and a minor triad by $\mathbf{T_8}$ or "down a major third." Lewin [14] used **LP** to equate two different sounding harmonic

progressions in two themes from Richard Wagner's *Das Rheingold*: "Tarnhelm" (**LP**{G♯,B,D♯} = {E,G,B}) and "Valhalla" (**LP**{G♭,B♭,D♭} = {B♭,D,F}). In these two progressions, the transpositional size (major third) remains the same, but the direction of transposition ("up" or "down") changes depending on the contextual structure of the input (major triad or minor triad). In the first two progressions of Figure 2a, the transpositional direction of the first note (down) remains the same, but the size (eight or four semitones) changes depending on the contextual structure of the input (major second or minor seventh). From this point of view that focuses on size and direction, neo-Riemannian contextual transformations and G contextual transformations resemble "duals" of one another, at least colloquially.

Figures 2b, 2c, and 2d furnish further demonstrations, varying type of ordering, scale, and attention or inattention to octave equivalence. The ends of Figures 2b and 2c show how a **G** transformation with an even superscript outputs a dyad with the same ordered pitch-class content as the input. Figure 2c's dyadic series has the same pitch-class content as that of Figure 2b, even though Figure 2c simply iterates **G'** while Figure 2b alternates between **G** and **G'**. Figure 2c's sleight-of-hand owes to the use of prime form (e.g. [03]) as the dyad's ordering (similar to [20]), whereby, in an ordered pair <x, y>, x corresponds to the first digit (the [0...]) of the dyad's prime form—-what might be called the "root" ([10])—-and y corresponds to the second digit (the [...n]) of the dyad's prime form. Some may see this as analytical overkill, saying instead that Figure 5 simply displays descending-third transpositions, beginning as parallel motion through register and continuing by departing from such. However, the use of **G'** suggests a certain type of voice leading, annotated with straight and mostly crossed lines in Figure 2c, that declines a parallel hearing in favor of another perhaps more covert, in the manner similar to how Lewin [14] approached some instances of parallel voice leading in Debussy's music.

III. G as Variation Mechanism in Some Music of Brahms

A configuration of **G** that is especially suitable for tonal music is a **G** that acts within some diatonic scale—-or more generally, on the seven diatonic letters irrespective of accidental—-on concurrent pitches or pitch classes in outer voices, with an arbitrary ordering of <soprano, bass>. This emphasis on outer voices is consistent with other prominent theories of tonal musical structure ([18], [8]). For tonal music more associated with "high" styles, a good choice for the one interval class spanned by these outer-voice pitches is the diatonic interval class of two steps (dic2), more commonly known as imperfect intervals: thirds, sixths, tenths, thirteenths, and so forth. This interval class tends to be used more often between simultaneous outer-voice pitches in these styles than each of the other three diatonic interval classes (unison/octave (dic0), second/seventh (dic1), fourth/fifth (dic3)) except at beginnings and endings of formal units like phrases and sections, where dic3 and dic0 are more common.

The transformation **G** works well as a variation mechanism in general because it preserves one pitch while changing the other, a balance of mutability and immutability that is at the heart of variation technique. It works well as a variation mechanism more specifically in two capacities: addition of pitches, and alteration of pitches. The first capacity supplies embellishments through applying **G** *syntagmatically* to one dyad to create the next dyad, as Figure 2 does. When **G** is configured as outlined above, this converts a one-to-one first-species texture into a many-to-one second-(or third-)species texture: a pitch of one of the outer voices remains fixed as a "cantus-firmus" note, and the paired pitch from the other voice shifts to another pitch idiomatically consonant with the fixed pitch. The second capacity makes changes through applying **G** *paradigmatically* to one dyad to create the corresponding dyad in the next variational rotation, as Figure 2 does not. For example, if the variation is over a ground bass, each application

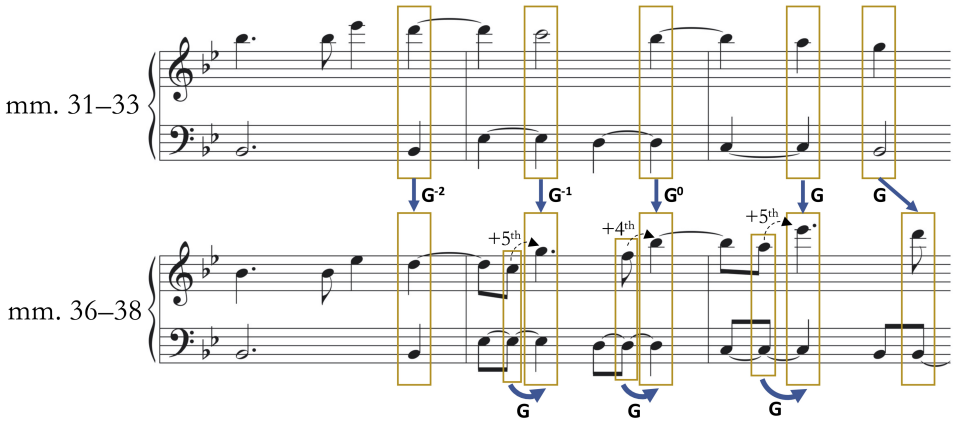


Figure 3: *Gradated application of* **G** *as variation technique: Brahms, Variations on a Theme of Haydn, op. 56, Finale, mm. 31—33 and 36-–38, soprano and bass lines only (registral ordering <top, bottom>, two-flat diatonic collection, registral pitch).*

of **G** would maintain a pitch in this fixed bass line while modifying the treble pitch above it (*mutatis mutandis*, consistent employment of **G′** works well for a reharmonizing variation of a fixed top line).

Figure 3 shown an instance of **G** operating in both paradigmatic and syntagmatic capacities. These excerpts come from the finale of Brahms's *Variations on a Theme of Haydn* of 1873, which is structured as a set of continuous variations on a repeating five-measure line in B-flat major (B♭-B♭-E♭-D-C-B♭-E♭-C-F-F) assembled from the outer voices of the theme's first phrase. This line first appears in the bass during the finale's first 65 measures, and moves to the treble for mm. 61-80 before returning to the bass in m. 81 soon before a coda in mm. 86–109. Measure 31 begins the seventh ground-bass variation, whose outer voices of its first three of five measures are shown in the top system of Figure 3. Here, a high treble line unhurriedly descends, mostly in parallel with the stepwise descending bass. Its syncopations create suspensions that decorate a series of imperfect harmonies above the bass line, enclosed in rectangles, that are displaced onto off beats in this cut-time music. By defining **G** as specified in this section, and additionally as acting on registral pitches, a succession of **G**-based transformations—shown in the middle of Figure 3-—paradigmatically converts the sixth variation's gently descending treble line of D-C-B♭-A… into the seventh variation's bouncily ascending treble line of D-G-B♭-E♭…, reduced in the lower grand staff of Figure 3. More specifically, this succession of transformations is gradated: the **G**-compound superscripts begin with their lowest value of -2, setting the treble line an octave lower, and incrementally increase through -1, 0, and 1, ultimately superseding the treble's pitch height in the earlier corresponding spot. Considering the seventh variation on its own, Brahms realizes this gradation syntagmatically with three distinct **G**s that fuel each embellishing leap in the melody, shown on the bottom of Figure 3. As another case of contextual-transformational heterogeny, these three leaps alternate between the size of a fifth and a fourth, indicated again with curved dashed arrows.

This same four-stage gradated succession of G-compounds undergirds another variation in a work Brahms completed three years after his *Haydn Variations*. The secondary theme in the finale of Brahms's first symphony begins as a set of continuous variations over a looped four-quarter-note bass line, first C-B-A-G in G major in the movement's exposition, and then F-E-D-C in C major in the movement's tonal resolution. The scale degrees ($\hat{4}$-$\hat{3}$-$\hat{2}$-$\hat{1}$) match those of the E♭-D-C-B♭

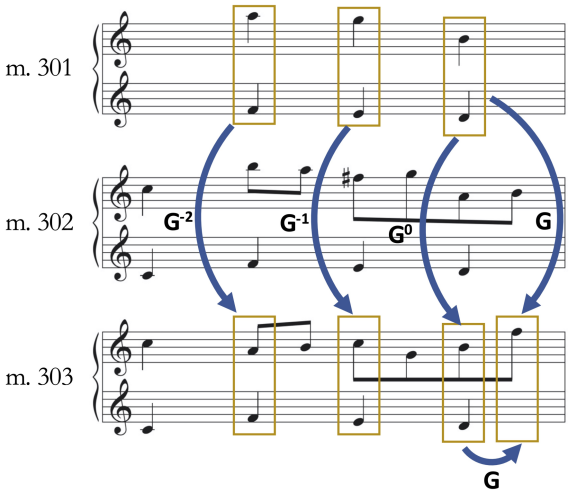


Figure 4: *Gradated application of* ***G*** *as variation technique: Brahms, Symphony No. 1, op. 68, iv, mm. 301—303, soprano and simplified bass lines only (registral ordering <top, bottom>, white-note scale, registral pitch).*

portion of the *Haydn Variations*'s ground bass upon which the **G** transformations were shown to act. Figure 4 transcribes and simplifies the outer voices for the beginning of the C-major version of this secondary theme. Measure 301 offers a first-species opening: two parallel tenths and a sixth that leads to a quasi-cadential arrival on an octave on the next downbeat, an interval that remains outside of **G**'s purview as defined here. The treble of m. 302 decorates the treble pitches of m. 301 with non-harmonic stepwise embellishments also outside of **G**'s purview. However, m. 303 relates to m. 301 with the same series of $\mathbf{G}^{-2}$, $\mathbf{G}^{-1}$, $\mathbf{G}^{0}$, and $\mathbf{G}^{(1)}$ transformations that was deployed for the analysis of the music in Figure 3: here, all four transformations are paradigmatic, and the last is also syntagmatic. As before, the result is a rising treble that starts an octave below the first pitch of its antecedent, and surpasses the antecedent by its end. While the line is significantly different, the preference for imperfect harmony before the cadence is preserved.

An outer-voice **G** that acts on pitch classes instead of registral pitches affords another perspective on more of this theme. In general, iterative applications of such a **G** on a dyad toggles its first (or, if **G'**, second) member between two states, which correspond to the two ordered-pitch-class-interval members of a non-zero interval class. For example, there are two diatonic pitch classes that form an imperfect interval above F: A and D. Therefore, **G**<A,F> = <D,F>, and **G**<D,F> = <A,F>. Such a **G** involutes, or, in other words, $\mathbf{G}^{-1} = \mathbf{G}$, and, more generally, $\mathbf{G}^{n} = \mathbf{G}^{n+2t}$, where $t, n \in \mathbb{Z}$. Figure 5 discloses this toggling in the theme's first nine measures, a span of time that comes full circle as m. 309 repeats m. 302, which ornaments the same consonant pitches from m. 301. The regions that are shaded yellow and purple keep track of the **G**-based toggling for each beat's dyad through these nine measures, where, above the fixed bass note, yellow signifies the starting imperfect harmony as the default and purple signifies the other (from a toggling perspective, antipodal) imperfect harmony. For example, the second beat's switch from the treble's initial A in mm. 301–3 (in yellow) to a contrasting D in mm. 304–6 (in purple) above the fixed bass note F can be indicated as **G**<A,F> = <D,F>. Applying **G** again restores the treble's F back to A for the next three measures: **G**<D,F> = <A,F>. (Figure 5 accommodates pitches outside of the C-major collection by defining **G** to act on the seven notated diatonic letters regardless of their inflections by accidentals.)

Not every outer-voice harmony exclusively expresses an imperfect quality during the second,

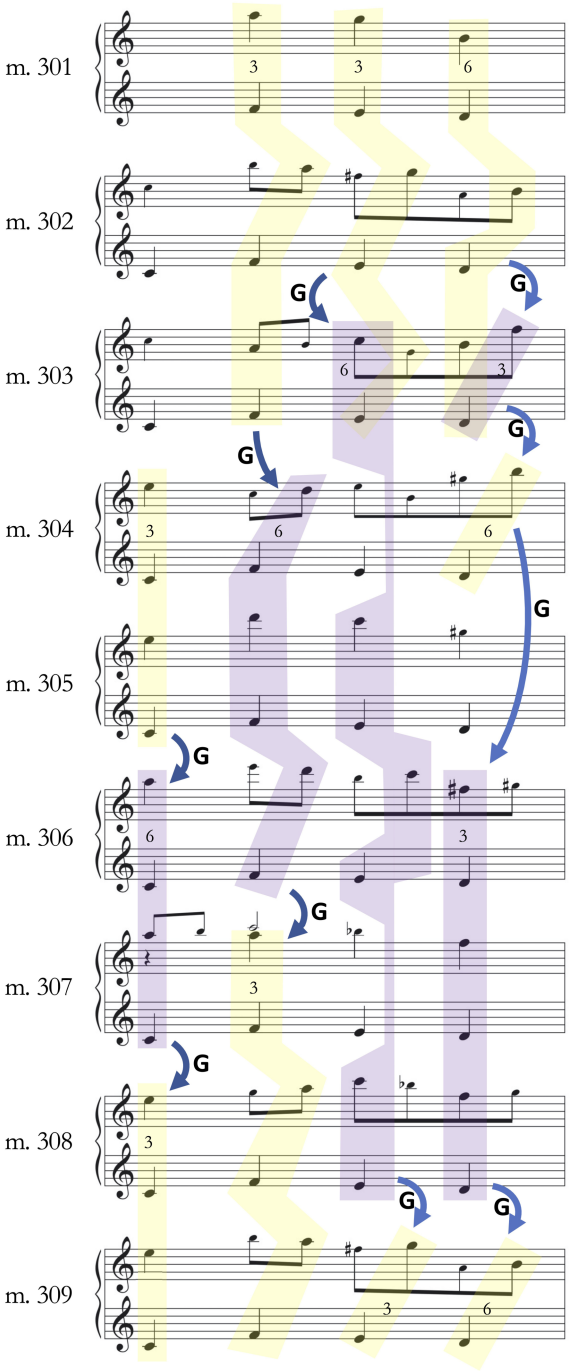


Figure 5: *Toggling application of* ***G*** *as variation technique: Brahms, Symphony No. 1, op. 68, iv, mm. 301—309, soprano and simplified bass lines only (registral ordering <top, bottom>, generalized diatonic scale, pitch class).*

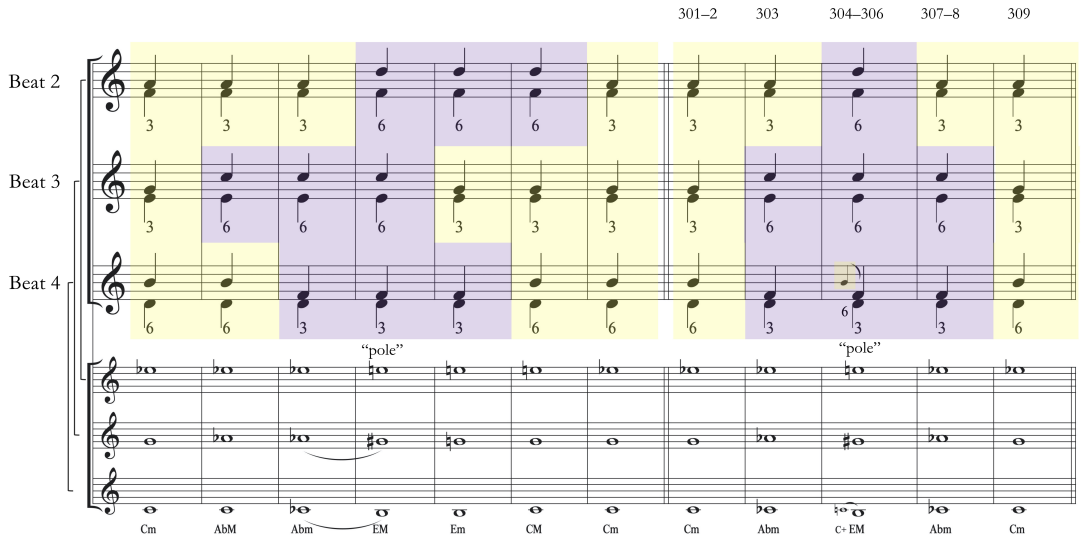


Figure 6: *(a) **G** toggling cycle as analogous to a hexatonic cycle (b) Portion of this cycle used by Brahms, Symphony No. 1, op. 68, iv, mm. 301—309 and its hexatonic analogue.*

third, and fourth beats of this theme; those treble pitches that express a different diatonic interval class with the ground bass receive a smaller font. However, there remains enough imperfect harmony to measure the advance through these nine measures using **G** as the unit of distance and energy. Each beat toggles between its two pitch classes at different times, not only within the measure, which is unavoidable, but also at different times between measures, which is not. The second and third beats toggle once, and the fourth beat toggles twice, before all three beats' pitches return to their default states. Even the treble pitches on the first beat, after they indulge octave quasi-cadences in the first two measures, join in the toggling, moving above the bass C from E to A in m. 306 and back to E in m. 308. This advance during the last three beats of each measure is analogous to motion along an instance of Richard Cohn's [4] hexatonic cycle, which Michael Siciliano [19] reinterpreted as staggered instances of toggling among three half-step dyads, as shown in the first part (a) of Figure 6, which reuses the default and contrasting coloration of Figure 5. Here, consonant triads are analogous to a mixture of harmonic thirds and sixths, whereas augmented triads are analogous to an exclusive use of one of these harmonic types or the other. However, Brahms's theme, summarized in Figure 6b, does not complete the cycle. Rather, it progresses halfway through the cycle to the completely purple "hexatonic pole" of the initial completely yellow default, which it embellishes with an "augmented triad"—-exclusively sixths during mm. 304–5, shown with grace notes in Figure 6b—-and then retraces its steps back to the default. This pole coincides with the transitory tonicization of A minor, the relative minor of the main key of C major.

In addition to serving as analyses in their own right, Figures 3–6 serve as documentation that **G** can model some of Brahms's variations, and therefore help to corroborate a claim that, if **G**, especially a gradated **G**, relates the opening of Schumann's op.14 to Brahms's op. 5, then one can think of the latter as a variation of the former. However, both the *Haydn Variations* and the First Symphony came after 1853: does **G** model variations before this date? One type of an eighteenth-century alteration modeled by **G'** (or occasionally $\mathbf{G'}^{-1}$), although typically governed by some other form besides variation, emerges when the composer presents two formally corresponding fixed stepwise rising treble motives, usually $\hat{3}$-$\hat{4}$-$\hat{5}$, twice: in one of its appearances (usually the

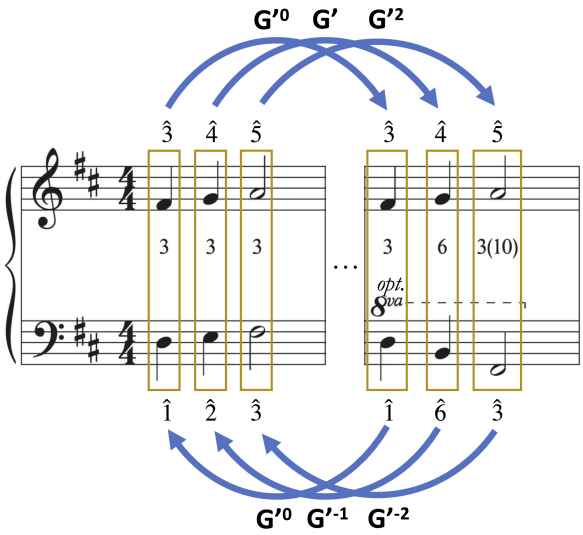


Figure 7: *An eighteenth-century motivic reharmonization using a three-stage gradated succession of* ***G'*** *- or* ***G'***$_{-1}$*-compounds.*

first), the bass harmonizes this motive with 1̂-2̂-3̂ in rising motion parallel to the treble; in the other; the bass harmonizes this motive with 1̂-6̂-3̂ in descending motion contrary to the treble. They may correspond as part of successive parallel basic ideas, phrases, or themes, or even as part of exposition and recapitulation.[4] As shown with an abstracted case of this in Figure 7, a three-stage gradated succession of **G'**–or **G'**$^{-1}$–compounds relates these two moments, depending on which progression begets the other. While this transformation can serve practical benefits, such as connecting a part of this bass line to a higher or lower register, it also offers aesthetic diversity.

Another relevant precedent links this eighteenth-century practice to the next section's **G**-enabled analysis of the opening of Brahms's op. 5. Before visiting the Schumanns, Brahms had just completed his C-major piano sonata, which was published as his op. 1, although it was composed after his F-sharp-minor piano sonata, his second opus. The C-major sonata opens with a basic idea that rises from 3̂ (E) to 6̂ (A) in the treble, harmonized by parallel tenths with the bass. This basic idea is followed by its repetition that apexes one step higher on B flat, lightly tonicizing F Major, again with consistent parallel tenths below. The next two measures function as basic-idea repetition (the treble still rises by step), basic-idea variation (the bass departs from its strict parallel motion), and cadence on the dominant. As shown in Figure 8, **G'** well expresses the bass's transformation from conjunct and generally ascending motion in mm. 1-4 (Figure 8a) to disjunct and generally descending motion in mm. 5-6 and the two pickup notes (Figure 8c). As Figure 8b sets out, the outer voices in mm. 3-4 are transposed up four diatonic steps and the prevailing diatonic collection is shifted two accidentals sharpward from the one flat of F major to the one sharp of G major, indicated using Julian Hook's [11] nomenclature from his study of signature

[4] Examples of this altered repetition of the bass under the 3̂-4̂-5̂ portion of a formal unit include:

- J.S. Bach, Cantata, BWV 140, vi: **G'**(m. 1) = m. 3 (basic idea and its immediate repetition transposed to dominant)
- J. Haydn, Piano Sonata, XVI: 35, i: **G'**$^{-1}$(mm. 40–41) = mm. 130–31 (exposition and recapitulation transposed from dominant)
- W.A. Mozart, Violin Concerto No. 2, K. 211, i: **G'**(m. 7) = m. 9 (basic idea and its immediate repetition)
- W.A. Mozart, Horn Concerto No. 3, K. 447, ii, **G'**(m. 3) = m. 11 (period and its immediate repetition)
- A. Salieri, Organ Concerto, IAS 27, i: **G'**(m. 116) = m. 117 (motive and its immediate repetition)

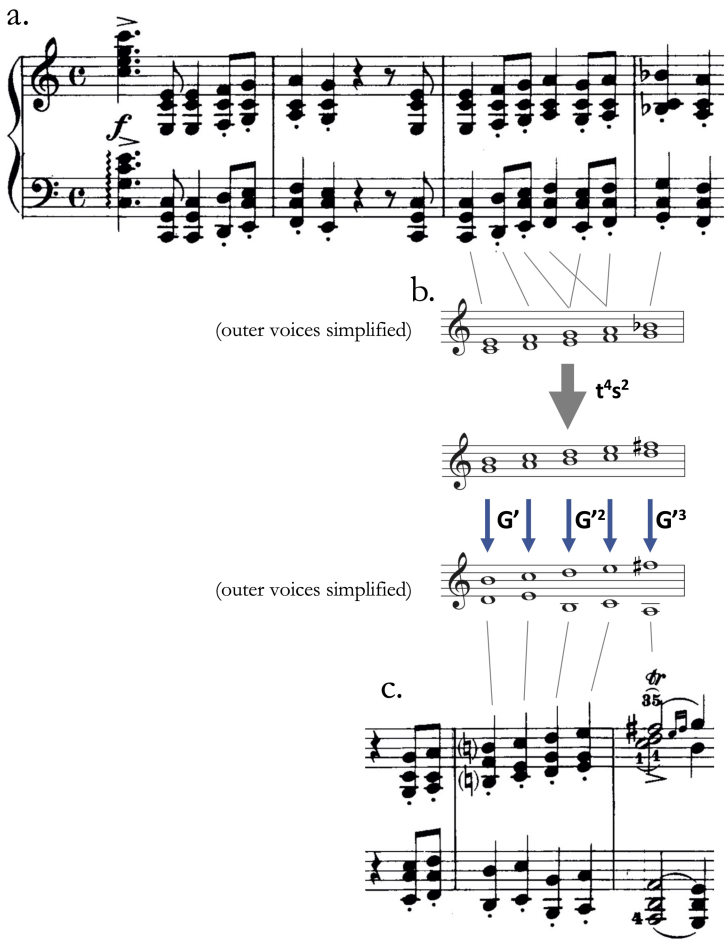


Figure 8: *(a) Brahms, Piano Sonata op. 1, i, mm. 1–4 (b) Figure 8a transformed by signature transformations and gradated* ***G'****(c) Brahms, op. 1, mm. 4–6.*

transformations. Then a three-stage gradated succession of **G'**-compounds, but with the first two stages operating on pairs of dyads, turns the outer voices from their incremental, lockstepped, and measured opening in mm. 1-4 to their vaulting, expanding, and dramatic conclusion in mm. 5-6.

IV. Schumann's Op. 14 and Brahms's Op. 5

In the second measure of the opening of Schumann's op. 14 piano sonata (Figure 1a), the local, mostly up-stemmed, maxima of the right-hand pitches—the jagged line A♭-G-C-B♭-F—bears the characteristic heterogenous stairstep design of a gradated **G**-series transform of parallel imperfect intervals (compare this treble line to that of mm. 36–38 of Figure 3). The left side of Figure 9 (a-c) shows the generation of this design from a succession of parallel thirds as a proposed precursory "deep structure." This is a mirrored form of the transformation from parallel thirds to an expanding wedge in the opening measures of Brahms's op. 1 in Figure 8, reprinted on the right side of Figure 9 (d, e) with corresponding parts connected with dashed lines for ease of comparison: Schumann's **G**-varied jagged line goes up in the treble, and Brahms's **G'**-varied

jagged line goes down in the bass.[5] If the pattern in Schumann's second measure were to continue, a C in the bass and an E in the treble would appear on the downbeat of the third measure, an expectation that could be made more palpable if the performer would indulge even the slightest of ritardandos at the end of the second measure. As visualized with the left-pointing arrows and "no" symbol in Figure 9, this dyad does not materialize here despite this expected continuation. Yet it does appear on the downbeat of m. 6 as the culmination of the opening phrase albeit with the E two octaves higher. Schumann gets to this climactic cadence via transposed repetitions, shown in Figure 1 with brackets and italic labels: most of mm. 2–3 transpose up a fourth to make most of mm. 4–5, and two beats spanning the downbeat of m. 6 transpose up an octave to make the next two beats.

In comparing the first six measures of Brahms F-minor piano sonata to the first seven measures of Schumann's F-minor piano sonata, I hear a rerouting of the path from the starting dyad <A♭,F> to the finishing dyad <E,C>.[6] Whereas Schumann maneuvered this course with the aforementioned extensions, Brahms does so through expansions: specifically, an expansion of Schumann's second measure, aligning its <E,C> continuation, unrealized on the downbeat of m. 3, with its realization four measures later. This expansion is most clearly accomplished in the bass, as the chromatic descent F-E-E♭-D-D♭-C is stretched out from Schumann's single measure to Brahms's five measures. But a rerouted expansion needs to involve not only time but also register, especially in the treble, which in Schumann's second measure starts at A♭4 and rises around an octave to prepare for the unrealized E5, but then rises around two more octaves in the next four measures to peak at E7. How does Brahms's expansion of Schumann's treble make up for this two-octave difference?

One half of the solution is simply to move the melody's initial A♭4 up to A♭5, which is Brahms's first treble pitch. The other half of the solution is, using the apparatus developed in this study, to apply a three-stage series of **G**-compounds to Schumann's second measure. Figure 10 presents a detailed schematic of this application, reusing coloration from Figures 1, 5, and 6. The outer voices have been simplified, with the treble in particular down an octave in the Brahms portions of Figures 10c and 10d. Figures 10a and 10b inspects the left portion of Figure 9 from a slightly different angle. Figure 10a shows a "deep structure" of parallel thirds over a lament bass: as Figure 10's **G** operates on the seven diatonic pitch names without reference to a specific key, this "deep" bass line could be F-E-D-C, F-E♭-D♭-C, or F-E-E♭-D-D♭-C. Figure 10b performs a three-stage series of **G**-compounds to this "deep structure," and superimposes the two-voice design onto Schumann's first seven measures, highlighting the four-measure gap. The horizontal gray arrows marked **G** add a syntagmatic component to the interpretation, rather than the purely paradigmatic replacements of Figure 9, and the diagonal lines show the shift to the next stage, which always take place during a syntagmatic embellishment, at least of a diatonic framework.

The E at the end of Figure 10b is still one octave shy of the treble's cadential goal. To get there, one might imagine that Brahms applies a three-stage series of **G**-compounds ($\mathbf{G}^0$, **G**, $\mathbf{G}^2$) to Schumann's second measure, varying Schumann's music in a manner that I have shown him to

[5] Here, "mirror" counterpoint refers to two lines that both inverted as individual lines, and swap positions in register, as if the notation of the two-voice construction has been reflected in a horizontal mirror.

[6] Instead of rerouting, one could apply the metaphor of repair, in that Brahms is "fixing" Schumann's detachment between the second measure's generation of an expectation of <E,C> to immediately follow the second measure, and <E,C>'s eventual appearance, displaced by four measures and two octaves. Following Harold Bloom's [2] theory of poetic influence, this metaphor would hypothesize that Brahms's response to Schumann's music is agonistic to some degree, that Brahms commits misprision and "misreads" Schumann's opening. Of Bloom's revisionary ratios, this misreading comes closest to what Bloom calls a *tessera* or fragment, such that Brahms's solution is "to retain its terms but to mean them in another sense, as though the precursor had failed to go far enough." [2, p. 14]. This approach is also consistent with Cohn's [5] interpretation of the opening of Brahms's op. 5 as itself involving "cracked and mended hemiolas," parenthetical insertions that replace one form of well-formedness with another.

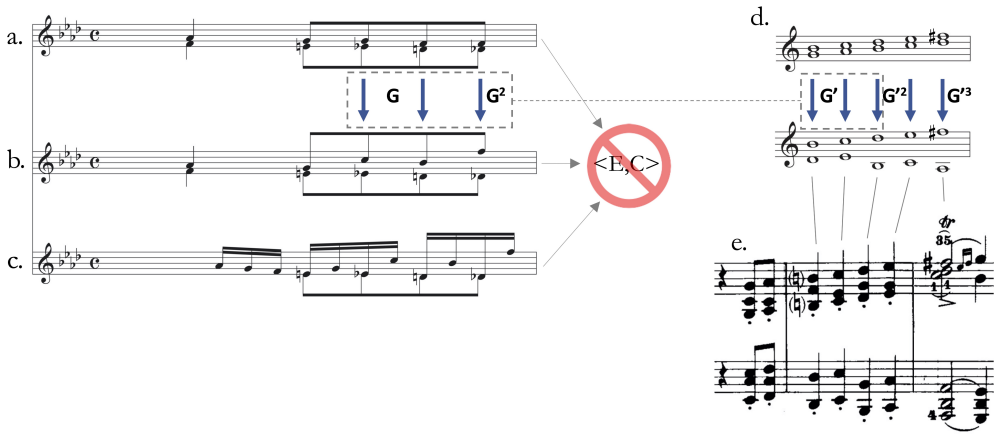


Figure 9: *(a) Chromatic lament bass harmonized in parallel thirds (b) Figure 9a transformed by gradated* ***G*** *(c) Schumann, op. 14, m. 2 (an unfolding of Figure 9b) (d) Figure 8a transformed by signature transformations and gradated* ***G*** *(e) Brahms, op. 1, mm. 4–6.*

use both before and after the fall of 1853, and producing the music of Figure 10c from the music of Figure 10b. I particularly value this way of relating these two passages, because it puts forward the notion that the hypothetical series of transformations that Schumann applied to a "deep" structure of purely stepwise parallel thirds to produce the contrary and stairstepping "surface" of the second measure of his F-minor sonata ($\mathbf{G}^0$, $\mathbf{G}$, $\mathbf{G}^2$) is the same series of transformations that Brahms applies to Schumann's "surface" to create the even steeper stairstepping "supersurface" of the first six measures of his F-minor sonata. Moreover, the link between "deep" and "surface" is more than hypothetical for Brahms: he compositionally realized this link in the first six measures of the piano sonata he had just completed months earlier, albeit with the stairsteps in the bass instead of the treble. The steeper ascent in the treble perfectly bridges the second octave, achieving the high E7 (again, shown down an octave at the end of Figure 10c and 10d for ease of reading).

One difference between the transformation from "deep" to "surface," and from "surface" to "supersurface," is the pacing by which the output moves through the three stages. The distribution of $\mathbf{G}^0$, $\mathbf{G}$, and $\mathbf{G}^2$ in the former is fairly regular with two dyads for each stage, as can be seen reading the transformations from left to right in between Figure 10a and 10b. However, for the latter—in between Figure 10b and 10c–the second stage of **G** predominates. Brahms's syntagmatic embellishment of <A♭,F> to <D♭,F> in the first measure passes the variation immediately into its second stage, in which it stays for five dyads, each a **G** transform of the rest of Schumann's second measure. Another syntagmatic embellishment of the final dyad—-<B♭,D♭> to <F,D♭>-—of these five commences the third and final stage right before the <E,C> conclusion. The result is a treble line whose pitch-class content only matches that of the very beginning and ending of Schumann's second measure and seventh downbeat, as shown with the yellow-as-default and purple-as-contrasting coloration employed earlier. All of the other treble pitches are different, which considerably obscures their **G**-enabled kinship.

These two syntagmatic additions to Schumann's treble content carry metric and rhythmic ramifications. Figure 10d summarizes how the pitches within the consonant framework of Figure 10c relocate into their final temporal and motivic positions; orange arrows indicate these displacements. Schumann's second measure expresses duple metrical relations on two levels: the even alternation of bass-then-treble within an isochronous compound melody puts a sixteenth pulse within an eighth pulse, and assigning two dyads per **G**-incremental stage puts this eighth

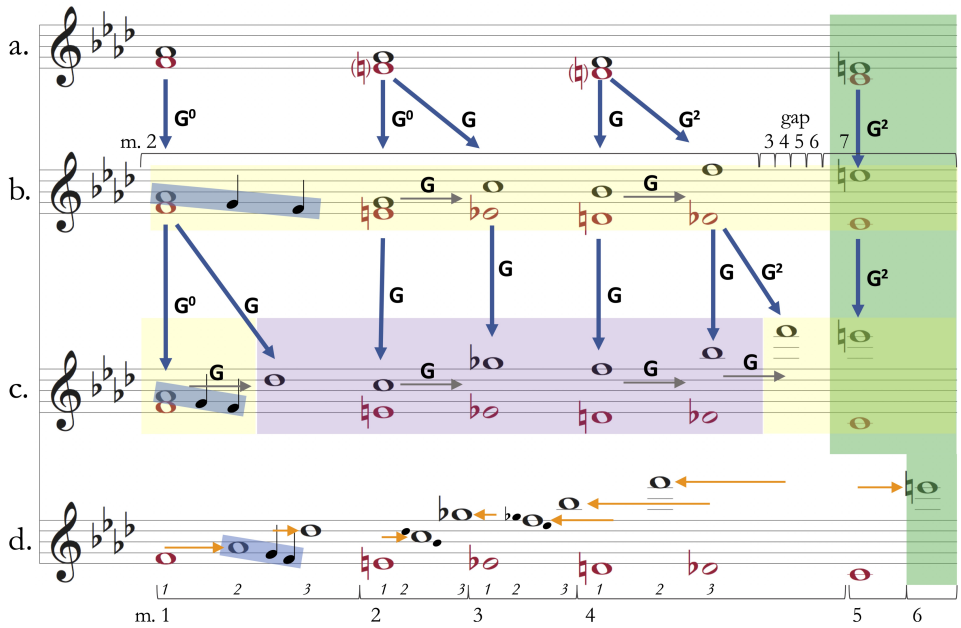


Figure 10: *(a) Lament bass harmonized in parallel thirds (b) Figure 5a transformed by gradated **G** to match Schumann, op. 14, i, m. 2 (compare to Figure 4b and Figure 4c) (c) Figure 5b transformed by gradated G (d) reduction of Brahms, Piano Sonata op. 5, i, mm. 1–6 (displacement of Figure 5c) (color scheme corresponds to that of Figure 1).*

pulse within a quarter pulse. However, the initial syntagmatic embellishment of <A♭,F> to <D♭,F> increments the number of pitch events during the first harmony from two to three—bass F, treble A♭, treble D♭-—which matches the younger composer's new meter of 3/4. Brahms maintains this same voice-beat assignment—bass-one, treble-two, treble-three—in the next two measures. Since this pair of measures remains in the second stage, where Schumann's pitches are only replaced rather than increased in number, Brahms must pull treble pitches earlier in time to meet each measure's two-treble-pitch quota, portrayed with left-pointing orange arrows. The resulting implicit bass suspensions create a powerful outer-voice torsion, especially in the context of the implicit treble D♭ suspension over the bass E on the downbeat of m. 2. The second and last-second syntagmatic embellishment of <B♭,D♭> to <F,D♭> furnishes an extra treble pitch, but three more would be needed to maintain the current voice-beat assignment: two treble pitches each for the bass pitches D and D♭. To square the uneven distribution, Brahms's fourth measure switches to a bass-one, treble-two, bass-three voice-beat assignment—bass D, treble F, bass D♭-—which also idiomatically accelerates the harmonic rhythm before the cadence.

V. Conclusions

This article presents one way in which mathematically-based music-theoretical tools may be mobilized to substantiate or qualify music-historical claims. Although this study involves a specific transformation as the tool and a specific intertextual influence in the claim, I suspect that other crossover opportunities await, not only through sophisticated statistical analyses of Big (Musical) Data (e.g. [9]), but also through close readings of individual works such as the one submitted herein. This article also endorses a transformational understanding of musical variation,

a perspective especially explored by Carlos Almada (e.g. [1]) but also one that I suspect is far from exhausted in its scholarly and pedagogical utility.

References

[1] Almada, Carlos (2019). Variation and Developing Variation under a Transformational Perspective. *Musica Theorica*, v. 4, n. 1, pp. 30—61.

[2] Bloom, Harold (1997). *The Anxiety of Influence*: A Theory of Poetry. Second Edition. New York: Oxford University Press.

[3] Bozarth, George (1990). Brahms's Lieder ohne Worte: The 'Poetic' Andantes of the Piano Sonatas. In: George S. Bozarth (ed.). *Brahms Studies*: Analytical and Historical Perspectives, pp. 345—378. Oxford: Clarendon Press.

[4] Cohn, Richard (1996). Maximally Smooth Cycles, Hexatonic Systems, and the Analysis of Late-Romantic Triadic Progressions. *Music Analysis*, v. 15, n. 1, pp. 9-–40.

[5] Cohn, Richard (2018). Brahms at Twenty: Hemiolic Varietals and Metric Malleability in an Early Sonata. In: Scott Murphy (ed.). *Brahms and the Shaping of Time*, pp. 178—204. Rochester: University of Rochester Press.

[6] Ehlert, Gero (2005). *Architektonik der Leidenschaften*: Eine Studie zu den Klaviersonaten von Johannes Brahms. Kassel: Bärenreiter.

[7] Eich, Katrin (2009). Die Klavierwerke. In: Wolfgang Sandberger (ed.). *Brahms Handbuch*, pp. 332—369. Stuttgart and Weimar: Metzler.

[8] Gjerdingen, Robert (2007). *Music in the Galant Style*. Oxford and New York: Oxford University Press.

[9] Hansen, Niels; Sadakata, Makiko; Pearce, Marcus (2016). Nonlinear Changes in the Rhythm of European Art Music: Quantitative Support for Historical Musicology. *Music Perception*, v. 33, n. 4, pp. 414-–431.

[10] Hindemith, P. (1942). *The Craft of Musical Composition*. New York, Associated Music Publishers; London, Schott & Co.

[11] Hook, Julian (2008). Signature Transformations. In: Jack Douthett, Martha M. Hyde, and Charles J. Smith (ed.). *Music Theory and Mathematics*: Chords, Collections, and Transformations, pp. 137—60. Rochester: University of Rochester Press.

[12] Hyer, Brian (1995). Reimag(in)ing Riemann. *Journal of Music Theory*, v. 39, n. 1, pp. 101—138.

[13] Lambert, Philip (2000). On Contextual Transformations. *Perspectives of New Music*, v. 38, n. 1, pp. 45—76.

[14] Lewin, David (1987). *Generalized Musical Intervals and Transformations*. New Haven: Yale University Press.

[15] Lewin, David (1987). Some Instances of Parallel Voice-Leading in Debussy. *19th-Century Music*, v. 11, n. 1, pp. 59-–72.

[16] Lewin, David (1993). *Musical Form and Transformation*: Four Analytic Essays. New Haven: Yale University Press.

[17] Macdonald, Hugh (2012). *Music in 1853*: The Biography of a Year. Woodbridge: The Boydell Press.

[18] Schenker, Heinrich (1979). *Free Composition*. Translated and edited by Ernst Oster. New York: Longman.

[19] Siciliano, Michael (2005). Toggling Cycles, Hexatonic Systems, and Some Analysis of Early Atonal Music. *Music Theory Spectrum*, v. 27, n. 2, pp. 221-–247.

[20] Straus, Joseph (2011). Contextual-Inversion Spaces. *Journal of Music Theory*, v. 55, n. 1, pp. 43–88.

[21] Tymoczko, Dmitri (2011). *A Geometry of Music*: Harmony and Counterpoint in the Extended Common Practice. Oxford and New York: Oxford University Press.

Python Scripts for Rhythmic Partitioning Analysis

*Marcos da Silva Sampaio
Universidade Federal da Bahia
sampaio.marcos@ufba.br
Orcid: 0000-0001-8029-769X

Pauxy Gentil-Nunes
Universidade Federal do Rio de Janeiro
pauxygnunes@musica.ufrj.br
Orcid: 0000-0001-6810-9609

DOI: 10.46926/musmat.2022v6n2.17-55

Abstract: *The* Rhythmic Partitioning Analysis *demands laborious tasks on segmentation and agglomeration/dispersion calculus. Parsemat software runs these tasks and renders indexogram and partitiogram charts. In the present paper, we introduce the Rhythmic Partitioning Scripts (RP Scripts) as an application of Rhythmic Partitioning in the Python environment. It adds some features absent in Parsemat, such as the access to measure indications of each partition, introduction of rest handling, annotation of texture info into digital scores, and other improvements. The RP Scripts collect musical events' locations and output locations and partitions' data into CSV files, render indexogram/partitiogram charts, and generate annotated MusicXML score files. RP Scripts have three components: calculator (RPC), plotter (RPP), and annotator (RPA) scripts.*

Keywords*: Rhythmic Partitioning Analysis. Textural Analysis. Music Analysis. Python scripts. Music21.*

I. Introduction

Parsemat software [12], developed by Pauxy Gentil-Nunes, assists in *Rhythmic Partitioning Analysis* of musical texture. Despite being crucial for studying the Partitional Analysis of musical texture, it lacks events' location in terms of bar numbers and measure positions, as well as rest handling. This absence impairs the identification and location of musical events in the analysis process.

*Thanks for Fapesb and UFBA.

Received: October 28th, 2022
Approved: December 9th, 2022

In the present paper, we introduce the *Rhythmic Partitioning Scripts* (or RP Scripts) to fill these gaps. These scripts output partitions data with the bars' location and in-measure position location, render partitiogram and indexogram charts, and annotate the partitions information into the given digital score.

RP Scripts are composed of the *Rhythmic Partitioning Calculator* script (RPC), *Rhythmic Partitioning Plotter* script (RPP), and *Rhythmic Partitioning Annotator* script (RPA). These scripts, written in Python [27], take advantage of the features of Music21 [6], Pandas [28], Matplotlib [19], and CSV libraries, allowing the use of Kern [25] and MusicXML [15] digital scores as input and CSV, SVG, PNG, and JPG files as output.[1] Thus, in this paper, we review the Rhythmic Partitioning Theory and Parsemat, present *RP Scripts* and introduce a short analysis of three pieces from Music21's corpus [5] to illustrate the data usage.

II. Partitional Analysis and Rhythmic Partitioning

Musical texture is understood here as the interaction between constituent parts of a musical plot.[2] It is a critical task in contemporary musical analysis. In this field, the pioneering work of Wallace Berry [4] inspired several researchers to develop models to describe the relationships and transformations between textural configurations of musical pieces, especially in the context of concert music [16, 1]. *Partitional Analysis* (henceforth, PA [13, 10, 11]) is one of the texture formalization initiatives developed through the mediation between Berry's work and the Theory of Integer Partitions [2, 3].

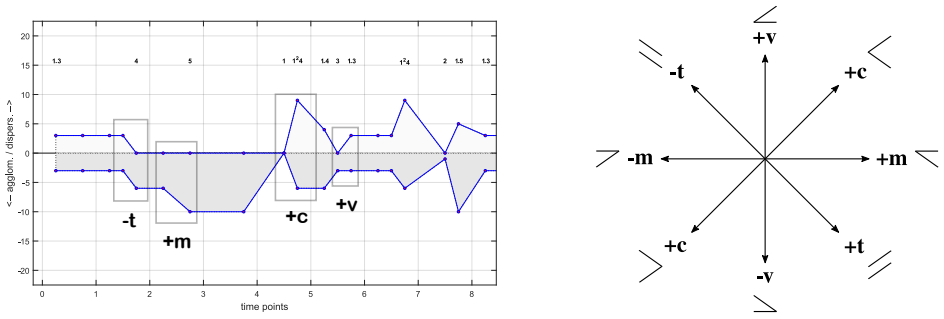
Partitions are representations of integers by the sum of other integers. Since each integer has a finite set of partitions, it is possible to establish an exhaustive taxonomy and map their relationships. One can, too, establish a biunivocal correspondence between partitions and textural configurations. The inventory of textural configurations of a given instrumental set is called the *lexical-set* in PA, whose cardinality is called *lexical sum*. For example, a four-part ensemble (like a string quartet or a four-voice choir) has 11 settings in its lexical-set: $L = \{(1), (2), (1+1), (3), (1+2), (1+1+1), (4), (1+3), (2+2), (1+1+1+2), (1+1+1+1)\}$. Each partition corresponds to a mode of grouping and interacting between parts or musicians. Musical works written for these groups can then be read as a continuous linear progression involving these 11 states.

When a part articulates, and others are suspended (as sustained durations arising from previous attacks), the common suspended state is considered as similarity or convergence and counted as an agglomeration relationship. Each configuration, or partition, has a specific degree of homorhythmic texture (that is, parts that articulate together) and polyphony (parts that articulate independently). This characteristic emerges from the qualitative evaluation of the binary relationship (i.e., pairwise assessments) between its elements, separating, on the one hand, the relationships of congruence, collaboration, or similarity and, on the other, the relations of incongruity, opposition, or difference. This count generates the agglomeration and dispersion indices, which form a pair (a, d).

In the case of the texture-plot, the basic grouping criteria are attack points (picked at time-points) and the durations of each note. Other types of partitioning can be defined by different standards, like the structural nature of events [9]; the performative relation between body and instrument [22]; the instrumental sonic resources involved [18]; compositional concepts and techniques [20], among others. Independent of the adopted criteria, partition $(2+2)$ is more

[1] Another implementation of partitioning functions in Python is the module `comp.parsepy`, by Pedro Faria Proença Gomes, a component of his compositional toolset [14]. This initiative is part of his Master's Thesis in press, advised by Dr. Liduino Pitombeira.

[2] Or *texture-plot*, according to Pablo Fessel [8], in opposition to the *texture-sonority*, concerned with the quality of timbre and other esthesic qualities.



(a) *Relations.* **(b)** *Angles schema.*

Figure 1: *Relations between successive partitions expressed by the angles between the correspondent agglomeration and dispersion indices in the indexogram (standard style). Adapted from Gentil-Nunes [10].*

crowded than partition $(1+1+1+1)$, as its parts are more massive and the number of distinct parts is smaller; on the other hand, it is more dispersed than partition (4), the most crowded of the lexical-set of 4. In this sense, there is perfect homology between the global organization of these distinct fields, which gives rise to the possibility of free transduction between them in a more organic and meaningful way than just a series of values (as proposed in Integral Serialism).

Partitional Analysis then constitutes itself as a field of investigation that includes analytical methods (as the assessment of the partitional progressions and structures aroused in graphical outputs, like the *bubbles*[3] or recurrence of indexes patterns), fundamental structures (as the Partitional Young Lattice, Partitiogram, textural classes, textural complexes), creative processes (as the use of partitional operators and taxonomies for evaluating compositional choices and plannings), among other proposals. In addition, PA was used by some researchers and composers to identify compositional signatures or shared textural features between pieces, helping morphological analysis and constructing models for practical musical tasks, like idiomatic writing and performance, orchestration, and voice-leading, among others.

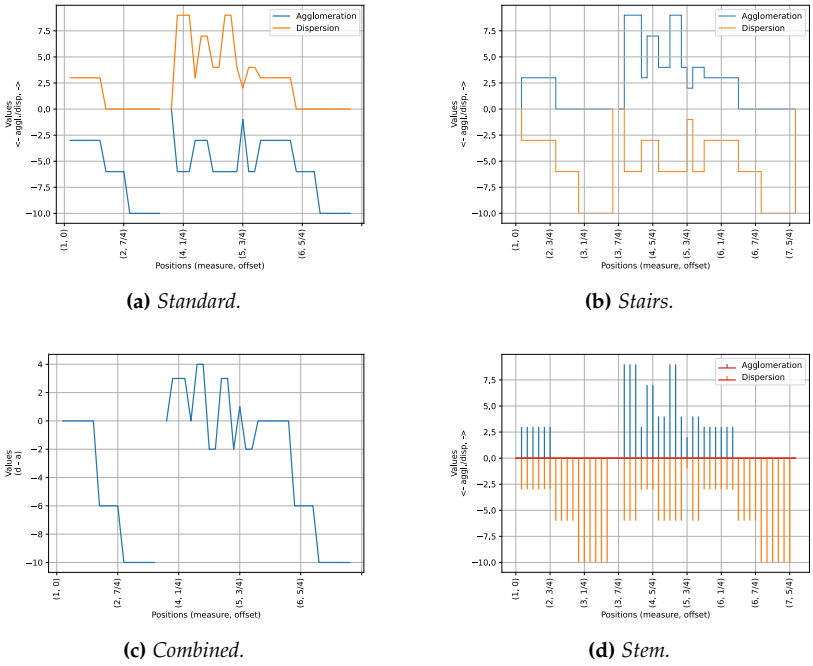
Regarding partitions notation, George Andrews [2] and the mathematicians that work with the Theory of Integer Partitions use to abbreviate them with indexes that express the multiplicity of the parts. When there are successive unique parts, they are separated by dots. For instance, the abbreviated notation of partition $(1+1+2+2+2+3+4)$ is $(1^2 2^3 3.4)$.

III. The indexogram

The indexogram is one of the visualization tools developed in the context of Partitional Analysis. It consists of plotting the agglomeration and dispersion indices in a mirrored arrangement (y-axis), i. e., the agglomeration expressed with a negative sign, relative to a median temporal axis (x-axis). In the standard mode, the patterns formed by the angles of both trajectories are read as one of the four principal relations between partitions [10]: resizing (m), revariance (v), transference (t), and concurrence (c), each one with a positive and negative sign $(+m, -m, +v, -v, +t, -t, +c, -c)$[4] (Figure 1).

[3] See Section III.

[4] The presentation of the relations between partitions is out of the scope of this paper. See Gentil-Nunes [10, 11] for further information.



(a) *Standard.* **(b)** *Stairs.*

(c) *Combined.* **(d)** *Stem.*

Figure 2: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?). Mm. 1–7+3/2. Indexogram types.*

The interaction between trajectories of the (a, d) indices in the indexogram forms broader structures, generally delimited by low values. These structures are called *bubbles* and can be read as significant textural movements responsible for delimiting sections in traditional concert music. Sometimes, these bubbles have recurrent contours, with or without graphical variations and transformation, forming patterns. The assessment of this kind of structure is an analytical method *per se*. However, these recurrences can eventually correspond, in the score, to musical fragments with no rhythmic or pitch similarities, which indicate a kind of textural motivic work that can be hard to detect by a simple glance at the score, justifying the use of the indexogram as a tool for exploring specific textural features.

The data contained in the indexogram is always characterized by the temporal trajectories of the (a, d) indices. On the other hand, this framework can be presented by distinct visualization styles. As an initial attempt, Gentil-Nunes [10] points to three: *standard* (Figure 2a), *stairs* (Figure 2b), and *combined* (Figure 2c).

The stairs style (Figure 2b) delineates the whole cutline of each partition's duration but, as a drawback, finishes to miss the angles that allow reading the relations. In fact, trying to assess the operators in a stairs indexogram implies the mental assumption of these angles. That is the main reason to embrace the standard view, once the essential function of the indexogram is not to iconically reproduce the esthesic dimension of the textural progressions but rather promote recognition of the sequence of operators and the bubbles.[5]

The combined style brings the difference between the indices expressed in a single line. In this case, the graph shows the prevalence and dynamic interaction between the indices.

Other alternatives already used include a stem style [17] (Figure 2d) and the temporal partiogram [12] (See Figure 4c, on page 22). The latter combines the indices (a, d) and the time points

[5]The same approach is adopted in Music Contour Theory [23].

in a single line delineated in a 3D arrangement. According to the analytical purposes, each style has its own application and advantages.

IV. Parsemat

Parsemat [12] is the original program that processes information regarding the textural partitions of a song from a MIDI or MusicXML file. The program analyzes the textural configurations at each point of attack, considering synchronized notes and their durations. Convergence is positive when there is a coincidence between these two data. The program also categorizes sustained pitches from previous attacks as synchronous.

The Parsemat program comes in two versions. The first version is a toolbox with 80 functions that the user can type on the command line within the Matlab program. These functions apply to two variables: the `note matrix`, the native format of the MIDI Toolbox [7], which is a matrix representation of MIDI events; and the variable `tab`, which consists of a list of attack points (`note-ons`) found in the piece, followed by the partitions resulted from the chosen analysis (`rhythmic`, `linear` or `per channel`).

The first command to type is `midi2nm`, which makes the routine for converting the MIDI file into a note matrix. The second command will determine the chosen analysis — `parsemarit(nm)`, `parsemalin(nm)`, or `parsemachan(nm)`. The result is always a `tab` variable. Finally, the user can choose the command for rendering the graph of choice — `partitiogram(tab)`, `indexogram(tab)`, or `tempartgram(tab)`, to result respectively in a *partitiogram*, *indexogram*, or *temporal partitiogram*.

The program has some ready-made scripts that perform all operations in sequence: `partrit`, `partlin`, `partchan`, `indrit`, `indlin`, `indchan`, `tempartrit`, `tempartlin`, and `tempartchan`. The script automatically carries out the entire sequence in response to the user command.

The second version of the program is standalone and can run on Windows and Mac OS systems (Figure 3). The interface presents buttons and menus that perform the reading operations, the assemblage of the variables `note matrix` and `tab`, which displays in the form of a spreadsheet embedded in the window, as well as the choice of analytical processes and graphics (Figure 4).

In the case of *Rhythmic Partitioning* (`parsemarit` function), the program performs the following operations:

1. Capture the list of all attack points.

2. Collate the list of attacks and durations of each note to check the situation at each point:

   (a) Simultaneous notes with the same duration (agglomeration).

   (b) Simultaneous notes with distinct durations (dispersion).

   (c) Suspended state notes - sustained from a previous attack (also agglomeration).

3. Counts the notes in the same situation. These groups generate the blocks that will make up the partition for each attack point.

4. Performs the calculation of agglomeration and dispersion indices for each partition, which then stand as coordinates (a, d) in the output graphs.

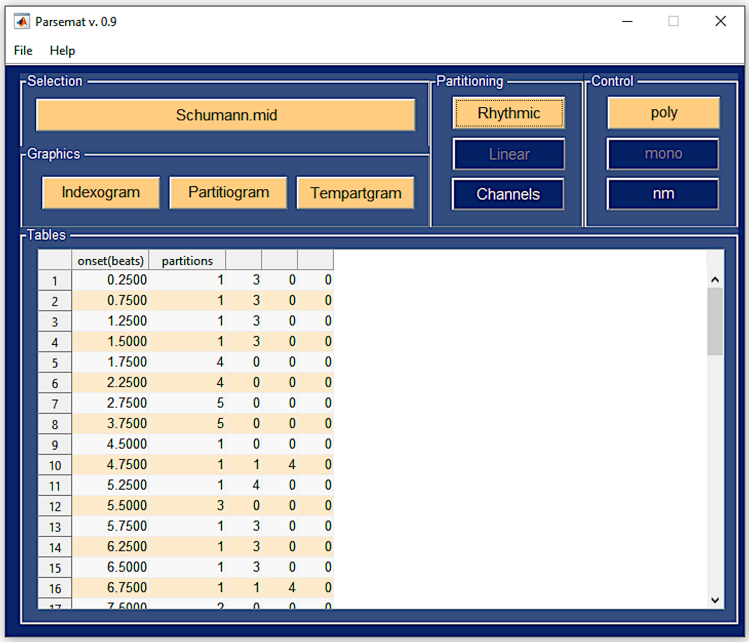
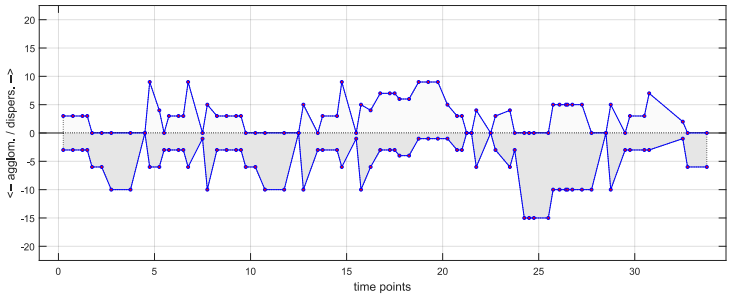
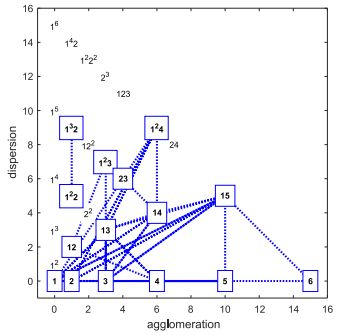


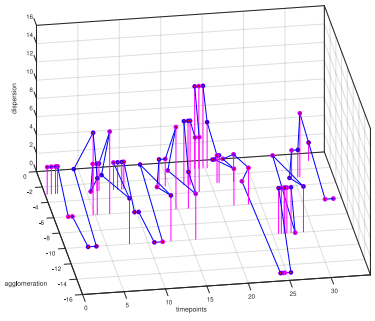
Figure 3: *Parsemat's interface.*



(a) *Indexogram.*



(b) *Partitiogram.*



(c) *Temporal partitiogram.*

Figure 4: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?). Parsemat's output. Generated by Parsemat [12].*

V. RP Scripts description

The RPC's main feature is calculating textural partition data from a given digital score. RPC collects musical events from the given digital score in MusicXML or Kern formats, gets their location in the score (measure numbers and offsets), calculates partitions, density-numbers[6], and agglomeration/dispersion values, and returns these data in a CSV file (See Listing 1). The output CSV file contains nine columns:

1. Index
2. Measure number
3. Offset
4. Global offset
5. Duration
6. Partition
7. Density number
8. Agglomeration index
9. Dispersion index

The *Index* column contains the events' locations in the format `measure+offset`. It is helpful for chart plotting. *Offset* is a Music21 class attribute that means the distance to the beginning. In this paper, the offset is related to the measure beginning, and global offset, to the piece beginning.

Listing 1: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?). Excerpt of RPC's output as a CSV file.*

```
"Index","Measure number","Offset","Global offset","Duration","
    Partition","Density-number","Agglomeration","Dispersion"
"1+0",1,0,0,1/4,"0",0,"",""
"1+1/4",1,1/4,1/4,3/2,"1.3",4,3.0,3.0
"1+1/2",1,1/2,1/2,3/2,"1.3",4,3.0,3.0
"2+0",2,0,3/4,3/2,"1.3",4,3.0,3.0
"2+1/4",2,1/4,1,3/2,"1.3",4,3.0,3.0
"2+1/2",2,1/2,5/4,3/2,"1.3",4,3.0,3.0
"2+3/4",2,3/4,3/2,3/2,"1.3",4,3.0,3.0
"2+1",2,1,7/4,1,"4",4,6.0,0.0
"2+5/4",2,5/4,2,1,"4",4,6.0,0.0
"2+3/2",2,3/2,9/4,1,"4",4,6.0,0.0
"2+7/4",2,7/4,5/2,1,"4",4,6.0,0.0
```

RPC takes advantage on multiple Music21's tools. The function `converter.parse` parses digital scores from different formats, such as Kern and MusicXML and outputs `stream.Stream` objects. These `Stream` objects contain multiple nested classes such as `Part`, `Voice`, `Measure`, `Note`, `Chord`, `Rest`, `Pitch`, and `Duration`.

RPC performs similar procedures to Parsemat (See Section IV):

[6]The *density-number* is an index referring to the number of concurrent sounding components in a given time point [4]. In this paper it is abbreviated as *dn*.

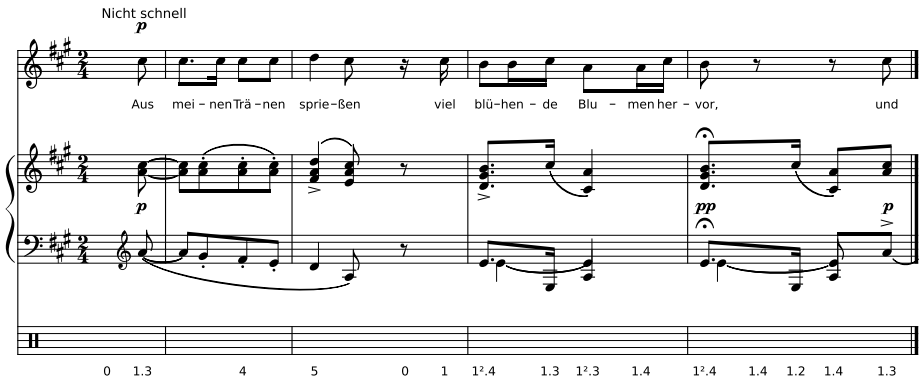


Figure 5: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?)., mm. 1–4. Rhythmic partitions annotated in a digital score. Generated by* RPA *Script.*

1. Extract musical events and their locations from the data input;
2. Map notes' and rests' beginnings and endings;
3. Loop through these boundaries to check other voices' notes;
4. Group events by duration to create partitions;
5. Calculate partitions' density-number and agglomeration/dispersion values;
6. Join adjacent partitions in *parsemae*.

RPA annotates the partitions information of RPC's CSV output file into the given digital score returning a new annotated MusicXML digital score. This file can be opened and edited in conventional score writers softwares (See Section VIII). It simply adds the partitions data into a new staff as note lyrics (Figure 5).

RPP takes advantage of *Pandas* [28] and *Matplotlib* [19] libraries functionalities. The script reads CSV data built by RPC and converts it to a `DataFrame` object. `DataFrame.plot` method generates the partitiogram and the indexogram and saves them in an SVG file. The functions `plot_simple_partitiogram` and `plot_simple_indexogram` among the functions correspondent to other indexogram styles solely add customized labels on `line` and `scatter` default charts and save them in SVG files. RPP outputs partitiogram and indexogram charts such in figures 2 (page 20), 6a, 6b (See both figures on page 28). Its source code is available in Appendix B.

i. RPC Structure

RPC is object-oriented and contains six object classes and auxiliary functions in a single module. Its source code is available in Appendix A. The auxiliary functions are helpful handling fractions, assisting events finding, and parsing Music21 events to `SingleEvent` objects. `Texture`, `Parsema`, and `ScoreSoundingMap` are the three most important script's classes. While `Texture` is the script's main class, `Parsema` represents the partitions, and `ScoreSoundingMap`, the music segmentation.

1. MusicalEvent
2. SingleEvent

3. Parsema
4. PartSoundingMap
5. ScoreSoundingMap
6. Texture

MusicalEvent A class that represents rests, notes, and chords. It simplifies Music21's structure, which contains different classes and nesting levels for these events. `MusicalEvent` class stores offset and global offset, number of pitches, duration, tie's type, and Music21 class of the given events (`Note`, `Chord` or `Rest`). This class has a `set_data_from_m21_obj` constructor method, with Music21's event, measure number, and measure offset as arguments.

SingleEvent An auxiliary class for sounding map creation. It is similar to `MusicalEvent` class, but with the additional boolean `sounding` attribute, and without `tie` and `m21_class` attributes.

Parsema A class representing repeated adjacent partitions. It stores the partitions sequence's location, duration, name, and list of `SingleEvents`. It provides methods to add events, and to get partition information, such as *agglomeration* and *dispersion* indexes (see Section II).

PartSoundingMap A map of the sounding events of a single musical part. It stores the list of part events and attacks' global offsets. It provides methods to parse `music21.stream.Part` and to get `SingleEvent` by location.

ScoreSoundingMap A map of sounding events for the complete musical piece. This class provides methods to add part sounding maps and create `Parsema` objects.

Texture The top-level class. It provides methods to generate `Parsema` objects from given `music21.stream.Stream` and to output the partitions data into CSV file.

ii. RPC procedure

RPC procedure consists of the following steps:

1. Parsing of digital score and conversion to the `music21.stream.base.Score` object with `music21.converter.parse` method;
2. Instantiation of `Texture` and `ScoreSoundingMap` objects;
3. Conversion of `Score`'s voices into parts with Score.voicesToParts method;
4. Instantiation of `PartSoundingMaps` objects;
5. Conversion of Music21's events into `SingleEvent` objects, storage of location data with `set_from_m21_part` method and `make_music_events_from_part` auxiliary function;
6. Creation of part's sounding and attack maps;
7. Instantiation of Sounding and attack analysis and `Parsema` method with `add_part_sounding_map`, `set_from_m21_part`, and `make_parsemae` methods;

8. Calculus of Density-number, agglomeration and dispersion with respectively `Parsema`'s methods;
9. Normalization with events of equal duration;
10. Creation of CSV file and output.

VI. RP Scripts installing and running

RP Scripts [24] depend on Python and a few libraries.[7] The command below installs Python libraries with built-in *pip* command:

```
pip install pandas numpy matplotlib music21
```

Since RPC, RPP, and RPA are standalone, there is no reason to install them in the system. Their running depends only on command line callback:

```
python rpc.py score.xml
python rpp.py score.csv
python rpa.py -s score.xml -c score.csv
```

RPP provides optional arguments (Listing 2) for choosing output image format, resolution and indexograms types (stairs, stem, combined, standard) and plotting bubble closing artificial lines. RPA demands the score (with `-s`) and csv files (with `-c`) as arguments to output the annotated MusicXML digital score.

Listing 2: *RPP's help output.*

```
usage: rpp [-h] [-f IMG_FORMAT] [-r RESOLUTION] [-a] [-c] [-e] [-t] [-b
    ] filename

Plot Partitiogram and Indexogram from RPC output

positional arguments:
filename

options:
-h, --help            show this help message and exit
-f IMG_FORMAT, --img_format IMG_FORMAT
                      Image format (svg, jpg or png). Default=svg
-r RESOLUTION, --resolution RESOLUTION
                      PNG image resolution. Default=300
-a, --all             Plot all available charts
-c, --close_bubbles   Close indexogram bubbles.
-e, --stem            Indexogram as a stem chart
-t, --stairs          Indexogram as a stair chart
-b, --combined        Indexogram as a combination of aglomeration and
                      dispersion

Rhythmic Partitioning Plotter
```

[7] Since the installing of Python and its libraries is well documented, this procedure is out of the scope of this paper.

Genre	Year	Composer	Title
Madrigal	1592	C. Monteverdi	*Poi ch'ella in sè tornò deserto e muto*, Third book of madrigals, n. 10
String quartet	1784	W. A. Mozart	String Quartet n. 17, K458, mov. I
Lied	1844?	R. Schumann	*Diechterliebe*, Op. 48, n. 2

Table 1: *Analysed corpus.*

VII. Application

We have generated partitioning data, partitiograms and indexograms for three pieces from Music21's repository [5] to illustrate the RP Scripts usage (Table 1).[8]

Monteverdi's madrigal contains well-distributed textural partitions from density-number zero to five (Figure 6a). Only partition (1^5) is absent in the piece. Most time, agglomeration and dispersion indexes are limited to the value of six, which correspond to density-number four (Figure 6b). Partitions with values higher than five are present only in strategic points such occur around measures 13, 18, 32, 65, 70, and 80. Furthermore, there are a few moments with lighter textures around measures 26 and 48.

Throughout the piece, peaks of dispersion and agglomeration are arranged in pairs, indicating a balance between a soft polyphony (as the figures of each voice are approximate with the same pace and rhythmic values but in asynchrony) and tiny fragments made of homorhythmic blocks. The indexogram made this textural dynamic visible by comparing superior and inferior peaks. The combination of latin vocal lyrics in concurrent parts does not follow this pattern in the middle of the phrases, eventually mixing syllables of different words (for instance, the last syllable of measure 7, Figure 6c). The convergence of rhythm and text is more substantial at the end of sections (for example, in mm. 18 to 19, Figure 6d).

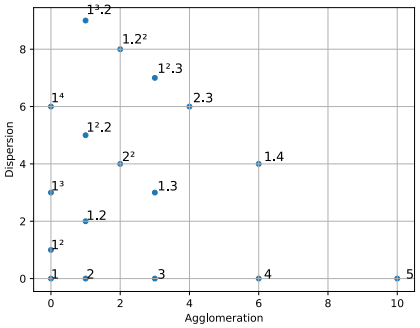
In Mozart's case (Figure 7), the quartet uses a repertoire of partitions very similar to Monteverdi's one—that is, the 11 partitions of the lexical-set of $dn = 4$, and some accessory partitions—in Monteverdi's case, coming from the fifth voice, and in the case of Mozart, the strings' double stops.

Monteverdi and Mozart use all partitions of $dn = 5$ but partition (1^5). In Mozart, this exclusion is understandable, as it would require a technique of double-stop polyphony that would be improbable in the instrumental language of his time. However, this same lack in Monteverdi is more surprising since it would be the natural expression of a five-part polyphony. An explanation for this could come from the rhythmic structure of the piece, which revolves around simple divisions of the quaternary measure. A five-voice polyphony would imply a complication of divisions outside the piece's character.

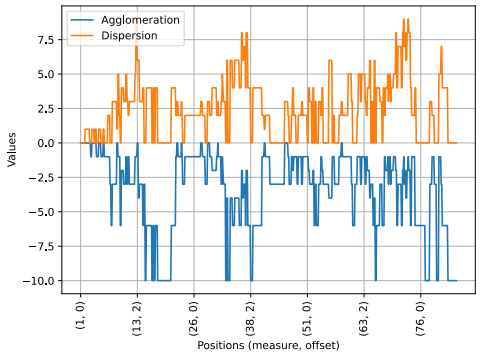
Mozart's partitions for $dn = 6$ and $dn = 8$ occur in the piece's final section and are in a purely cadential context. The arrangement in pairs of dispersion and agglomeration peaks also occurs in the Mozart excerpt. For example, in mm. 100–107.2 (Figure 7c), the alternation occurs between antecedent and consequent, which present themselves with contrasting profiles (dispersed - agglomerated), which points to the sense of completion and closure typical of agglomerated partitions (blocks).

In Schumman's excerpt of *Diechterliebe*—a piece for voice and piano—each dn is explored at its base, that is, in its most massive partitions, thus leaving aside the most dispersed partitions,

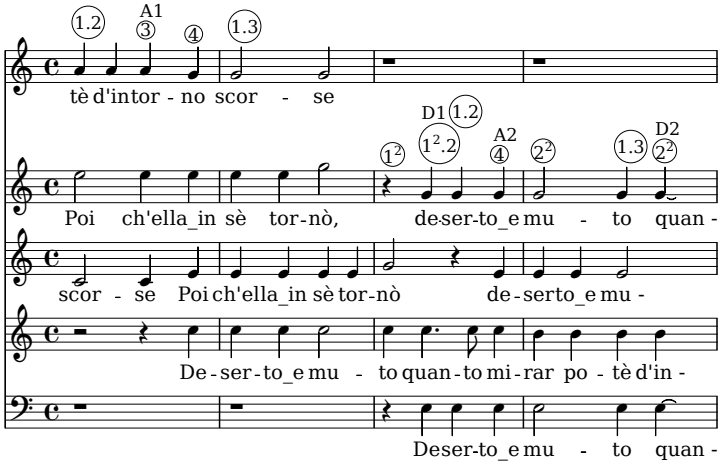
[8] We manually edited Monteverdi's digital score to add note tie endings for RPC's processing. See more information in Section VIII.



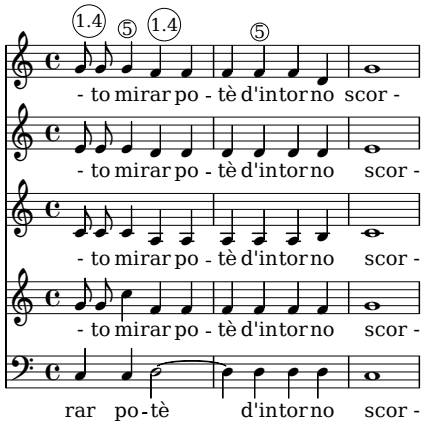
(a) *Partitiogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*



(b) *Indexogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*

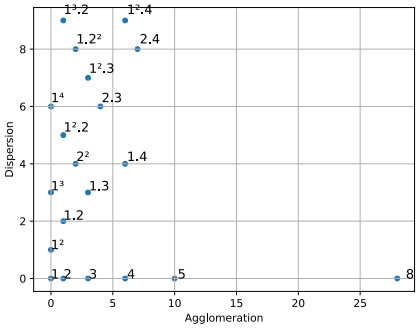


(c) *Mm. 7–10. Generated by Music21 [6] (see Section VIII) and Lilypond [21].*

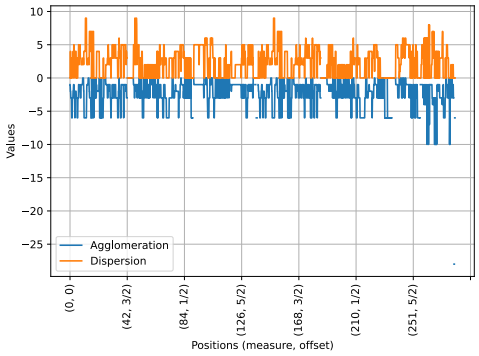


(d) *Mm. 17–19. Generated by Music21 [6] (see Section VIII) and Lilypond [21].*

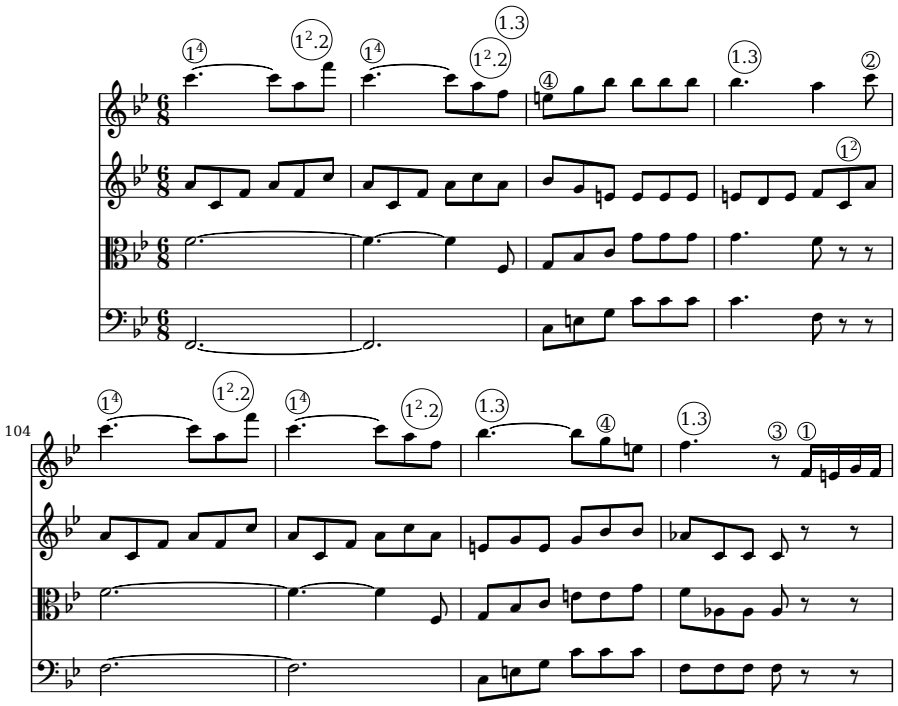
Figure 6: *C. Monteverdi. "Poi ch'ella in sè tornò deserto e muto", Il terzo libro de madrigali a cinque voci, n. 10, Venice (1592).*



(a) *Partitiogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*



(b) *Indexogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*



(c) *Mm. 100–107. Generated by Music21 [6] (see Section VIII) and Lilypond [21], edited manually to add the partitions annotations.*

Figure 7: *W.A. Mozart. String Quartet n. 17, K 458, mov. I (1784).*

corresponding to polyphonies (Figures 8a and 8b[9]). For example, of the 18 partitions in the lexical-set of $dn = 5$, only its 9 most agglomerated partitions are used.

On the other hand, we can see that the phrasal relationship between antecedent and consequent is also related to the dispersion-agglomeration progression. In the initial bubbles, closure occurs in agglomerated partitions (mm. 1–3, 4–6, 8–12, Figure 8c). Interestingly, this relationship is inverted in the last two bubbles of the excerpt (mm. 14–15 and 16–17, Figure 8d), forming a textural palindrome with the initial bubbles, which shows that contrast between dispersed and agglomerated partitions can work in both directions.

VIII. Discussion

Since RPC allows MusicXML and KRN files as input, its application potential is expressive. Major score writers such as Dorico, Finale, MuseScore, and Sibelius can export their scores to MusicXML files.[10] Furthermore, MuseScore and KernScores have large digital scores repositories in these file formats.

CSV files are popular, easy to parse, and readable by spreadsheet softwares. Therefore, this file format allows using output data for multiple purposes, such as data analysis and plotting. Moreover, spreadsheets softwares can easily filter partitions that are difficult to find in the indexograms. Additionally, RPA's output helps find all partitions directly into the music score.

The events' location by their measure numbers and offset is a notable feature of RPC. This information is helpful in piece comprehension since it allows the indexogram's X-axis labeling. Moreover, using these locations, along with Music21's `show` method, makes it possible to display the score of any specific piece point. For instance, the code below extracts and shows measures 7 to 10 on Figure 6c.

```
import music21
score = music21.converter.parse('monteverdi.xml')
measures = score.measures(7, 10)
measures.show()
```

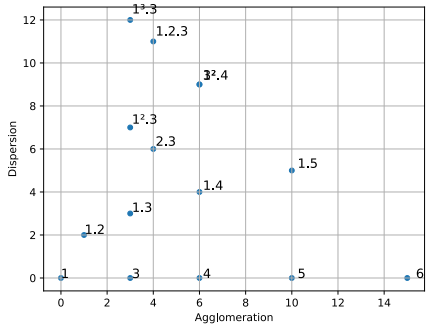
Another RP Scripts' highlight is the possibility of processing large corpora. Since they are standalone scripts, a concatenation in a shell script is possible. For instance, the single line below calls RPC and RPP to create CSV, indexogram, and partitiogram files from all the MusicXML files in a directory:

```
for f in *.xml; do python rpc.py $f && python rpp.py ${f%.xml}.csv &&
    python rpa.py -s $f.xml -c ${f%.xml}.csv; done
```

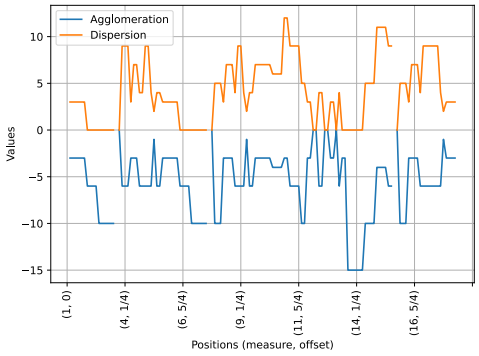
Parsemat's and RPC's outputs differ in two aspects: voice and rest handling. The voice processing is different due to the particularities of the MIDI and MusicXML data parsing. Given two equal MIDI notes coded in two different voices, if they are in the same channel, Parsemat processes them as a single part. RPC splits all part voices into new parts before processing partitions. Thus, RPC processes these equal notes as separate parts. This difference is more visible in instruments that allow multiple voices, such as the piano. For instance, in Schumann's fourth measure, the E4 note in the left hand is written twice (Figure 9a). Since this music staff occurs in only one channel, Parsemat processes only one occurrence of them. According to Parsemat, this excerpt's partition is (1) and (1^2). Since RPC splits these voices (Figure 9b), this excerpt's partition

[9] In spite on the anacrusis measure, the piece's indexogram (Figure 8b) starts in measure number 1 because the anacrusis measure is codified in this way in the piece's source. See a discussion about music codification in Section VIII.

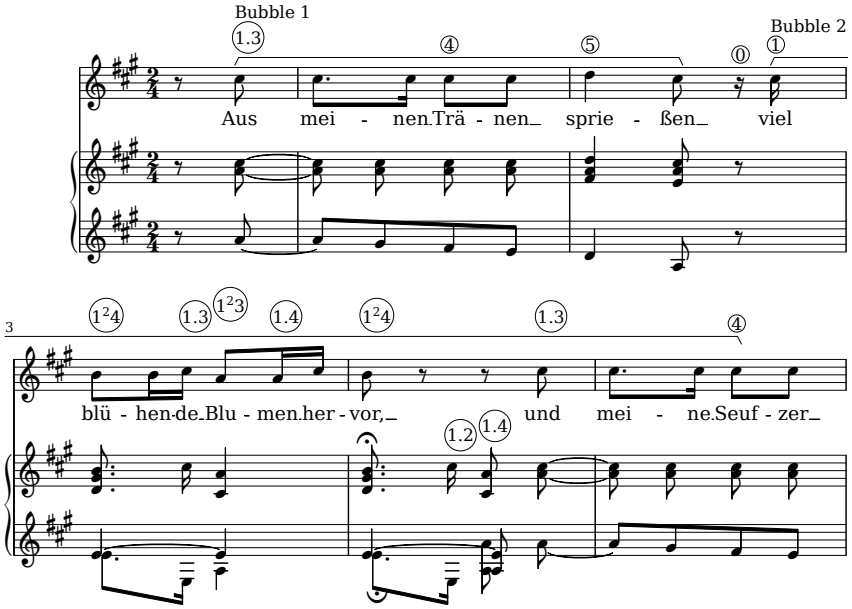
[10] See a complete software list with MusicXML export support at https://www.musicxml.com/software/.



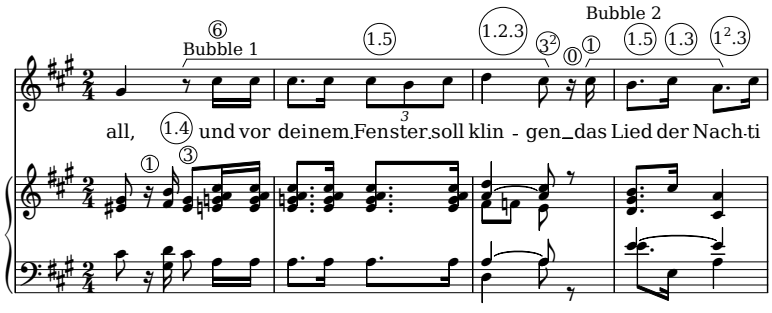
(a) *Partitiogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*



(b) *Indexogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*

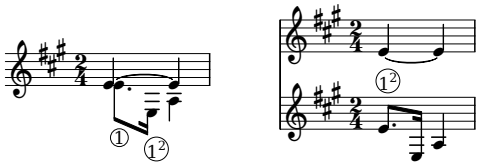


(c) *Mm. 1–5. Generated by Music21 [6] (see Section VIII) and Lilypond [21], edited manually to add the partitions and bubbles annotations.*



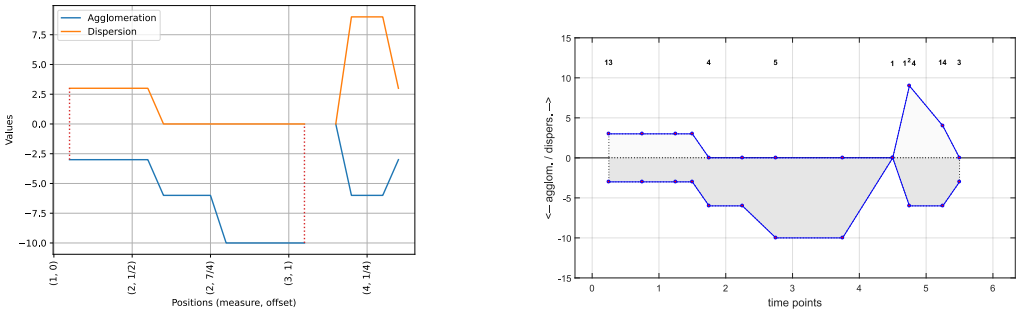
(d) *Mm. 13–16. Generated by Music21 [6] (see Section VIII) and Lilypond [21], edited manually to add the partitions and bubbles annotations.*

Figure 8: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?).*



(a) *Original.* **(b)** *Voices splitting into parts.*

Figure 9: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?). Voices processing approaches, m. 4, piano's left hand. Generated by Music21 [6] (see Section VIII) and Lilypond [21].*



(a) *RPC and RPP. Generated by RPC and RPP Scripts (see Appendixes A and B).* **(b)** *Parsemat. Generated by Parsemat [12].*

Figure 10: *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?). Mm. 0–4+3/4. Indexogram excerpts. See Section III.*

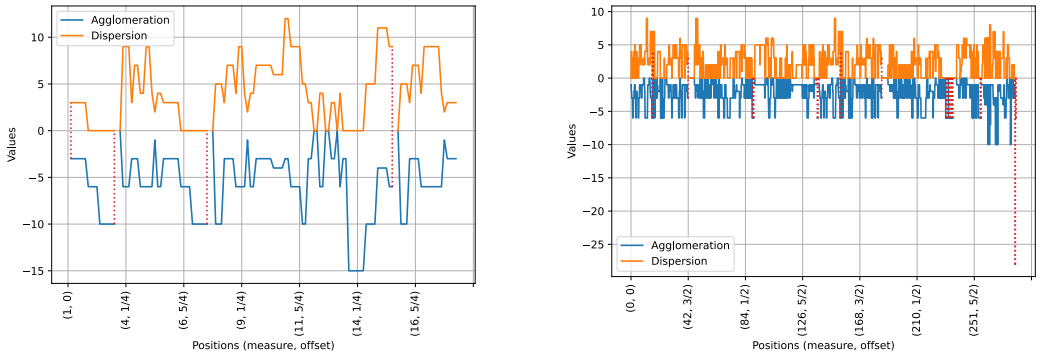
is only (1^2). This algorithm does not merge different voices in this situation. The present authors consider the inclusion of these possibilities as interface options in future releases of Parsemat and RPC.

RPC is sensitive to a precise musical representation. Thus, ambiguous decisions in coded music lead to processing errors during Music21 parsing and, consequently, during the script's processing. For instance, RPC needs explicit encoding of note tie endings to calculate the partitions and return bad results in processing scores without this information. This issue is not particular to this script but a common problem of the music processing. Accordingly to Elaine Selfridge-Field, "Common notation evolved with a view toward economy, but many conventions that save space or time in print complicate the operational instructions required to process musical information automatically" [26].

The other difference between Parsemat and RPC occurs in rest processing. RPC returns rest events with agglomeration and dispersion null values (not zero), while Parsemat's current version does not return rest events[11]. This procedure impacts indexogram creation resulting in empty spaces in RPC/RPP indexogram and linking points in the Parsemat indexogram. For instance, the rest at measure 2 (Figure 8c) is visible in RPC/RPP's indexogram (Figure 10a), but not in Parsemat's (Figure 10b, around time point 4).

Although RPC/RPP's approach reveals the rests in the indexogram, it compromises the bubbles identification. A possible solution to bubble visualization is drawing vertical lines at the edges

[11]According to Parsemat's website [12], "As the location of pauses affects the formal analysis, there is an option to read noteoffs in the command line version that will be inserted in the following program standalone versions. At the moment, the Parsemat standalone version simply ignores the pauses when creating the partitioning tables."



(a) *R. Schumann. Diechterliebe, Op. 48, n. 2 (1844?).*

(b) *W.A. Mozart. String Quartet n. 17, K 458, mov. I (1784).*

Figure 11: *Vertical lines closing indexograms' bubbles. Generated by RPC and RPP Scripts (see Appendixes A and B).*

of the rests (Figure 11). This solution improves the chart understanding in some cases, such as Schumman's indexogram (Compare figures 11a and 8b). However, these lines can pollute chart comprehension in more complex indexograms, such as Mozart's one (Figure 11b). The alternation between rests and notes in measures 231 and 236 pollutes this chart.

The representation of the pause as a discontinuity in the indexogram's temporal axis is a visual solution that aids the analysis. Anyway, silence as a rhythmic texture remains a conceptual issue to be addressed in future works.

IX. Conclusion

In the present paper, we introduced the *Rhythmic Partitioning Scripts* to get and plot events' locations, and annotating partitions info into digital scores, filling Parsemat's gaps. We presented their structures and source codes and analyzed three scores to demonstrate their usage.

The RP Scripts' Python basis allows integration with other tools such as Music21 to plot scores and run different types of music analysis. Furthermore, their input and output data formats are well known and permit analysis of large corpora of music scores.

As possible future work, these scripts can receive Linear- and Per-Event Partitioning functionalities and Graphical User Interface and be part of Music21 as a package.

References

[1] Alves, José Orlando (2005). *Invariâncias e disposições texturais: do planejamento composicional à reflexão sobre o processo criativo*. Tese (Doutorado em Música), Unicamp.

[2] Andrews, George (1984). *The Theory of Partitions*. Cambridge: Cambridge University Press.

[3] Andrews, George; Eriksson, Kimmo (2004). *Integer Partitions*. Cambridge: Cambridge University Press.

[4] Berry, Wallace (1976). *Structural functions in music*. New York: Dover Publications, Inc, pp. 184–194.

[5] Cuthbert, Michael Scott (2022). List of works found in the Music21 corpus. https://web.mit.edu/music21/doc/about/referenceCorpus.html.

[6] Cuthbert, Michael Scott; Ariza, Christopher (2010). Music21: a toolkit for computer-aided musicology and symbolic music data. International Society for Music Information Retrieval Conference, 11. *Proceedings ...* Utrecht: Universiteit Utrecht, pp. 637–642.

[7] Eerola, Tuomas; Toiviainen, Petri (2004). MIR in MATLAB: The MIDI Toolbox. International Society for Music Information Retrieval Conference, 5. *Proceedings ...* Barcelona: Universitat Pompeu Fabra.

[8] Fessel, Pablo (2007). La doble génesis del concepto de textura musical. *Revista Eletrônica de Musicologia*, 9, http://rem.ufpr.br/_REM/REMv11/05/05-fessel-textura.html.

[9] Fortes, Rafael (2016). *Modelagem e particionamento de Unidades Musicais Sistêmicas*. Dissertação (Mestrado em Música), Universidade Federal do Rio de Janeiro.

[10] Gentil-Nunes, Pauxy (2009). *Análise particional: uma mediação entre composição musical e a teoria das partições*. Tese (Doutorado em Música), Universidade Federal do Rio de Janeiro.

[11] Gentil-Nunes, Pauxy (2017). Partitiogram, Mnet, Vnet and Tnet: Embedded Abstractions Inside Compositional Games. In Pareyon, G. et al. (Eds) *The Musical-Mathematical Mind: Patterns and Transformations*, pp. 111–118. Berlin: Springer.

[12] Gentil-Nunes, Pauxy (2022). PARSEMAT: Parseme toolbox software package v. 0.9 beta. https://pauxy.net/parsemat-3/.

[13] Gentil-Nunes, Pauxy; Carvalho, Alexandre (2003). Densidade e linearidade na configuração de texturas musicais. Colóquio de Pesquisa do PPGM-UFRJ, IV. *Anais ...* , pp. 40–49, Rio de Janeiro: UFRJ.

[14] Gomes, Pedro Faria Proença (2022). CompTools: Tools for Assisting in Composing and Analyzing Music. Release 1.0.0. https://github.com/pedrofariacomposer/comptools.

[15] Good, Michael (2001). MusicXML for notation and analysis. *Computing in Musicology*, 12, pp. 113–124.

[16] Guigue, Didier (2009). *Esthétique de la sonorité - L'héritage de Debussy dans la musique pour piano du xxe siècle*. Paris: L'Harmattan.

[17] Guigue, Didier (2018). The function of orchestration in serial music: The case of webern's variations op. 30 and a proposal of theoretical analysis. *Musmat — Brazilian Journal of Music and Mathematics*, v. 2, n. 1, pp. 114–138.

[18] Guigue, Didier; Santana, Charles de Paiva (2018). The structural function of musical texture: Towards a computer-assisted analysis of orchestration. Journées d'Informatique Musicale JIM 2018 *Proceedings ...* Amiens.

[19] Hunter, John D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, v. 9, n. 3, pp. 90–95.

[20] Moreira, Daniel (2019). *Textural design: A Compositional Theory for the Organization of Musical Texture*. Tese (Doutorado em Música), Universidade Federal do Rio de Janeiro.

[21] Nienhuys, Han-Wen; Nieuwenhuizen, Jan (2003). Lilypond, a system for automated music engraving. Colloquium on Musical Informatics, XIV *Proceedings ...*, v. 1, pp. 167–171, Firenza: Tempo Reale.

[22] Ramos, Bernardo (2017). *Análise de textura violonística: teoria e aplicação*. Dissertação (Mestrado em Música), Universidade Federal do Rio de Janeiro.

[23] Sampaio, Marcos da Silva (2018). Contour similarity algorithms. *MusMat — Brazilian Journal of Music and Mathematics*, v. 2, n. 2, pp. 58–78.

[24] Sampaio, Marcos da Silva; Gentil-Nunes, Pauxy (2022). RP Scripts: Rhythmic Partitioning Scripts, release 1.0. https://github.com/msampaio/rpScripts.

[25] Craig Stuart Sapp. Online database of scores in the Humdrum file format. International Society for Music Information Retrieval Conference, 6 *Proceedings ...*, p. 2, London: Queen Mary University of London.

[26] Selfridge-Field, Eleanor (Ed.). *Beyond MIDI: the handbook of musical codes*. Cambridge, MA: MIT Press.

[27] Van Rossum, Guido; Drake, Fred L. (2009). *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace.

[28] McKinney, Wes (2010). Data Structures for Statistical Computing in Python. Python in Science Conference, 9 *Proceedings ...* , pp. 56–61, Austin, TX.

A. RPC' source-code

```
1  import argparse
2  import copy
3  import csv
4  import fractions
5  import math
6  import music21
7  import numpy
8
9  def get_number_combinations_pairs(n):
10     return n * (n - 1) / 2
11
12 def make_fraction(value):
13     if isinstance(value, fractions.Fraction):
14         return fractions.Fraction(int(value.numerator), int(value.
               denominator))
15     else:
16         a, b = value.as_integer_ratio()
17         return fractions.Fraction(int(a), int(b))
18
19 def get_common_fractions_denominator(fractions_lst):
20     denominators = [fr.denominator for fr in fractions_lst]
21     return numpy.lcm.reduce(denominators)
```

```
22
23 def get_common_denominator_from_list(seq):
24     diffs = [b - a for a, b in zip(seq, seq[1:])]
25     values = map(make_fraction, sorted(list(set(diffs))))
26     return fractions.Fraction(1, get_common_fractions_denominator(
           values))
27
28 def find_nearest_smaller(value, seq):
29     if value < seq[0]:
30         return -1
31
32     if value > seq[-1]:
33         return seq[-1]
34
35     size = len(seq)
36     if size == 1 and value >= seq[0]:
37         return seq[0]
38
39     middle_pointer = math.floor(size/2)
40     left = seq[:middle_pointer]
41     right = seq[middle_pointer:]
42
43     if value < right[0]:
44         return find_nearest_smaller(value, left)
45     else:
46         return find_nearest_smaller(value, right)
47
48 def auxiliary_find_interval(value, dic, i=0):
49     size = len(dic.keys())
50
51     if i > size - 1:
52         raise IndexError('Given index is out of dic')
53
54     keys = list(dic.keys())
55     while i < size - 1 and value >= dic[keys[i + 1]]:
56         i += 1
57
58     return keys[i], i
59
60 def aux_make_events_from_part(m21_part):
61     '''Return a dictionary with location and Musical Events
62     from a given Music21 part object.
63     '''
64
65     measures = m21_part.getElementsByClass(music21.stream.Measure)
66
67     events = {}
68
```

```
69      for m21_measure in measures:
70          notes_and_rests = m21_measure.notesAndRests
71          for m21_obj in notes_and_rests:
72              m_event = MusicalEvent()
73              m_event.set_data_from_m21_obj(m21_obj, m21_measure.number,
                    m21_measure.offset)
74              events.update({
75                  m_event.global_offset: m_event
76              })
77
78      return events
79
80
81  def aux_join_music_events(events):
82
83      # Add null event at the end
84      last_location = list(events.keys())[-1]
85      last_event = events[last_location]
86      last_location += last_event.duration + 1
87      current_event = MusicalEvent()
88      current_event.is_null = True
89      events.update({
90          last_location: MusicalEvent()
91      })
92
93      # Start with null
94      last_event = None
95      last_location = None
96      joined_events = {}
97
98      for location, current_event in events.items():
99          if current_event.is_null: # any - null
100             joined_events.update({last_location: last_event})
101         else:
102             if not last_event: # null - any
103                 last_event = current_event
104                 last_location = location
105             else:
106                 if current_event.is_rest():
107                     if last_event.is_rest(): # rest - rest
108                         last_event.duration += current_event.duration
109                     else: # note - rest
110                         joined_events.update({last_location: last_event
                                })
111                         last_event = current_event
112                         last_location = location
113                 else:
114                     if last_event.is_rest(): # rest - note
```

```
115                         joined_events.update({last_location: last_event
                                })
116                         last_event = current_event
117                         last_location = location
118                     else: # note - note
119                         if current_event.tie:
120                             if current_event.tie == 'start': # note -
                                    note.start
121                                 joined_events.update({last_location:
                                        last_event})
122                                 last_event = current_event
123                                 last_location = location
124                             else: # note - note.continue or note.stop
125                                 last_event.duration += current_event.
                                        duration
126                                 pass
127                         else:
128                                 joined_events.update({last_location:
                                        last_event})
129                                 last_event = current_event
130                                 last_location = location
131
132     return joined_events
133
134 def make_music_events_from_part(m21_part):
135     events = aux_make_events_from_part(m21_part)
136     return aux_join_music_events(events)
137
138 def pretty_partition_from_list(seq):
139     if not seq:
140         return '0'
141     dic = {}
142     for el in seq:
143         if el not in dic.keys():
144             dic[el] = 0
145         dic[el] += 1
146     partition = '.'.join([str(k) if v < 2 else '{}^{}'.format(k, v)
147         for k, v in sorted(dic.items())
148     ])
149
150     return partition
151
152 class CustomException(Exception):
153     pass
154
155 class MusicalEvent(object):
156     def __init__(self):
157         self.offset = 0
```

```
158         self.global_offset = 0
159         self.number_of_pitches = 0
160         self.duration = 0
161         self.tie = None
162         self.m21_class = None
163         self.is_null = False
164 
165     def __str__(self) -> str:
166         return ' '.join(list(map(str, [self.number_of_pitches, self.
                duration,  self.tie])))
167 
168     def __repr__(self):
169         return '<E {}>'.format(self.__str__())
170 
171     def is_rest(self):
172         return self.m21_class == music21.note.Rest
173 
174     def set_data_from_m21_obj(self, m21_obj, measure_number,
            measure_offset):
175         self.measure_number = measure_number
176         self.offset = make_fraction(m21_obj.offset)
177         self.global_offset = self.offset + make_fraction(measure_offset
                )
178         self.duration = make_fraction(m21_obj.duration.quarterLength)
179         self.m21_class = m21_obj.__class__
180 
181         if self.is_rest():
182             self.number_of_pitches = 0
183         else:
184             if m21_obj.isNote:
185                 self.number_of_pitches = 1
186             else:
187                 self.number_of_pitches = len(m21_obj.pitches)
188             if m21_obj.tie:
189                 if m21_obj.tie.type in ['start', 'continue', 'stop']:
190                     self.tie = m21_obj.tie.type
191 
192 class SingleEvent(object):
193     def __init__(self):
194         self.number_of_pitches = 0
195         self.duration = 0
196         self.measure_number = 0
197         self.offset = 0
198         self.sounding = False
199         self.partition_info = []
200 
201 class Parsema(object):
202     def __init__(self):
```

```
203         self.measure_number = None
204         self.offset = None
205         self.global_offset = None
206         self.duration = 0
207         self.single_events = []
208         self.partition_info = []
209         self.partition_pretty = ''
210 
211     def __repr__(self):
212         return '<P: {} ({}, {}), dur {}>'.format(self.partition_pretty,
                self.measure_number, self.offset, self.duration)
213 
214     def add_single_events(self, single_events):
215         self.single_events = single_events
216         durations = [event.duration for event in single_events if event
                ]
217         if durations:
218             self.duration = min(durations)
219 
220         self.set_partition_info()
221         self.partition_pretty = pretty_partition_from_list(self.
                partition_info)
222 
223     def set_partition_info(self):
224         partitions = {}
225         number_of_pitches_set = set([
226             s_event.number_of_pitches
227             for s_event in self.single_events
228         ])
229         if list(number_of_pitches_set) == [0]:
230             return [0]
231         for s_event in self.single_events:
232             key = (s_event.sounding, s_event.duration)
233             if key not in partitions.keys() and s_event.
                    number_of_pitches > 0:
234                 partitions[key] = 0
235             if s_event.number_of_pitches > 0:
236                 partitions[key] += s_event.number_of_pitches
237         self.partition_info = sorted(partitions.values())
238 
239     def get_density_number(self):
240         return int(sum(self.partition_info))
241 
242     def count_binary_relations(self):
243         density_number = self.get_density_number()
244         return get_number_combinations_pairs(density_number)
245 
246     def get_agglomeration_index(self):
```

```
247         if self.partition_info == []:
248             return None
249         return float(sum([get_number_combinations_pairs(n) for n in
                self.partition_info]))
250
251     def get_dispersion_index(self):
252         if self.partition_info == []:
253             return None
254         return float(self.count_binary_relations() - self.
                get_agglomeration_index())
255
256 class PartSoundingMap(object):
257     def __init__(self):
258         self.single_events = None
259         self.attack_global_offsets = []
260
261     def __str__(self):
262         return len(self.single_events.keys())
263
264     def __repr__(self):
265         return '<PSM: {} events>'.format(self.__str__())
266
267     def set_from_m21_part(self, m21_part):
268         music_events = make_music_events_from_part(m21_part)
269         self.single_events = {}
270         for global_offset, m_event in music_events.items():
271             # interval: closed start and open end.
272             closed_beginning = global_offset
273             open_ending = closed_beginning + m_event.duration
274
275             single_event = SingleEvent()
276             single_event.number_of_pitches = m_event.number_of_pitches
277             single_event.duration = m_event.duration
278             single_event.measure_number = m_event.measure_number
279             single_event.offset = m_event.offset
280
281             self.single_events.update({
282                 (closed_beginning, open_ending): single_event
283             })
284             self.attack_global_offsets.append(closed_beginning)
285
286     def get_single_event_by_location(self, global_offset):
287         beginning = find_nearest_smaller(global_offset, self.
                attack_global_offsets)
288
289         if beginning == -1: # No event to return
290             return
291
```

```
292         ind = self.attack_global_offsets.index(beginning)
293         _, ending = list(self.single_events.keys())[ind]
294         s_event = None
295         if global_offset >= beginning and global_offset < ending:
296             s_event = copy.deepcopy(self.single_events[(beginning,
                    ending)])
297             duration_diff = global_offset - beginning
298             duration = s_event.duration
299             duration = duration - duration_diff
300             sounding = duration_diff > 0
301             s_event.duration = duration
302             if s_event.number_of_pitches > 0:
303                 s_event.sounding = sounding
304             else:
305                 s_event.sounding = False
306         return s_event
307
308 class ScoreSoundingMap(object):
309     def __init__(self):
310         self.sounding_maps = []
311         self.attacks = []
312         self.measure_offsets = {}
313
314     def __repr__(self):
315         return '<SSM: {} maps, {} attacks>'.format(len(self.
                sounding_maps), len(self.attacks))
316
317     def add_part_sounding_map(self, m21_part):
318         psm = PartSoundingMap()
319         psm.set_from_m21_part(m21_part)
320         if psm.single_events:
321             self.sounding_maps.append(psm)
322             self.attacks.extend(psm.attack_global_offsets)
323             self.attacks = sorted(set(self.attacks))
324
325     def add_score_sounding_maps(self, m21_score):
326         # Get and fill measure offsets
327         offset_map = m21_score.parts[0].offsetMap()
328         self.measure_offsets = {
329             om.element.number: make_fraction(om.element.offset)
330             for om in offset_map
331             if isinstance(om.element, music21.stream.Measure)
332         }
333
334         # Get and fill sounding parts
335         parts = m21_score.voicesToParts()
336
337         for m21_part in parts:
```

```
338             self.add_part_sounding_map(m21_part)
339
340     def get_single_events_by_location(self, global_offset):
341         single_events = []
342         for sounding_map in self.sounding_maps:
343             s_event = sounding_map.get_single_event_by_location(
                    global_offset)
344             if s_event:
345                 single_events.append(s_event)
346         return single_events
347
348     def make_parsemae(self):
349         parsemae = []
350
351         offset_map = {ofs: ms for ms, ofs in self.measure_offsets.items
                ()}
352         all_offsets = list(offset_map.keys())
353
354         for attack in self.attacks:
355             measure_offset = find_nearest_smaller(attack, all_offsets)
356             measure_number = offset_map[measure_offset]
357             offset = make_fraction(attack) - make_fraction(
                    measure_offset)
358
359             parsema = Parsema()
360             parsema.add_single_events(self.
                    get_single_events_by_location(attack))
361             parsema.global_offset = attack
362             parsema.measure_number = measure_number
363             parsema.offset = offset
364             parsemae.append(parsema)
365
366         if not parsemae:
367             return
368
369         # Merge parsemae
370         merged_parsemae = []
371         first_parsema = parsemae[0]
372         for parsema in parsemae[1:]:
373             if parsema.partition_info == first_parsema.partition_info:
374                 first_parsema.duration += parsema.duration
375             else:
376                 merged_parsemae.append(first_parsema)
377                 first_parsema = parsema
378
379         merged_parsemae.append(first_parsema)
380
381         return merged_parsemae
```

```
382
383
384 class Texture(object):
385     def __init__(self):
386         self.parsemae = []
387         self._measure_offsets = {}
388
389     def __repr__(self):
390         return '<T: {} parsemae>'.format(len(self.parsemae))
391
392     def make_from_music21_score(self, m21_score):
393         ssm = ScoreSoundingMap()
394         ssm.add_score_sounding_maps(m21_score)
395         self.parsemae = ssm.make_parsemae()
396         self._measure_offsets = ssm.measure_offsets
397
398     def _auxiliary_get_data(self):
399         columns = [
400             'Index', # 0
401             'Measure number', # 1
402             'Offset', # 2
403             'Global offset', # 3
404             'Duration', # 4
405             'Partition', # 5
406             'Density-number', # 6
407             'Agglomeration', # 7
408             'Dispersion', # 8
409         ]
410         data = []
411         for parsema in self.parsemae:
412             ind = tuple([parsema.measure_number, parsema.offset])
413             data.append([
414                 ind,
415                 parsema.measure_number,
416                 parsema.offset,
417                 parsema.global_offset,
418                 parsema.duration,
419                 parsema.partition_pretty,
420                 parsema.get_density_number(),
421                 parsema.get_agglomeration_index(),
422                 parsema.get_dispersion_index(),
423             ])
424         dic = {
425             'header': columns,
426             'data': data
427         }
428         return dic
429
```

```
430     def _auxiliary_get_data_complete(self):
431         # check indexes
432         auxiliary_dic = self._auxiliary_get_data()
433         data = auxiliary_dic['data']
434         data_map = {row[3]: row for row in data}
435         global_offsets = [row[3] for row in data]
436         common = make_fraction(get_common_denominator_from_list(
                global_offsets))
437         size = global_offsets[-1] + data[-1][4]
438 
439         new_data = []
440         current_global_offset = global_offsets[0]
441         last_row = data[0]
442 
443         measure_index = 0
444         while current_global_offset < size:
445             current_measure, measure_index = auxiliary_find_interval(
                    current_global_offset, self._measure_offsets,
                    measure_index)
446 
447             if current_global_offset in data_map:
448                 row = copy.deepcopy(data_map[current_global_offset])
449                 last_row = copy.deepcopy(row)
450             else:
451                 row = copy.deepcopy(last_row)
452                 row[2] = current_global_offset - self._measure_offsets[
                        current_measure]
453                 row[3] = current_global_offset
454 
455             row[0] = '{}+{}'.format(str(current_measure), str(row[2]))
456             row[1] = current_measure
457             new_data.append(row)
458 
459             last_row = row
460             current_global_offset = make_fraction(current_global_offset
                    + common)
461 
462         dic = {
463             'header': auxiliary_dic['header'],
464             'data': new_data,
465         }
466 
467         return dic
468 
469     def get_data(self, equal_duration_events=True):
470         '''Get parsemae data as dictionary with data and index. If
                only_parsema_list attribute is False, the data is filled
                with equal duration events.'''
```

```
471
472         if equal_duration_events:
473             return self._auxiliary_get_data_complete()
474         else:
475             return self._auxiliary_get_data()
476
477
478 if __name__ == '__main__':
479     parser = argparse.ArgumentParser(
480                 prog = 'rpc',
481                 description = 'Rhythmic Partitioning Calculator',
482                 epilog = 'Rhythmic Partitioning Calculator')
483     parser.add_argument('filename')
484
485     args = parser.parse_args()
486     fname = args.filename
487
488     print('Running script on {} filename...'.format(fname))
489     try:
490         sco = music21.converter.parse(fname)
491     except:
492         raise CustomException('File must be XML or KRN.')
493
494     texture = Texture()
495     texture.make_from_music21_score(sco)
496     dic = texture.get_data(equal_duration_events=True)
497
498     # Filename
499     split_name = fname.split('.')
500     if len(split_name) > 2:
501         base = '.'.join(split_name[:-1])
502     else:
503         base = split_name[0]
504     dest = base + '.csv'
505
506     with open(dest, 'w') as fp:
507         csv_writer = csv.writer(fp, quoting=csv.QUOTE_NONNUMERIC)
508         csv_writer.writerow(dic['header'])
509         csv_writer.writerows(dic['data'])
```

B. RPP's source-code

```
1 from fractions import Fraction
2 from matplotlib import pyplot as plt
3 import argparse
4 import pandas
5
```

```
POW_DICT = {
    '1': '\N{superscript one}',
    '2': '\N{superscript two}',
    '3': '\N{superscript three}',
    '4': '\N{superscript four}',
    '5': '\N{superscript five}',
    '6': '\N{superscript six}',
    '7': '\N{superscript seven}',
    '8': '\N{superscript eight}',
    '9': '\N{superscript nine}',
}

class CustomException(Exception):
    pass

def parse_fraction(value):
    if isinstance(value, str):
        if '/' in value:
            return Fraction(*list(map(int, value.split('/'))))
    return value

def parse_index(v):
    a, b = v.split('+')
    return (a, parse_fraction(b))

def parse_pow(partition):
    parts = partition.split('.')
    new_parts = []
    for part in parts:
        value = part.split('^')
        if len(value) > 1:
            base, exp = value
            _exp = []
            for el in list(exp):
                _exp.append(POW_DICT[el])
            value = base + ''.join(_exp)
        else:
            value = value[0]
        new_parts.append(value)
    return '.'.join(new_parts)

def make_dataframe(fname):
    df = pandas.read_csv(fname)
    for c in ['Agglomeration', 'Dispersion']:
        df[c] = df[c].apply(float)

    for c in ['Offset', 'Global offset', 'Duration']:
        df[c] = df[c].apply(parse_fraction)
```

```
54
55      df.index = df['Index'].apply(parse_index).values
56      df['Partition'] = df['Partition'].apply(parse_pow)
57      df = df.drop('Index', axis=1)
58
59      return df
60
61  def invert_dataframe(df):
62      inverted = pandas.DataFrame([
63          df.Agglomeration * -1,
64          df.Dispersion,
65      ], index=['Agglomeration', 'Dispersion'], columns=df.index).T
66      return inverted
67
68  def plot_simple_partitiogram(df, img_format='svg', with_labels=True,
        outfile=None):
69      seq = [
70          [partition, len(_df), _df.Agglomeration.iloc[0], _df.Dispersion
                .iloc[0]]
71          for partition, _df in df.groupby('Partition')
72      ]
73      columns=['Partition', 'Quantity', 'Agglomeration', 'Dispersion']
74      df = pandas.DataFrame(seq, columns=columns)
75
76      plt.clf()
77      ax = df.plot(
78          grid=True,
79          kind='scatter',
80          x='Agglomeration',
81          y='Dispersion',
82      )
83
84      if with_labels:
85          factor = 1.025
86          fontsize = 12
87
88          for _, s in df.iterrows():
89              x = s['Agglomeration']
90              y = s['Dispersion']
91              v = s['Partition']
92              plt.text(x * factor, y * factor , v, fontsize=fontsize)
93
94      if img_format == 'svg':
95          plt.savefig(outfile)
96      else:
97          plt.savefig(outfile, dpi=RESOLUTION)
98      plt.close()
99
```

```
100 def plot_simple_indexogram(df, img_format='svg', outfile=None,
        close_bubbles=False):
101     def draw_vertical_line(row, x):
102         ymin = row[1].Agglomeration * -1
103         ymax = row[1].Dispersion
104         plt.vlines(x=x, ymin=ymin, ymax=ymax, linestyles='dotted',
                colors='C3')
105
106     inverted = invert_dataframe(df)
107
108     plt.clf()
109
110     ax = inverted.plot(grid=True)
111     ax.set_ylabel('Values\n<- aggl./disp. ->')
112     ax.set_xlabel('Positions (measure, offset)')
113
114     # draw vertical lines to close the bubbles
115     if close_bubbles:
116         rest_segment = False
117         last_row = None
118         for i, row in enumerate(df.iterrows()):
119             _agg = row[1].Agglomeration
120             if pandas.isnull(_agg):
121                 if not rest_segment:
122                     if last_row:
123                         x = i - 1
124                         draw_vertical_line(last_row, x)
125                     rest_segment = True
126             else:
127                 if rest_segment:
128                     x = i
129                     draw_vertical_line(row, x)
130                     rest_segment = False
131             last_row = row
132
133     plt.xticks(rotation=90)
134     plt.tight_layout()
135
136     if img_format == 'svg':
137         plt.savefig(outfile)
138     else:
139         plt.savefig(outfile, dpi=RESOLUTION)
140     plt.close()
141
142 def plot_stem_indexogram(df, img_format='svg', outfile=None):
143     inverted = invert_dataframe(df)
144
145     ind = ['({}, {})'.format(a, b) for a, b in inverted.index.values]
```

```
146     size = len(ind)
147     step = int(size / 8)
148 
149     plt.clf()
150     plt.stem(ind, inverted.Dispersion.values, markerfmt=' ')
151     plt.stem(ind, inverted.Agglomeration.values, markerfmt=' ', linefmt
            ='C1-')
152     plt.xticks(range(0, size, step))
153     plt.xlabel('Positions (measure, offset)')
154     plt.ylabel('Values\n<- aggl./disp. ->')
155     plt.grid()
156     plt.legend(inverted.columns)
157     plt.xticks(rotation=90)
158     plt.tight_layout()
159 
160     if img_format == 'svg':
161         plt.savefig(outfile)
162     else:
163         plt.savefig(outfile, dpi=RESOLUTION)
164     plt.close()
165 
166 def plot_stairs_indexogram(df, img_format='svg', outfile=None):
167     inverted = invert_dataframe(df)
168 
169     ind = ['({}, {})'.format(a, b) for a, b in inverted.index.values]
170     size = len(ind)
171     step = int(size / 8)
172 
173     plt.clf()
174     plt.stairs(inverted.Dispersion.values[:-1], ind)
175     plt.stairs(inverted.Agglomeration.values[:-1], ind)
176     plt.xticks(range(0, size, step))
177     plt.xlabel('Positions (measure, offset)')
178     plt.ylabel('Values\n<- aggl./disp. ->')
179     plt.grid()
180     plt.legend(inverted.columns)
181     plt.xticks(rotation=90)
182     plt.tight_layout()
183 
184     if img_format == 'svg':
185         plt.savefig(outfile)
186     else:
187         plt.savefig(outfile, dpi=RESOLUTION)
188     plt.close()
189 
190 def plot_combined_indexogram(df, img_format='svg', outfile=None):
191     inverted = invert_dataframe(df)
192     series = inverted.Dispersion + inverted.Agglomeration
```

```
193
194     plt.clf()
195     ax = series.plot(grid=True)
196     ax.set_ylabel('Values\n(d - a)')
197     ax.set_xlabel('Positions (measure, offset)')
198
199     plt.xticks(rotation=90)
200     plt.tight_layout()
201
202     if img_format == 'svg':
203         plt.savefig(outfile)
204     else:
205         plt.savefig(outfile, dpi=RESOLUTION)
206     plt.close()
207
208 if __name__ == '__main__':
209     parser = argparse.ArgumentParser(
210                     prog = 'rpp',
211                     description = "Plot Partitiogram and Indexogram
                            from RPC's output",
212                     epilog = 'Rhythmic Partitioning Plotter')
213
214     parser.add_argument('filename')
215     parser.add_argument("-f", "--img_format", help = "Image format (svg
            , jpg or png). Default=svg", default='svg')
216     parser.add_argument("-r", "--resolution", help = "PNG image
            resolution. Default=300", default=300)
217     parser.add_argument("-a", "--all", help = "Plot all available
            charts", action='store_true')
218     parser.add_argument("-c", "--close_bubbles", help = "Close
            indexogram bubbles.", default=False, action='store_true')
219     parser.add_argument("-e", "--stem", help = "Indexogram as a stem
            chart", action='store_true')
220     parser.add_argument("-t", "--stairs", help = "Indexogram as a stair
             chart", action='store_true')
221     parser.add_argument("-b", "--combined", help = "Indexogram as a
            combination of aglomeration and dispersion", action='store_true'
            )
222     args = parser.parse_args()
223
224     try:
225         RESOLUTION = int(args.resolution)
226     except:
227         raise CustomException('Resolution must be an integer from 0 to
                1200')
228
229     close_bubbles = args.close_bubbles
230     if close_bubbles:
```

```
231         close_bubbles = True
232
233     img_format = args.img_format.lower()
234     if img_format not in ['svg', 'jpg', 'png']:
235         raise CustomException('Image format must be svg, jpg or png.')
236
237     fname = args.filename
238
239     print('Running script on {} filename...'.format(fname))
240
241     indexogram_choices = {
242         'simple': plot_simple_indexogram,
243         'stem': plot_stem_indexogram,
244         'stairs': plot_stairs_indexogram,
245         'combined': plot_combined_indexogram,
246     }
247
248     try:
249         df = make_dataframe(fname)
250         bname = fname.rstrip('.csv')
251
252         partitiogram_name = bname + '-partitiogram.' + img_format
253         plot_simple_partitiogram(df, img_format, outfile=
                partitiogram_name)
254
255         if args.all:
256             for k, fn in indexogram_choices.items():
257                 outfile = bname + '-indexogram-{}.'.format(k) +
                        img_format
258                 fn(df, img_format, outfile=outfile)
259         elif args.stem:
260             k = 'stem'
261             fn = indexogram_choices[k]
262             outfile = bname + '-indexogram-{}.'.format(k) + img_format
263             fn(df, img_format, outfile=outfile)
264         elif args.stairs:
265             k = 'stairs'
266             fn = indexogram_choices[k]
267             outfile = bname + '-indexogram-{}.'.format(k) + img_format
268             fn(df, img_format, outfile=outfile)
269         elif args.combined:
270             k = 'combined'
271             fn = indexogram_choices[k]
272             outfile = bname + '-indexogram-{}.'.format(k) + img_format
273             fn(df, img_format, outfile=outfile)
274         else:
275             k = 'simple'
276             fn = indexogram_choices[k]
```

```
277             outfile = bname + '-indexogram.' + img_format
278             fn(df, img_format, outfile=outfile, close_bubbles=
                    close_bubbles)
279
280     except:
281         raise CustomException('Something wrong with given csv file.')
```

C. RPA's source-code

```
1  import argparse
2  import csv
3  import fractions
4  import math
5  import music21
6
7  def find_nearest_smaller(value, seq):
8      if value < seq[0]:
9          return -1
10
11     if value > seq[-1]:
12         return seq[-1]
13
14     size = len(seq)
15
16     if size == 1 and value >= seq[0]:
17         return seq[0]
18
19     middle_pointer = math.floor(size/2)
20     left = seq[:middle_pointer]
21     right = seq[middle_pointer:]
22
23     if value < right[0]:
24         return find_nearest_smaller(value, left)
25     else:
26         return find_nearest_smaller(value, right)
27
28 def simplify_csv(csv_fname):
29     seq = []
30     last_row = None
31     with open(csv_fname, 'r') as fp:
32         i = 0
33         for row in csv.reader(fp):
34             if i > 0:
35                 if i == 1:
36                     last_row = row
37                 if row[5] != last_row[5]:
38                     seq.append(last_row)
```

```
                    last_row = row
            i += 1
    return seq

def make_offset_map(sco):
    measures = sco.parts[0].getElementsByClass(music21.stream.Measure).
        stream()
    return {om.element.offset: om.element.number for om in measures.
        offsetMap()}

def get_events_location(sco, csv_fname):
    offset_map = make_offset_map(sco)
    offsets = list(offset_map.keys())
    seq = simplify_csv(csv_fname)

    events_location = {}

    for row in seq:
        if row[3] == '0':
            a, b = 0, 1
        elif '/' in row[3]:
            a, b = list(map(int, row[3].split('/')))
        else:
            a, b = int(row[3]), 1
        global_offset = fractions.Fraction(a, b)
        partition = row[5]
        measure_offset = find_nearest_smaller(global_offset, offsets)
        measure_number = offset_map[measure_offset]
        offset = global_offset - measure_offset
        if measure_number not in events_location:
            events_location[measure_number] = []
        events_location[measure_number].append((offset, partition))

    return events_location

def main(sco, csv_fname, outfile):
    events_location = get_events_location(sco, csv_fname)

    p0 = sco.parts[0]
    new_part = music21.stream.Stream()
    new_part.insert(0, music21.clef.PercussionClef())

    measures = {}

    for m in p0.getElementsByClass(music21.stream.Measure):
        new_measure = music21.stream.Measure()
        new_measure.number = m.number
        new_measure.offset = m.offset
```

```
85          if m.number in events_location.keys():
86              for _offset, partition in events_location[m.number]:
87                  rest = music21.note.Rest(quarterLength=1/256)
88                  rest.offset = _offset
89                  rest.addLyric(partition)
90                  new_measure.insert(_offset, rest)
91              new_measure = new_measure.makeRests(fillGaps=True)
92              for el in new_measure:
93                  el.style.color = 'white'
94                  el.style.hideObjectOnPrint = True
95          measures.update({m.number: new_measure})
96
97      for m in measures.values():
98          new_part.append(m)
99
100     new_part = new_part.makeRests(fillGaps=True)
101
102     sco.insert(0, new_part)
103     sco.write(fmt='xml', fp=outfile)
104
105 if __name__ == '__main__':
106     parser = argparse.ArgumentParser(
107                 prog = 'rpa',
108                 description = 'Rhythmic Partitioning Annotator',
109                 epilog = 'Rhythmic Partitioning Annotator')
110     parser.add_argument("-s", "--score", help = "Score filename.")
111     parser.add_argument("-c", "--csv", help = "CSV filename.")
112
113     args = parser.parse_args()
114     sco_fname = args.score
115     csv_fname = args.csv
116
117     print('Running script on {} filename...'.format(sco_fname))
118
119     sco = music21.converter.parse(sco_fname)
120     outfile = csv_fname.rstrip('.csv') + '-annotated.xml'
121
122     main(sco, csv_fname, outfile)
```

On Xenakis's Games of Musical Strategy

Stefano Kalonaris
RIKEN Center for Advanced Intelligence Project (AIP)
stefano.kalonaris@riken.jp
Orcid: 0000-0002-7372-323X
DOI: 10.46926/musmat.2022v6n2.56-71

Abstract: *In this article, Xenakis's trilogy of pieces based on zero-sum games is investigated, revealing axiomatic inconsistencies between game-theoretical models and their musical translations implemented according to the aesthetic preferences of the composer. The problematic elements regard 1) the formal definition of the games and 2) the limits of objective utility functions in the music domain. After introducing these games of musical strategy and the few existing practical implementations found in the literature, a detailed comparison between the mathematical models and Xenakis's renditions is presented to highlight their divergence. Then, the feasibility from a musical perspective of the rational decision-making required by the models is explored through a computational simulation of game dynamics using Reinforcement Learning. Lastly, the article concludes by contextualising the findings to the increasingly ubiquitous role of the machine in the creative processes of musical composition and generation. In doing so, Xenakis's original works offer themselves as a springboard for (re-)imagining and developing novel approaches integrating human-machine decision processes with musical design and interaction.*

Keywords*: Iannis Xenakis, Game Theory, Reinforcement Learning.*

I Introduction

Duel (1959), *Stratégie* (1962) and *Linaia-Agon* (1972) form a trilogy of pieces that Xenakis wrote using identical underlying principles. Such principles imply *external* rather than *internal* conflict, with the latter relating to intrinsic factors (*i.e.*, the dialogical relation between the sound rendition and the symbolic schema) and the former involving extrinsic factors. Xenakis calls the music originating from these *heteronomous* (external conflict) and *autonomous* (internal conflict). These three works can be classified as *game pieces* [7, 9], which would draw inspiration from a notion of "game" linked to ludic activities, social theory, anthropology and game design. For example, John Zorn's *Cobra* [4] focuses on emergent social dynamics, Mauricio Kagel's *Match* [11] is inspired by tennis, and Mathius Shadow-Sky's *Ludus Musicae Temporarium* [22] is based on the work of Huizinga [10] and Caillois [6].

Xenakis's trilogy, however, differs in that it is stricter in its application of formal and rational theories on decision-making to the music domain. More specifically, it is rooted in the applied

Received: October 27th, 2022
Approved: December 21st, 2022

mathematics field known as *game theory*, which models scenarios of conflict or cooperation between agents. While game theory is popular in economics, social network theory, and computer science (with applications ranging from artificial intelligence to network systems), game-theoretical musical pieces remain relatively under-explored. Besides some sporadic experiments with Bayesian games of imperfect information in the context of networked music performance [14, 13, 12], game-theoretical formalisms are not a popular paradigm for composition or sound design, because of the difficulty of mapping objective utility functions in the music domain. That is, abiding by the principles of optimisation, rational decision-making and mathematical solutions might be, at times, undesirable if in conflict with musical and aesthetic goals. Imaginary rewards (*e.g.*, "points" won or lost in a game) may be less motivating than sonic, experiential counterparts (*e.g.*, the perceived quality of the music or the musical interactions originating from the musical game piece).

Xenakis's game pieces are not devoid of these issues, and it is important to consider them in the context of the (then) newly re-defined role of the composer: caught in between "inventing schemes (previously forms) and exploring the limits of these schemes", and "effecting the scientific synthesis of the new methods of construction and of sound emission" [32, p.133]. While uncompromisingly and single-mindedly striving for novelty and originality, Xenakis retained an executive role rooted in the idea of the composer as the sole owner of the work. His indisputable aesthetic compass, for example, is asserted when stating that "the winner has won simply because he has better followed the rules imposed by the composer, who, by consequence, claims all responsibility for the 'beauty' or 'ugliness' of the music" [31].

However, game theory has an aesthetic of its own. In fact, this might be the true *raison d'etre* of these works, as Xenakis confesses that he had "[...] been interested in social questions, in the relationship between people and the aesthetic aspect of all that" [28, p.49]. One might say that Xenakis's games of musical strategy find their most defining characteristic to be this tension between internal (the composer's quest for aesthetic integrity) and external conflict (opposing interests which must be allowed to emerge).

When revisiting Xenakis's games of musical strategy, it is also crucial to acknowledge the changes brought forward by the increasingly common application of AI techniques and methods in the music domain and how these retroactively affect one's understanding and appreciation of Xenakis's game pieces. According to Xenakis, the role of the machine is perceived either through negative or positive bias or as an explorative process which, however, is ultimately not sufficient *per se* as a means to artistic value [32]. In agreement with this viewpoint, this article is, nevertheless, focused on analyses and implementations of Xenakis's trilogy comprising a computer-assisted factor.

I.i Background

Despite their simplicity and potential for myriad variations and implementations, Xenakis's game pieces have not enjoyed as much attention as the remainder of his body of work. Liuni & Morelli [18] rendered *Duel* as a live installation in which members of the public could take the role of the conductors. Their movements were analysed by computer vision algorithms to drive the score, which was, in turn, visualised on a large screen. This rendition concentrated on the interactive participation of the human players, and the musical events prescribed in the original score were realised using audio samples instead of real orchestras. Regarding *Linaia-Agon*, there exists a documentary DVD [24] with original radio broadcasts and newly recorded live and studio performances of the piece using a computational interface.

As for the computational analysis of Xenakis's trilogy, Sluchin & Malt [25] simulated *Duel* based on four different methods for choosing tactics (see Section II.iii). Beyond the numerical simulation

of game dynamics, that work, and a follow-up study [26] which included a computational interface to switch between different strategies, does not point to an audio rendition of the piece. *Linaia-Agon*, finally, was analysed by DeLio [8], Sluchin [23], and Beguš [3].

The next section precedes the formal analysis of Xenakis's trilogy by introducing the reader to some fundamental notions in game theory.

II Fundamental Notions

In the context of game theory, "a game is a description of strategic interaction that includes the constraints on the actions that the players can take and the players' interests, but does not specify the actions that the players do take" [21, p.2]. The basic elements of a game are:

- a finite set of **players** N
- for each player $i \in N$ a non-empty set A_i representing the set of **actions** available to player i
- for each player $i \in N$ a **preference relation** $\succsim_i$ on $A = \times_{j \in N} A_j$ (the set of outcomes by A)

Morgenstern & von Neumann [19] are credited as the initiators of modern game theory, which is normally divided into two main branches: **non-cooperative** and **cooperative** game theory. The former considers each player's individual actions as primitives, whereas the latter sees joint actions as primitives, and assumes that binding agreements can be made by players and within groups of players. For the sake of this article, only non-cooperative games are discussed.

Another fundamental characterisation of games that will be crucial for Xenakis's game pieces is based on the **utility function** which maps rewards to actions $u_i : A \to \mathbb{R}$, so that $u_i(a) \geq u_i(b)$ whenever $a \succsim_i b$. These "rewards" are hereinafter referred to as **payoffs**. To this end, one can distinguish between **constant-sum** and **variable-sum** games. In the former, the payoffs for each possible combination of actions sum up to the same constant C. A particular case of constant-sum games is **zero-sum** games. This means, simply, that for every combination of players' actions, one player's gains are the other's losses, and therefore payoffs sum up to 0. In variable-sum games, on the other hand, the rewards are neither symmetrical (opposites) nor sum up to a constant.

A **strategy** is a decision algorithm which considers the options available under a given scenario. In other words, a strategy is a complete contingent plan that defines the action a player will take in all states of the game. A **strategy profile** is a set of strategies for all players that fully specify all actions in a game.

Insofar as the discussions of this article are concerned, the most important aspects of a game entail notions of *time*, *information*, and *equilibrium*.

II.i Time

This dimension determines whether games are *simultaneous* or *sequential*, meaning whether players take actions synchronously and independently from each other, or in turns, respectively. Simultaneous and sequential games are normally notated differently. For the former, one uses the **normal form** (or **strategic form**), while for the latter the **extensive form** (or **game tree**). For simplicity, two players are considered, hereinafter x and y. Examples of two games and their graphical representations can be seen in Figure 1. In zero-sum games, such as the game shown on the left (known as *Rock, Paper, Scissors*), for each cell in the matrix, payoffs are expressed as a single signed integer. Since this is a zero-sum game, the payoffs are symmetrical (opposites); for example, -1 means $(-1, 1)$ where the first value in the tuple would be assigned to x and the second to y. Instead, in the sequential game depicted on the right, the payoffs are not symmetrical and are explicitly expressed as tuples of values.

		y		
		Rock	Paper	Scissors
x	Rock	0	-1	1
	Paper	1	0	-1
	Scissors	-1	1	0

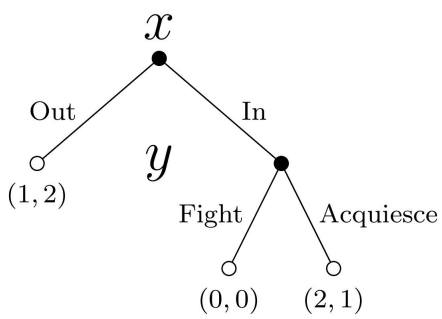


Figure 1: *Simultaneous (left) and sequential (right) games, and their notation: normal and extensive form, respectively.*

II.ii Information

In this dimension, one must distinguish between **perfect** *vs.* **imperfect** and **complete** *vs.* **incomplete** information. The former describes whether or not players have knowledge of each others' actions and history. The latter is instead concerned with common knowledge of each player's utility functions, payoffs, strategies and "types". The two are not mutually exclusive: for example, a game could have perfect and incomplete information, and so forth. These distinctions are not trivial and fundamentally affect the decision-making process.

It is possible to convert simultaneous games from normal to extensive form, and vice versa (**induced normal form**). To illustrate this procedure, a zero-sum game known as *Matching Pennies* is considered. To convert it to extended form using the game matrix, one introduces an **information set**, indicated as a dotted ellipse enveloping decision leaf nodes. The information set shown in the game tree on the right in Figure 2 indicates that player y, while moving after player x, has no knowledge of what action the opponent chose and, therefore, whether she finds herself under the right or left leaf node. This uncertainty would define such a game as an extensive form game of incomplete information.

		y	
		heads	tails
x	heads	1	-1
	tails	-1	1

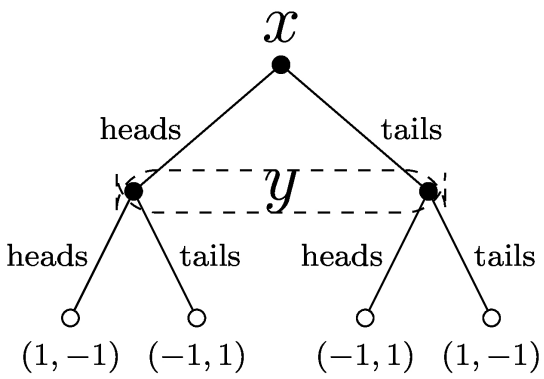


Figure 2: *A popular zero-sum game known as* Matching Pennies*, shown in matrix form (left), and in an equivalent extensive form (right).*

However, this article is only concerned with games with perfect and complete information, either simultaneous or sequential. From now on, the former will be referred to simply as **strategic games**, whereas extensive-form games with complete information will be referred to as **extensive**

games. For the latter, two more elements are needed for a formal definition:

- a set H of sequences where each member is a **history** and each component of a history is an action. A history $(a^k)_{k=1,\dots,K} \in H$ is **terminal** if it is infinite or if there is no a^{K+1} such that $(a^k)_{k=1,\dots,K+1} \in H$. The set of terminal histories is indicated as Z
- a **player function** P that assigns to each non-terminal history (each member of $H \setminus Z$) a member of N

II.iii Equilibrium

There are several approaches to solving games. Solutions are optimal combinations of strategies that ensure the best outcome for a given or all players.

For strategic games, different game-theoretic solution concepts include **maximax** (maximise one's payoff), **maximin** (maximise one's minimum payoff, a.k.a. choosing the best of the worst possible outcomes), and **minimax** (minimise one's maximum loss). In a two-player zero-sum game, when the matrix has a **saddle point**, meaning a given action pair yields the best outcome for both players (*i.e.*, neither could do any better), the maximin and minimax strategies produce the same result. Thus, one can define **Nash equilibrium** for a strategic game $\langle N, (A_i), (\succsim_i)\rangle$ as the solution where no player has an incentive to change strategy given that no one else does, and express it formally as a profile $a^* \in A$ for every player $i \in N$ so that $(a^*_{-i}, a^*_i) \succsim_i (a^*_{-i}, a_i)$ for all $a_i \in A_i$.

When there is no saddle point, one can use the notion of equilibrium defined in terms of **mixed strategies** [19]. This involves randomising one's action selection with weighted probabilities, which ensure statistically optimal outcomes. In other words, each player makes the other indifferent between choosing one action or another, so neither player has an incentive to try another strategy. As a practical example, one can use the previously seen game of *Matching Pennies*. One assumes that player x plays heads with probability p and tails with probability $1 - p$. Similarly, player y plays heads with probability q and tails with probability $1 - q$. According to the payoff values in Figure 2, x's rewards will be $1 \cdot q - 1 \cdot (1 - q)$ for playing heads and $-1 \cdot q + 1 \cdot (1 - q)$ for playing tails. In equilibrium, player x is willing to randomise only when she is indifferent between heads and tails. Thus, the two equations must be equal, yielding $q = \frac{1}{2}$. Following identical reasoning for player y, one obtains $p = \frac{1}{2}$. Therefore, both players will play heads or tails with a probability of $\frac{1}{2}$. This is a simple if trivial example, but mixed strategies, if allowed, can always guarantee an equilibrium.

For an extensive game $\langle N, H, P, (\succsim_i)\rangle$, and having defined $O(s)$ as the outcome of a strategy profile $s = (s_i)_{i \in N}$, the Nash equilibrium will be the strategy profile s^* for every player $i \in N$ such that $O(s^*_{-i}, s^*_i) \succsim_i O(s^*_{-i}, s_i)$ for every strategy s_i of player i. Mixed strategies can work analogously to what is seen in strategic games, whereby a Nash equilibrium in mixed strategies for extensive games can be expressed as a profile σ^* of mixed strategies so that $O(\sigma^*_{-i}, \sigma^*_i) \succsim_i O(\sigma^*_{-i}, \sigma_i)$ for every mixed strategy σ_i of player i.

However, because of the sequential nature of extensive games, the notion of **subgame perfect equilibrium** is introduced. This is a refinement of the Nash equilibrium defined above which accounts for history-dependent best responses so that, for every non-terminal history $h \in H \setminus Z$ for which the player function is $P(h) = i$, $O(s^*_{-i}|_h, s^*_i|_h) \succsim_i O(s^*_{-i}|_h, s_i|_h)$ for every strategy s_i of player i, for the subgame $G(h)$. Normally, subgame perfect equilibrium is obtained using **backwards induction**: starting from the terminal history, one finds the best response strategy profiles or the Nash equilibria in the subgame, assigns these strategy profiles and the associated payoffs to the subgame, and moves successively towards the beginning of the game.

The next section looks at how Xenakis leveraged game-theoretical concepts to design musical game pieces.

III Xenakis's Game Pieces

Xenakis's trilogy is based on zero-sum games with complete and perfect information: each player, when making a decision, has knowledge of all the events that have previously occurred (*i.e.*, actions taken by the opponent are observable). Xenakis expresses all three games as strategic games using the normal form, and he provides mixed strategy calculations since these game matrices have no saddle points.

The details for each zero-sum game piece in Xenakis's trilogy follow.

III.i *Duel*

Duel is described in Chapter IV of *Formalized Music* [32], where the reader is referred for finer details. This game piece sees two conductors and their respective orchestras (hereinafter x and y, in keeping with the convention adopted thus far) competing against each other via means of juxtaposing musical events (or *tactics*, in Xenakis's choice of terms). Said tactics are chosen based on combinations of pairwise actions (*i.e.*, the payoff matrix) associated with an aesthetic outcome value stipulated by the composer. Details on the musical instructions for these events are omitted here for the sake of brevity. The payoff matrix undergoes several transformations to ensure a fair game, and its final form is shown in Figure 3.

		y						
		I	II	III	IV	V	VI	
x	I	-1	+1	+3	-1	+1	-1	$\frac{14}{56}$
	II	+1	-1	-1	-1	+1	-1	$\frac{6}{56}$
	III	+3	-1	-3	+5	+1	-3	$\frac{6}{56}$
	IV	-1	+3	+3	-1	-1	-1	$\frac{6}{56}$
	V	+1	-1	+1	+1	-1	-1	$\frac{8}{56}$
	VI	-1	-1	-3	-1	-1	+3	$\frac{16}{56}$
		$\frac{19}{56}$	$\frac{7}{56}$	$\frac{6}{56}$	$\frac{1}{56}$	$\frac{7}{56}$	$\frac{16}{56}$	

Figure 3: Duel*'s payoff matrix and associated weights of the mixed strategies.*

The mixed strategy Nash equilibrium for *Duel* can be calculated by determining the probability corresponding to each strategy so that x is indifferent to the actions of y, and vice versa. Each conductor might select the next event based purely on these probabilistic weights. For example, x will pick event I with a probability of $\frac{14}{56}$, and so forth.

III.ii *Stratégie*

Like in *Duel*, there are 6 possible tactics to choose from, but their musical content differs in *Stratégie*. Conceptually an extension of *Duel*, *Stratégie* augments the scope of the score by allowing combinations of 2 or 3 tactics for each conductor. These yield 19 possible combinations for each conductor to choose from and 361 possible pairs of such composite events. The resulting matrix is omitted here. In addition to this matrix, *Stratégie* also provides two additional 3×3 matrices that, by aggregating two-by-two and three-by-three compatible combinations of fundamental tactics, reduce the decisional complexity for the conductors.

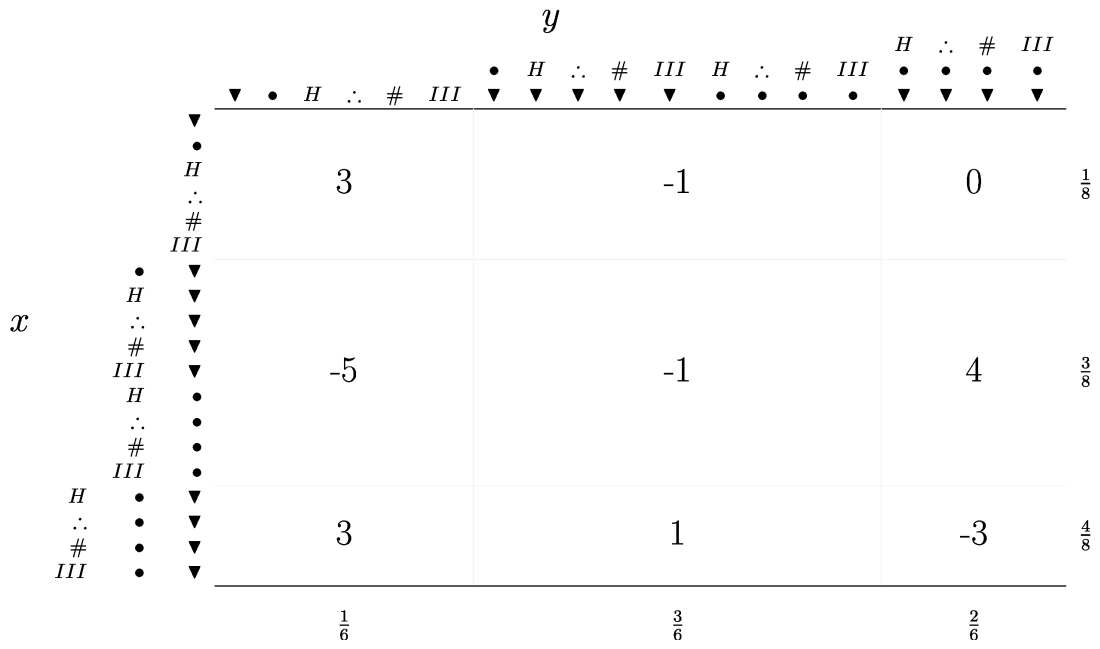


Figure 4: *One of* Stratégie*'s* 3×3 *payoff matrices and associated weights of the mixed strategies.*

III.iii *Linaia-Agon*

This piece is inspired by mythological tales of a human musician, Linos, challenging the god Apollo. *Linaia-Agon* is a three-part work, of which the second is structured around zero-sum games of strategy, not unlike those seen so far. This part comprises two games: *Choice of Combats* and *Combats*. Figure 5 shows the former on the left and one of the combats on the right, with Linos (played by a trombone) indicated as x and Apollo (played by a tuba) marked as y, for the sake of consistency with earlier seen game matrices. In *Choice of Combats*, the available tactics are indicated as α, β, γ and correspond to specific notes (different for trombone and tuba). The sequence of their occurrence in the game then goes to determine the following part (*i.e.*, *Combats*), where each of these three Greek letters corresponds to a new game matrix, each with different payoffs (and resulting mixed strategy Nash equilibrium) and instrumentation (only for Apollo). In these three games, there are four tactics that one can choose.

Xenakis provides instructions and guidelines for playing his games of musical strategy. If carefully examined, however, these explanations, along with additional evidence gathered from interviews containing direct references or notions relating to the trilogy under scrutiny, might

x \ y	α	β	γ	
α	-3	-8	7	$\frac{4}{15}$
β	2	2	-3	$\frac{10}{15}$
γ	-8	12	2	$\frac{1}{15}$
	$\frac{6}{15}$	$\frac{3}{15}$	$\frac{6}{15}$	

x \ y	$\backsim$	$\vdots$	$/$	$\varnothing$	
$\backsim$	1	-2	-2	3	$\frac{92}{335}$
$\vdots$	1	-1	0	-2	$\frac{78}{335}$
$/$	0	4	-2	-2	$\frac{89}{335}$
$\varnothing$	-2	-1	5	1	$\frac{76}{335}$
	$\frac{161}{335}$	$\frac{61}{335}$	$\frac{72}{335}$	$\frac{41}{335}$	

Figure 5: *On the left:* Linaia-agon*'s payoff matrix for the* Choice of Combats *game. On the right:* Linaia-agon*'s* β *game matrix, one of the three* Combats*. Tactics symbols and corresponding musical content are as follows:* $\backsim$ *indicates frequency and amplitude modulation,* $\vdots$ *stands for staccato articulation,* $/$ *means glissandi, and* $\varnothing$ *was originally left null in Xenakis's score but is here introduced to denote silence.*

reveal contradictions and incongruencies. The points of contention are discussed in the next section.

IV Insight

Although discrepancies between the theoretical axioms and the score implementations have been flagged [2] for other works of Xenakis, similar discussions concerning his game pieces have not, to the author's knowledge, been had as yet. This section aims to provide deeper insight and a critical understanding of Xenakis's games of musical strategy.

IV.i Game Formalism

Perhaps the most important issue in Xenakis's trilogy concerns the formal definition of the game model. In particular, there is little clarity with respect to the time dimension (as defined in Section II.i). In the original description of *Duel* and *Stratégie*, Xenakis states that the payoffs refer to "couples of simultaneous events" [32, p.114] and again that "pairs of tactics are performed simultaneously" [32, p.126]. However, in Figure IV-4 [32, p.126], reproduced here in Figure 6, one can see that the tactics in a pair are instead asynchronous: one conductor starts by choosing event i and the other conductor responds with event j. This is further corroborated when, after having decided who is x and who is y by means of a coin toss, "deciding who starts the game is determined by a second toss" [32, p.126].

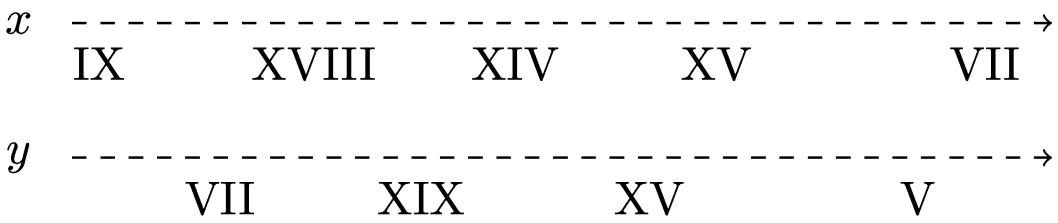


Figure 6: *A reproduction of Xenakis's original Figure IV-4 in* Formalized Music*. Note: in this version, only the temporal sequence of tactics is displayed, whereas corresponding payoffs are omitted.*

Moreover, when discussing how to award points, Xenakis suggests "to have one or two referees counting the points in two columns, one for conductor X and one for conductor Y, both in positive numbers". As already explained, however, this is a zero-sum game meaning that one's gains are the other's losses. Therefore, this instruction is somewhat confusing. Another option for awarding points is to use "an automatic system that consists of an individual board for each conductor". In this case, Xenakis explains that, for example, "if conductor X chooses tactic XV against Y's IV, he presses the button at the intersection of row XV and column IV". The use of the word "against" suggests that one conductor selects a tactic conditioned on the tactic of the opponent. So far, all evidence points to sequential games in extensive form. The choice of representation is not a reason for debate, since it is possible to convert from normal to extensive forms, as seen in Section II.i. However, univocal clarity about the time dimension of Xenakis's game pieces becomes problematic because in sequential games players who move later in the game can condition their choices on observed moves made earlier in the game. Conversely, in simultaneous games, players must all choose their own strategies without knowing what strategies are chosen by other players. These two types of games are solved differently, based on this information dependency, as seen in Section II.iii.

Xenakis's games are **repeated**, and at each repetition (hereinafter, **stage game**) both conductors are presented again with the entire game matrix, thus, effectively, starting anew. Repeated games are normally described as extensive form games where each stage game is modelled on a normal form. That is, each stage game is considered either as a strategic game or as an extensive form game with imperfect information derived from the normal form. As Xenakis's games have been shown to be sequential, it can be concluded that they should be classified as finitely repeated sequential games with perfect and complete information, or, more succinctly, **finitely repeated extensive games**. To summarise:

- The strategic relationship between players is expressed by a normal form (the payoff matrix)
- Players play an extensive form game (a sequential game) derived from the normal form
- Payoffs are handed out after every stage game
- Every stage game is the same in every stage
- One can find the minimax point using the normal form representation of the stage game and define the subgame perfect equilibrium in the corresponding repeated game

Having formally defined the game model for Xenakis's trilogy, the next step is to look at the decision-making processes that might provide viable solutions.

IV.ii Decision-making

Xenakis discusses decision-making strategies explicitly when referring to *Stratégie*, but these equally apply to *Duel*. He provides the following options:

1. arbitrary choice
2. *a priori* agreement on a sequence of action pairs
3. "drawing from an urn containing balls [. . .] in different proportions"
4. eliminating one conductor (the remaining one directs both orchestras)
5. conditioning on "the winnings or losses contained in the game matrix"

Option (3) is effectively equivalent to abiding by the mixed strategy Nash equilibrium. Arguably, probabilistic pooling is a difficult task for the average folk: randomising using weighted probabilities would require some aid, which is what Xenakis suggests (*i.e.*, urns). The composer further recommends that this be done offline, before the performance, and adequately rehearsed. However, this option, along with (1), (2) and (4), is disregarded. This is somewhat strange given that a good part of the chapter dedicated to *Duel* and *Stratègie* is spent on optimisations based on randomisation strategies. Except for the last option, all these decision-making methods are deemed unsuitable and termed *degenerate*, lacking "any conditioning for conflict, and therefore without any new compositional argument" [32, p.113]. One is thus left with subgame perfect equilibrium either via a) backward induction or b) choosing the best response at each stage game and disregarding the history up to that point. Whether to use one or the other is largely dependent on how the game is going to end. Xenakis provides three options to limit the game: based on a fixed number of stage games (histories), on a fixed cumulative payoff value or on a fixed time length.

To use backward induction to calculate the subgame perfect equilibrium, it is necessary to know the terminal history Z. There could be two ways to determine the number of histories. In one case this would be decided on the spot, just before starting the game. This would put conductors under considerable strain as they would have to be apt in rapid backward induction calculations. Otherwise, the number of histories could be determined before the start of the game, leaving ample time to calculate the subgame perfect equilibrium using backward induction *a priori*. Then, the conductors would simply follow the sequence of action pairs in H during the performance. This case would reduce the decisional power of the conductors to solely choosing a time for the next action, and it would arguably be at least as dismissive of "conditioning for conflict" as the other degenerate options listed earlier.

If the number of stage games is unknown (*i.e.*, the game ends upon reaching a stipulated cumulative payoff or time value) and conductors are advised not to use mixed strategies (see above), then they must simply select the action that yields the best outcome in each stage game. This is akin to a memoryless system since each conductor only considers the current state to deliberate. Besides offering an impoverished notion of decision-making agency (arguably not particularly skilful in dealing with expectation or insightful in the opponent's ways), purely reacting at each decision node without knowledge of the past highlights another problematic aspect of Xenakis's game design. That is, there might be times when having to choose between equivalent payoffs for different action pairs in the game matrix, as in the case of *Duel* (see Figure 3) and *Linaia-Agon*'s *Choice of Combat* (see Figure 5). How would then a conductor break these ties? Random selection must be excluded since it is *degenerate* according to Xenakis (see above). What about personal preference regarding the aesthetic value of candidate actions or action pairs? This would seem reasonable, although in conflict with the aesthetic top-down control advocated when one is told that "I am the judge - the one who determines which solution is more interesting" [28, p.108].

Finally, it must be noted that a subgame perfect equilibrium requires that, regardless of what players observe, they will continue to maintain the original assumptions that the opponent is 1) rational 2) knows the game or perceives it identically to how it has been specified and 3) does not make mistakes. If the sole aim was to keep with a less lenient approach that foregrounds the mathematical foundations of the game piece over an ill-defined utility function in the music domain, then computational conductors could be employed. To investigate this option, the next section offers a simple simulation using computational conductors that can learn optimal strategies.

V Experiment: *Duel*

Duel is considered as a case study. The experiment presented here is not a mere replication of that conducted by Sluchin & Malt [25] because it involves computational decision-making agents that are able to learn how to solve *Duel* via either minimax or mixed strategies. The game is treated as a simultaneous repeated game, allowing the conductors to continue playing it as many times as necessary to learn the best response. A conductor is modelled using Reinforcement Learning (RL), one of the three main paradigms used in machine learning (the others being: supervised and unsupervised learning) and the most widely applied to gaming problems, in general. RL involves a loop whereby agents in an environment take actions and receive feedback from some reward function, thus incrementally learning to optimise their cumulative rewards. RL is normally modelled as a Markov Decision Process (MDP): at each discrete time step t, the agent receives the current state s_t and a reward r_t, chooses an action a_t from the set of available actions A, which modifies the environment to a new state s_{t+1} and yields a new reward r_{t+1} based on the transition (s_t, a_t, s_{t+1}). Through this iterative process, the learning agent aims to learn a policy: $\pi : A \times S \rightarrow [0, 1], \pi(a, s) = \Pr(a_t = a \mid s_t = s)$ that maximises the expected cumulative reward. Arguably, RL is very suitable for modelling game-theoretical tasks [20].

V.i Q-Learning

Q-learning [30] is a model-free RL algorithm, meaning that it does not use the transition probability distribution associated with an MDP, but it is rather a trial-and-error approach. Q-learning was chosen for the experiment presented in this section because it is applicable exclusively to discrete action and state spaces, which is the case of Xenakis's game pieces. Q stands for the quality of a state-action combination, and it is expressed as $Q : S \times A \rightarrow \mathbb{R}$ using a value iteration update equation that accounts for the weighted average of the old value and the new information, as well as other parameters such as a learning rate α and a discount factor γ. Simply put, the Q-learning algorithm works as follows: initialise a Q-table ($n \times m$ with n = number of actions and m = number of states), choose an action, measure the reward, update the Q-table, repeat.

V.ii Tournament

For this experiment, three types of conductors are used: one that learns the mixed strategy Nash equilibrium using Q-learning (hereinafter, *NashQ*), another that learns the minimax strategy using Q-learning (from now on referred to as *MinimaxQ*), and yet another which does not learn but simply selects tactics at random (hence, called *Random*). The simulation was implemented using the Python[1] programming language with few additional dependencies. These included the *nashpy* library[2] which offers different algorithms for the calculation of the Nash Equilibria, such as *vertex enumeration* [29] or the Lemke & Howson [17] method. Learning conductors were instantiated with a default value of $\gamma = 0.99$. In this case, a tournament comprises all possible combinations of conductors, with repetition (*e.g.*, $[A, A], [B, B]$, etc.) but with order invariance (*e.g.*, $[A, B] == [B, A]$). One instance of a repeated game between any given pair will hereafter be referred to as an *epoch*, whereas one repetition of the game in an epoch will be called an *episode*.

Figure 7 shows one epoch in a tournament, between a NashQ and a MinimaxQ, while Figure 8 shows the duel between two NashQ. Figure 9, instead, shows a duel involving a Random conductor, as a comparison baseline. From these plots, it is possible to note that, after an initial

[1] https://python.org
[2] https://pypi.org/project/nashpy/

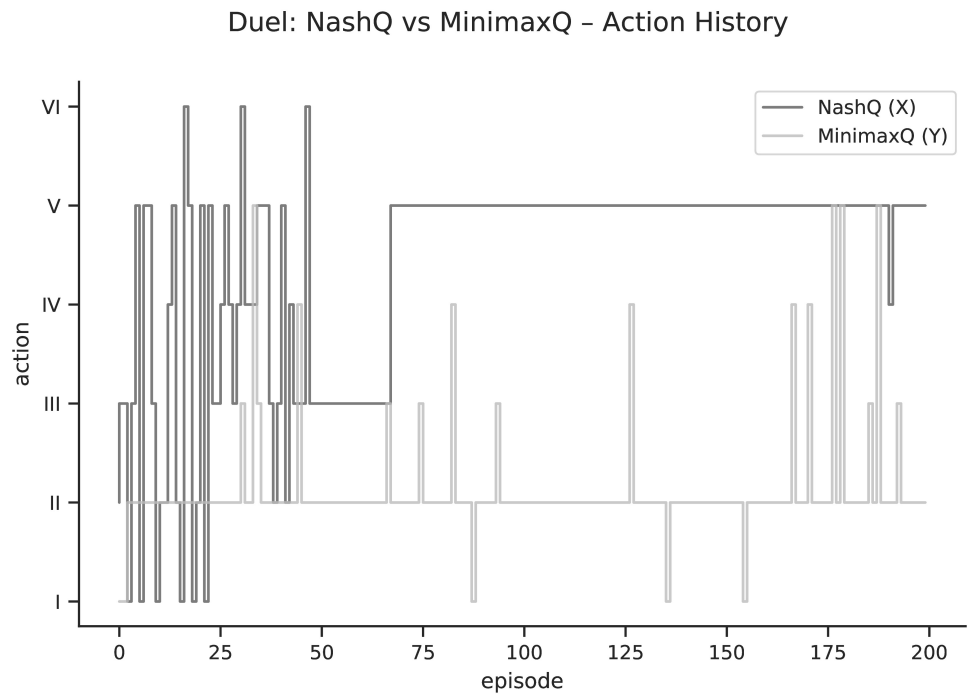


Figure 7: *Duel between a NashQ conductor and a MinimaxQ opponent, playing a repeated game of 200 episodes.*

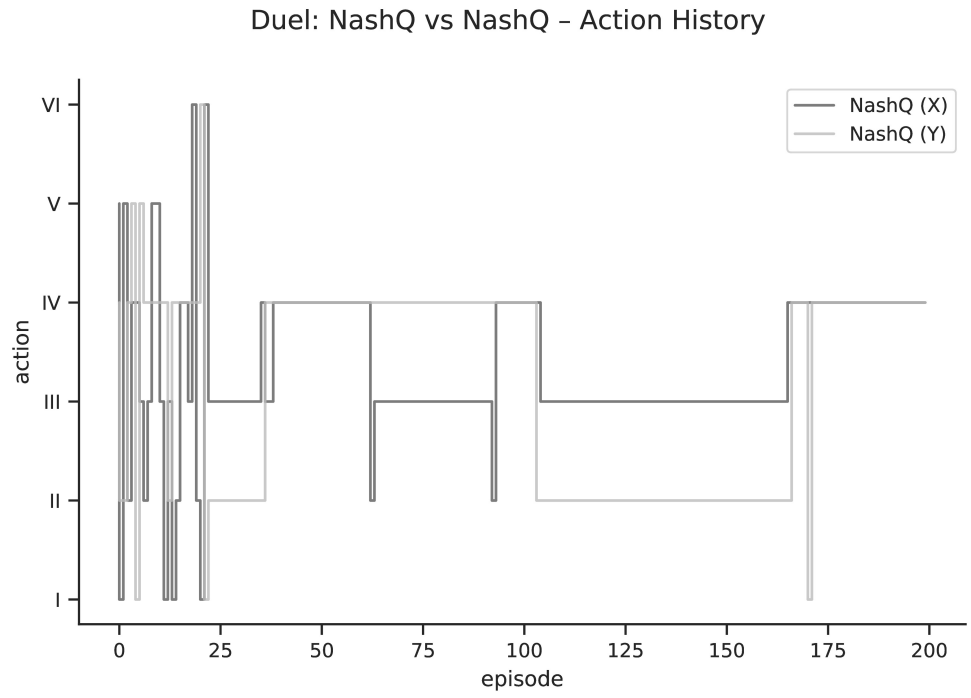


Figure 8: *Duel between two NashQ conductors, playing a repeated game of 200 episodes.*

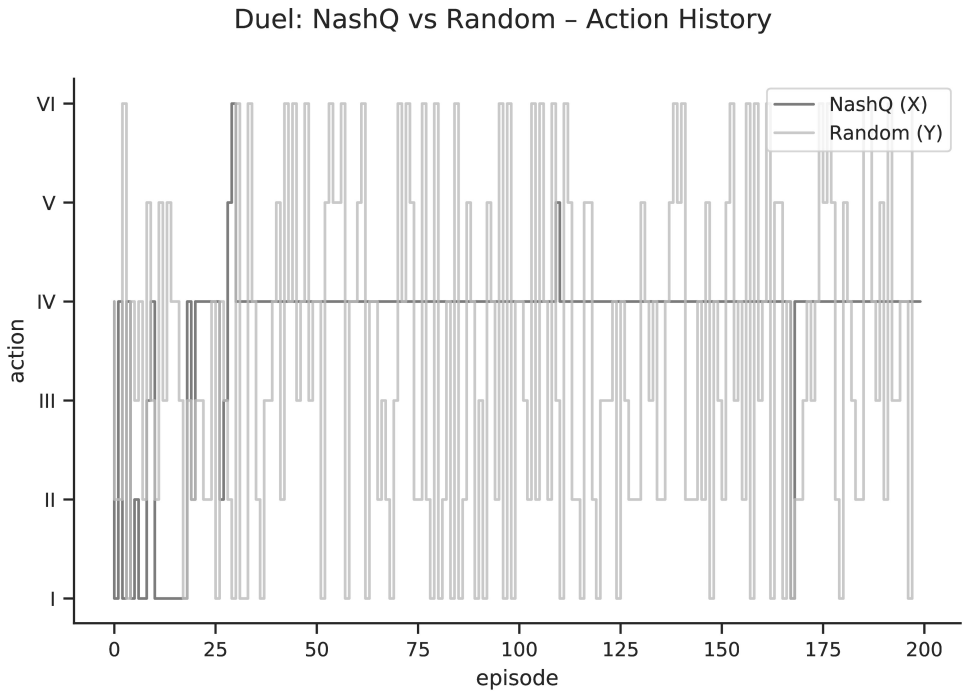


Figure 9: *Duel between a NashQ conductors and a Random conductor, playing a repeated game of 200 episodes.*

phase of learning, the conductors converge towards stable strategies, meaning that they end up playing the same tactic for many consecutive episodes.

In a real-performance scenario, how would these learnt behaviours and policies potentially affect the resulting music? Although there is general consensus on the basic principles of design in visual arts [1], the same cannot be said for musical design and composition. However, the notion of *contrast* (or variety) seems to be a constant among all the arts, including music. Using contrast as an evaluation metric, would imply that learned, informed conductors might be liable for considerably lower musical contrast (boring music, some might say?), whilst being numerically optimal.

VI Reflection & Future Work

It is hopefully clear by now how the choice of an applied mathematics framework gives rise to some interesting disjunctures. The axioms of game theory, which presuppose a well-defined utility function, can be challenging to uphold when artistic, creative, and aesthetic goals compete with theoretical ones. One must then consider that the intrinsic value of a game-based framework resides in the possibility to subvert its rational/theoretical axioms in favour of more palatable, musically pleasing, or desirable outcomes. Doing so allows x or y to make sub-optimal or biased choices in terms of payoff that are instead optimal from an artistic (subjective) viewpoint. Given the current involvement of computers in creative and musical tasks, it is important to contextualise the lessons learned so far. Solving Xenakis's game matrices is a trivial task. Simulating *naturalistic decision-making* [16] that might communicate a sense of negotiation between aesthetic or musical concerns and the mathematical imperatives of the game, on the other hand, is not. To this end,

when designing computational versions of Xenakis's zero-sum games of strategy, one could deliberately introduce some sub-optimality [15], even opening up to simple options such as finite state machines. In this case, decision policies could stochastically include not only *mixed strategies* and *minimax*, but also subversive states based on the simulation of some aesthetic preference, for example, or some intrinsic attitude trait (*e.g.*, impulsive, greedy, rational, etc.). Employing simple approaches such as automata might sound naïve in an age of large language models with billions of parameters [5] and an in-depth discussion of the best approaches to simulate a process as complex as human decision-making is beyond the scope of this article (and possibly beyond the scope of machine learning techniques to date). Thus, these speculations are merely included as a springboard for further exploration of Xenakis's game pieces. For example, paradigms such as interactive evolutionary computation [27], whereby the reward function (the fitness function, in this case) is updated according to the feedback given by a human meta-conductor, could be leveraged. Furthermore, and particularly given the recent shift toward increased online presence due to the COVID-19 pandemic, telematic, remote implementations of Xenakis's game pieces could also be envisaged. These are just a few re-imagined incarnations of Xenakis's trilogy among the endless possibilities available.

VII Conclusion

This article considered the games of musical strategy composed by Xenakis between 1959 and 1972 and discussed philosophical, musical, and mathematical properties of these works. In identifying a problematic disparity between the game-theoretical models and their rendition in Xenakis's game-based works, the relationship between rational decision-making and the notion of payoff or reward in a musical context foregrounded further incongruencies. It was posited that a payoff matrix alone is not a sufficient incentive or motivation to resolve musical conflict and that, if one wishes to use applied mathematics for musical interaction in a strict fashion, disappointments regarding both numerical solutions and musical texture are likely to emerge. These claims were substantiated by an in-depth probe at a theoretical level, accompanied by supporting evidence offered in the form of direct references in Xenakis's game description and instructions. More evidence was gathered via a simulation of *Duel*, where computational learning agents took on the role of conductors and battled in an extensive tournament. Said agents were modelled using Reinforcement Learning to optimise their responses over time, based on different solutions (namely mixed strategies and minimax). An additional agent that responded arbitrarily (without learning) was also used as a comparison baseline. In summary, this article attempted to show that the experience of playing a game of musical strategy is a complex phenomenon where disparate factors converge, transcending the pure mechanics stipulated by the mathematical model of the game. Arguably, a real conductor, even if appropriately informed about the axioms of the game and its solutions, would still exhibit behaviours based not only on the utility function, but (presumably) also on aesthetic preference, feedback from the audience, the orchestra(s), the acoustic environment, and so forth. Notwithstanding the limitations of the mathematical framework that this article investigated, and given the new potential provided by machine learning and networked performance, game theory-based pieces continue to be exciting vehicles for structuring musical interaction and design that can open up to novel and unforeseen modalities of musical expression.

References

[1] Bang, Molly (1991). *Picture This:* Perception & Composition. Boston: Bulfinch Press.

[2] Bayer, Francis (1981). *De Schoenberg à Cage*. Paris: Kliencksieck.

[3] Beguš, Jelena Janković (2016). Playing the game with aleatorics and narrativity: *Linaia-Agon* by Iannis Xenakis. *New Sound*, v. 48, n. 2, pp. 109–130.

[4] Brackett, John (2010). Some Notes on John Zorn's Cobra. *American Music*, v. 28, n. 1, pp. 44–75.

[5] Brown, Tom; *et al* (2020). Language models are few-shot learners. In: Larochelle, H. *et al* (Ed.), *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901.

[6] Caillois, Roger (1961). *Man, Play and Games*. New York: Free Press of Glencoe.

[7] Cox, Christoph (2004). The Game Pieces. In: *Audio Culture:* Readings in Modern Music. New York: Continuum.

[8] DeLio, Thomas (1987). Structure and strategy: Iannis Xenakis' Linaia-Agon. *Interface*, v. 16, n. 3, pp. 143–164.

[9] Havryliv, Mark; Vergara-Richards, Emiliano (2006). From Battle Metris to Symbiotic Symphony: A New Model For Musical Games. In: *Proceedings of the 2006 International Conference on Game Research and Development*, Perth, Australia, pp. 260–268.

[10] Huizinga, Johan (1955). *Homo Ludens.* A study of the play-element in culture. Boston: Beacon Press.

[11] Kagel, Mauricio (1964). Match. Universal Edition.

[12] Kalonaris, Stefano (2016). Markov Networks for Free Improvisers. In: *Proceedings of the* 42^{nd} *International Computer Music Conference*, Utrecht, The Netherlands, pp. 181–185.

[13] Kalonaris, Stefano (2017). Adaptive specialisation and music games on networks. In: *Proceedings of the* 13^{th} *International Symposium on Computer Music Multidisciplinary Research*, Matosinhos, Portugal, pp. 420–429.

[14] Kalonaris, Stefano (2018a). Beyond Schemata in Collective Improvisation: A Support Tool for Music Interactions. *Leonardo Music Journal*, v. 28, n. 1, pp. 34–37.

[15] Kalonaris, Stefano (2018b). Satisficing goals and methods in human-machine music improvisations: Experiments with *Dory*. *Journal of Creative Music Systems*, v. 2, n. 2.

[16] Klein, Gary A.; *et al* (1993). *Decision Making in Action:* Models and Method. New York: Ablex.

[17] Lemke, Carlton E.; Howson, Jr. Joseph T. (1964). Equilibrium points of bimatrix games. *Journal of The Society for Industrial and Applied Mathematics*, v. 12, n. 2, pp. 413–423.

[18] Liuni, Marco; Morelli, Davide (2006). Playing Music: An installation based on Xenakis' musical games. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, New York, NY, USA, pp. 322–325.

[19] Morgenstern, Oskar; von Neumann, John (1947). *The Theory of Games and Economic Behavior*. Princeton: Princeton University Press.

[20] Nowé, Ann; Vrancx, Peter; De Hauwere, Yann-Michaël (2012). Game theory and multi-agent reinforcement learning. In: M. Wiering & M. van Otterlo (Eds.), *Reinforcement Learning:* State-of-the-Art pp. 441–470. Berlin, Heidelberg: Springer Berlin Heidelberg.

[21] Osborne, Martin J.; Rubinstein, Ariel (1994). *A Course in Game Theory*. Cambridge: MIT Press.

[22] Shadow-Sky, Mathius (1980). Ludus Musicae Temporarium. http://centrebombe.org/livre/1980.b.html. Accessed: December 22nd, 2022.

[23] Sluchin, Benny (2005). Linaia-Agon: Towards an Interpretation Based on the Theory. In: *Proceedings of the International Symposium Iannis Xenakis*, pp. 299–311.

[24] Sluchin, Benny (2015). Linaia-Agon By Iannis Xenakis: Surpassing One's Limit / Le Dépassement De Soi. https://moderecords.com/catalog/284-xenakis/. Accessed: February 14th, 2022.

[25] Sluchin, Benny; Malt, Mikhail (2011a). Open form and two combinatorial musical models: The cases of Domaines and Duel. In: C. Agon *et al* (Ed.), *Mathematics and Computation in Music* pp. 255–269. Berlin, Heidelberg: Springer Berlin Heidelberg.

[26] Sluchin, Benny; Malt, Mikhail (2011b). Play and game in Duel and Strategy. In: *Proceedings of the Xenakis International Symposium*.

[27] Takagi, Hideyuki (2001). Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE*, v. 89, n. 9, pp. 1275–1296.

[28] Varga, Bálint András (2003). *Conversations with Iannis Xenakis*. London: Faber & Faber.

[29] von Stengel, Bernhard (2007). Equilibrium Computation for Two-Player Games in Strategic and Extensive Form. In: N. Nisan *et al* (Ed.), *Algorithmic game theory* pp. 53–78. Cambridge: Cambridge University Press.

[30] Watkins, Christopher J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. Thesis, Kings College.

[31] Xenakis, Iannis (1972). Duel. Éditions Salabert.

[32] Xenakis, Iannis (1992). *Formalized music:* thought and mathematics in composition. Hillsdale: Pendragon Press.

Building a Knowledge Base of Rhythms

Charles Ames
info@charlesames.net
Orcid: 0000-0001-7263-3518
DOI: 10.46926/musmat.2022v6n2.72-94

Abstract: *This article is about composing programs. It explores generating up-front a knowledge base of rhythms as an alternative to generating rhythms on the fly. Since the knowledge base contains much more information than any one composing program will ever actually use, it employs ISAM technology to persist the information in one random-access file. The fundamental rhythmic entities are monophonic patterns and polyphonic textures. Patterns are defined as successions of pulse events (rests, attacks, ties). Textures present multiple patterns simultaneously. Various entity properties are calculated up front and stored alongside content data. For textures, these properties include an attacks profile, which is a vector. A persistent lookup map is realized to efficiently identify textures sharing attacks profiles in common. A pattern-to-pattern comparison identifies relationships which are documented in an ISAM map; these pattern-to-pattern relationships are then used to build a random-access map of texture-to-texture relationships. The article closes with a series of applications demonstrating how rhythms may be selected by combining knowledge-base queries with random shuffling and constraint filtering.*

Keywords*: Composing program. Rhythm knowledge base. ISAM.*

I. Introduction

Rhythm is the foundation of my music-theoretical thinking. To give a visual analogy, rhythm provides the shapes which pitch illuminates with color. Pretty much all of my composing programs lay out the rhythm first and fill in the pitches later. That applies to programs which emulate familiar styles and also to programs which explore pan-chromaticism and emancipated dissonance.

Most of my programs have taken an on-the-fly approach to generating rhythms. For example, my 1987 "Cybernetic Composer"[1] generated rhythms for rock, 2 jazz styles, and ragtime using context-sensitive grammars which divided long durations into shorter ones. Three of the four styles alternated 'composed' tunes with 'improvisatory' material; the 'composed' sections followed templates which generated 'fresh' rhythms for some segments and copied earlier rhythms for other segments. One feature of the rock style was an improvisatory break during which the accompaniment rested while the lead played through.

Received: October 21st, 2022
Approved: December 21st, 2022

[1]The "Cybernetic Composer" is described on my site at https://charlesames.net/cybernetic-composer/index.html.

You can judge from the examples provided on my site whether these grammars were effective; however, the site examples were selected to present my work in the best light. The two rock examples happen to present fairly active material for their improvisatory breaks, but this didn't always happen. The rhythm generator basically worked its way down the decision tree by flipping coins, and I was unable at the time to figure out how to incline those decisions toward more active results. Later it occurred to me that however many different rhythms my grammars were capable of producing, it would still be practical to run an enumeration algorithm that would list all candidates up front. Having that, my program could assign an activity score (attacks per beat) to each candidate. But by then the *Age of Intelligent Machines* exhibit was already on the road.

The idea of enumerating material up front stuck with me. I took a stab at it in the programs for my 1988 solo-violin piece, "Concurrence" [1]. Here the rhythmic material comprised different ways of dividing a quarter note into sixteenths, given that the instrument could do any of four things in each sixteenth: Initiate a new note, tie from a previous note, slur from a previous note, or rest. But this time around my concern wasn't for the qualities of specific patterns but rather for rather the degree of similarity between any two patterns. This was measured by calculating the minimum number of primitive operations required to transform one pattern into the other. The rhythm-selecting programs worked from a template which laid out the compositional form as a sequence of nodes, one per quarter note. These nodes were connected with references that indicated "this node is similar to that previous node" or "this node contrasts with that previous node".

The present effort generalizes what I did for "Concurrence" into a knowledge base of rhythms. First off, the idea of dividing quarter notes into sixteenths generalizes into the idea of dividing some unspecified longer duration into pulses of nominally-equal shorter durations. Beyond that, the fixed-length, monophonic patterns of "Concurrence" are here extended to textures with two, three, or four pulses, layering up to three patterns polyphonically. Hovering over this effort has been the prospect of combinatorial explosion. Faced with this prospect, the distinction between freshly attacked notes and slurred-to notes had to be dropped. I found it necessary to limit the maximum pattern length to 4 and to fix the number of layered patterns at 3 (fewer layers are accommodated by allowing some layers to rest). Even under these restrictions it required some 4 days to build the file; most of this time being taken up by the process of gleaning texture-to-texture relationships.

At this writing I have only the vaguest idea what kind of pieces this knowledge base will be used for, only that there will be more than one (health permitting) and that the first of these pieces will employ additive rhythms. Since the knowledge base is intended for multiple use by myself and possibly others (if there's any interest), its design is neutral and generic.

II. ISAM Persistence

According to my production framework[2] I should be implementing the overall compositional process as a succession of stages, with earlier stages leaving XML-formatted files of data for later stages to pick up. However here I am creating a knowledge base of building blocks, which knowledge base is potentially usable by many projects. Therefore the data needs to be persistent (stored in a file) and random-access. Although the knowledge base may hold a great deal of material, only a limited subset will be brought in to any individual application. Also there will be a need to query the knowledge base for sets of entities which meet certain criteria or for sets of entities which bear a certain relationship to some entity already in hand.

[2] https://charlesames.net/glossary/production-framework.html.

As a retired database programmer my first instinct was to go the full SQL route, but SQL databases impose considerable overhead. The requirements just listed can instead be satisfied using *Indexed Sequential Access Method* (ISAM), a technology central to SQL databases which is also available in stand-alone implementations. The ISAM implementation I found to do this in **Java** was **MapDB**.[3] **MapDB** is "free and open-source" and at this writing is into its 3.X production release. Despite this history the documentation remains sketchy. The online **Javadoc** equivalent simply lists methods without either method or parameter descriptions. Explanatory documents are few and far between, and of those that I could find, the sample code that did what I wanted to do wouldn't even compile.

That's the bad news. The good news is that **MapDB** implements a `BTreeMap` class, which acts like **Java**'s `SortedMap` interface but which stores its data in a random-access file rather than in memory. Like a `SortedMap`, a `BTreeMap` pairs *keys* with associated *values*. You can store **Java** objects as `BTreeMap` values with the proviso that the stored objects cannot reference other objects. So if object A relies on object B you have to implement object B with a unique identifier (**MapDB** seems to like long integers). Then if you've pulled object A from its `BTreeMap`, and want to look at a B contained within A, you can get object B's identifier from object A and use the identifier to pull object B from the BTreeMap where object B resides.

In the SQL world I would have stored each class of object in a *database table* with the unique identifier as its *primary key*. To facilitate queries for objects I would have implemented *table indices* enumerating the relevant properties and which objects have them.

MapDB has no tables and hence no table indices. However one can construct a `BTreeMap` which serves a purpose similar to a table index. In my first attempt to do this I built the *key* from one or more object properties and used the unique object identifier (a long) as the *value*. This didn't work because a `Map` (both `SortedMap` and `BTreeMap` implement this interface) associates exactly one value with each valid key. My second attempt incorporated the unique object identifier as the rightmost key element. This also didn't work — the sample code provided for this situation wouldn't compile — however I did manage to find a successful kludge by packing key elements into long integers.

III. Entities

The `PulseEventType` enumeration lists three things that one layer within a texture can do during a pulse: *`REST`*, *`ATTACK`*, and *`TIE`*.

Instances of the `PulsePattern` class persist in the knowledge base. A `PulsePattern` describes a succession of pulses and what happens in each pulse. This succession is implemented as an array of `PulseEventType` elements. The elements of this array are accessed using a `position` index, while the `length` of a `PulsePattern` instance is the array length (i.e. the number of pulses). Each `PulsePattern` is identified by a serially-generated long integer. There is only one constraint in forming a `PulsePattern`: a *`TIE`* can never follow a *`REST`*.

Instances of the `PulseTexture` class also persist in the knowledge base. A `PulseTexture` layers multiple simultaneous `PulsePattern` instances, all of the same `length`. This is implemented as an array of long integers, whose elements are `PulsePattern` identifiers. The elements of this array are accessed using a `layer` index, while the `depth` of a `PulseTexture` instance is the array length (i.e. the number of overlaid patterns). Each `PulseTexture` is identified by a serially-generated long integer. The definition of a `PulseTexture` abstracts away the order of patterns within the texture. Thus if a `PulseTexture` contains patterns A, B, and C, that would apply to pattern A on

[3] https://MapDB.org/.

top, pattern B in the middle, and pattern C on the bottom. However it would also apply to pattern B on top, pattern C in the middle, and pattern A on the bottom.

A specific *statement* of a `PulseTexture` is obtained by combining the `PulseTexture` with a mapping from musical parts to texture layers. For any given `PulseTexture` of `depth` 3, up to six distinct statements are available, corresponding to the six permutations of the set $\{0, 1, 2\}$. Part-to-layer mappings are represented in the knowledge base using the `Permutation` class. `Permutation` instances combine an array of 3 ordinals with a long-integer identifier derived by packing the number of array elements (3) in the leftmost nibble and the individual ordinals in successive nibbles thereafter. The six `Permutation` instances persist in the knowledge base, though in practice they are cached into an in-memory map.

Both simple and compound texture statements can be represented using the `TextureStatement` class, whose components are pairings of `PulseTexture` instances with `Permutation` instances. The `TextureStatement` class effectively presents a two-dimensional array of `PulseEventType` elements, with a `part` index ranging from 0 to `depth`-1 and a `position` index ranging from 0 to `length`-1. Instances of the `TextureStatement` class do *not* persist in the knowledge base; however the end product of any session working with the knowledge base will typically be one or more `TextureStatement` instances.

Instances of the `Relation` class persist in the knowledge base because in order to get **MapDB** to work I needed to associate each `Relation` instance with a persistent long-integer identifier. A `Relation` instance combines a `RelationCategory` with an integer `offset`. `RelationCategory` is a software enumeration which among other things includes code to determine whether two patterns are related in that way. The `offset` depends upon the `RelationCategory`. For examples the `offset` for `RelationCategory.`*`ROTATE`* indicates how far to right shift, while the `offset` for `RelationCategory.`*`MASK`* indicates which pulse position is affected. The full `RelationCategory` enumeration is presented below.

The remaining class whose instances persist in the knowledge base is `Profile`. A `Profile` is array of integers which count how often a certain `PulseEventType` (or combination thereof) occurs simultaneously during the corresponding pulse in a `PulseTexture` instance. The elements of this array are accessed using a `position` index, while the `length` of a `Profile` instance is the array length (i.e. the number of pulses). `Profile` instances also include a long-integer identifier which packs the number of array elements (the `length`) in the leftmost nibble and the individual counts in successive nibbles thereafter.

IV. Scalar Properties

The most fundamental property of a `PulseTexture` is `length`. The most fundamental properties of a `PulseTexture` are `length` and `depth`.

There is a family of scalar properties which have absolute versions, which are integer counts, and relative versions. The relative version is a floating-point number calculated as the count divided by its upper limit. Examples for the following definitions will be drawn from `PulseTexture` #16520, which has the following content:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶▶ – –]
2	78	[– – ▶]

The content is represented pictographically: "▶" means *ATTACK*, "−" means *TIE*, and " " means *REST*.

i. Attacks

The `attacksCount` and `attacksRatio` are dual properties shared by `PulsePattern` and `PulseTexture` instances. For `PulsePattern` the `attacksCount` is the number of pulses with the *ATTACK* event type, for which the upper limit is the pattern length. For `PulseTexture` the `attacksCount` remains the number of *ATTACK* pulses, however the upper limit expands to `length` × `depth`.

Referring back to `PulseTexture` #16520, `PulsePattern` #40 has 2 attacks, `PulsePattern` #54 has 2 attacks, and `PulsePattern` #78 has 1 attack. Therefore `PulsePattern` #16520 has an `attacksCount` of $2 + 2 + 1 = 5$ attacks. The `attacksRatio` is $5/12 = 42\%$ of a possible 12.

ii. Coverage

The `coverageCount` and `coverageRatio` are also dual properties shared `PulsePattern` and `PulseTexture`. For a `PulsePattern` instance the `coverageCount` is the number of *ATTACK* pulses plus the number of *TIE* pulses, for which the upper limit is the pattern length. For `PulseTexture` the upper limit expands to `length` × `depth`.

Referring back to `PulseTexture` #16520, `PulsePattern` #40 has 3 non-rest events, `PulsePattern` #54 has 4 non-rest events, and `PulsePattern` #78 has 3 nonrests. Therefore `PulsePattern` #16520 has a `coverageCount` of $3 + 4 + 3 = 10$. The `coverageRatio` is $10/12 = 83\%$ of a possible 12.

The compliment to `coverageCount` is the count of rests, which for `PulsePattern` #16520 is 2 or 17% of a possible 12.

iii. Dispersion

The `dispersionCount` and `dispersionRatio` are dual properties of `PulseTexture` instances. The `dispersionCount` is the number of pulses with an *ATTACK* in at least one layer. The upper limit is `length`.

`PulseTexture` #16520 has 1 attack in pulse-position 0 (`PulsePattern` #54), 2 attacks in pulse-position 1 (`PulsePattern` #40 and #54), 1 attacks in pulse-position 2 (`PulsePattern` #78), and 1 attack in position 3 (`PulsePattern` #40). This gives a `dispersionCount` of 4 with a 100% `dispersionRatio` out of a possible 4.

The compliment to `dispersionCount` is the count of pulses without an *ATTACK* in any layer. For `PulsePattern` #16520 this is 0 or 0% of a possible 4.

iv. Imbalance

The `imbalanceCount` and `imbalanceRatio` are dual properties of `PulseTexture` instances. The `imbalanceCount` is calculated by iterating through the layers, determining the maximum and minimum number of attacks, then subtracting the minimum from the maximum. If this max-min is zero, all layers will share the same number of attacks. Otherwise at least one layer will have more than its fair share activity. The upper limit obtains when all attacks happen in the same layer.

For `PulseTexture` #16520 the maximum number of attacks is 2 (`PulsePattern` #40 and #54) and the minimum number of attacks is 1 (`PulsePattern` #78). Thus the `imbalanceCount` is $2 - 1 = 1$ and the `imbalanceRatio` is 25% of a possible 4. An `imbalanceCount` of 4 (100%) would have resulted if one pattern had 4 attacks and another of the remaining 2 patterns had no attacks.

The compliment to `imbalanceCount` is `length` − `imbalanceCount`. For `PulsePattern` #16520 this is $4 - 0 = 4$ or 100% of a possible 4.

V. Order of Patterns in Textures

The enumeration algorithm for `PulseTexture` instances iterates through all possible pattern IDs for layer 0. Layer-1 pattern IDs range from the layer-0 pattern ID upwards, while layer-2 pattern IDs range from the layer-1 pattern ID upwards. This allows the same pattern to appear twice or three times in the same texture, but prevents any two textures from presenting the same set of patterns in different permutations.

Anyone employing a `PulseTexture` instance as a building block for a musical passage will quickly need to determine which musical part will play which layer. Accepting the default order is generally not a desirable option. If the voices are co-equal then a better option would be to permute layers randomly. If the context is metric and one voice has a lead role with the others providing accompaniment, then it may be desirable to assign those layers which most coincide with strong beats to the accompaniment and give the more syncopated layer to the lead voice. This metric option requires strong-beat position data.

Yet another option is to find the permutation which orders the patterns from most to least active. To quantify the level of activity within a pattern I implemented a *beauty contest*[4] using the formula:

$$\text{pattern activity} = 16 \times \text{attacks} + \text{ties} + \text{rand}[0, 1]. \tag{1}$$

For example consider `PulseTexture` #16520:

- [▶ − ▶] contains 2 attacks and 1 tie. A random offset of 0.524 gives an activity score of 33.524.
- [▶ ▶ − −] contains 2 attacks and 2 ties. A random offset of 0.340 gives an activity score of 34.340.
- [− − ▶] contains 1 attacks and 2 ties. A random offset of 0.721 gives an activity score of 18.721.

The permutation which orders the patterns from most to least active is therefore $(1, 0, 2)$. Notice that the number of attacks always swamps the number of ties, while the random offset exerts influence only when two patterns share the same counts of attacks and ties.

VI. Profiles

I adhere to the premise that musical meter is established through the convergence of polyphonic attacks on strong beats and divergence of attacks on weak beats.[5] The rhythmic knowledge base described here does not itself favor 'metric' textures; however, it does provide a handle which can be used to identify them. This handle is the *attack profile*.

[4] Beauty contests are simpler among the merit-based problem-solving strategies discussed in [2]. The specific heading is "'Beauty Contest' and 'Blackboard' Models", pp. 75–76.

[5] This premise underlies an article by Karl Kohn [3]. Kohn showed me a renotation example during an undergraduate composition lesson; however the lesson slipped from memory until we met again a few years ago. Convergence of polyphonic attacks is also the premise underlying my "Complementary Rhythm Generator". The backstory given on my site (`https://charlesames.net/rhythm/index.html`) says that I came to the premise while studying the collected motets of Guillaume de Machaut. I now realize that Kohn's lesson primed me for this realization.

The *attacks profile* is a vector property of the `PulseTexture` class. It is an array of small integers (actually bytes) giving the number of simultaneous `ATTACK` events, by pulse `position`. When two textures have the same attack profile, they will be heard to have the same aggregate rhythm. If the profile divides into regular groups initiated by larger `ATTACK` counts, then this aggregate rhythm will be heard as meter.

Two textures having the same attacks profile is a binary relationship which establishes *equivalence classes* among `PulseTexture` entities. Since `PulseTexture` entities have short lengths it is possible to pack their profiles into the 16 nibbles of a long integer, then create a lookup map of `PulseTexture` instances using this packed profile as the most significant key element.

For example, `PulseTexture` #16520 has attack profile $[1,2,1,1]$. A query of the lookup map returns 756 `PulseTexture` instances sharing the same attack profile. These results include `PulseTexture` #16520. Here is a random sampling of other textures returned by the same query:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶ ▶ – –]
2	78	[– – ▶]

`PulseTexture` #15176

Layer	PatternID	Content
0	39	[▶ –]
1	44	[▶ ▶]
2	69	[– ▶ ▶]

`PulseTexture` #11757

Layer	PatternID	Content
0	36	[▶▶]
1	40	[▶ – ▶]
2	60	[▶ – –]

`PulseTexture` #17033

Layer	PatternID	Content
0	41	[▶ – –]
1	42	[▶]
2	71	[– ▶▶▶]

`PulseTexture` #15097

Layer	PatternID	Content
0	39	[▶ –]
1	42	[▶]
2	71	[– ▶▶▶]

`PulseTexture` #17112

Layer	PatternID	Content
0	41	[▶ – –]
1	42	[▶ ▶]
2	71	[– ▶ ▶]

Yet to be implemented as of this writing are the *rest profile*, the *coverage profile*, and the *non-attacks profile*. The *rest profile* gives the number of simultaneous `REST` events by pulse. The *coverage profile* gives the number of simultaneous non-`REST` events (`ATTACK` or `TIE`) by pulse; it is the compliment of the *rest profile*. The *non-attacks profile* gives the number of simultaneous non-`ATTACK` events (`REST` or `TIE`) by pulse; it is the compliment of the *attacks profile*. Among these the *coverage profile* is most clearly useful.

VII. Relations

My original plan was to enumerate all possible `PulseTexture` instances of `depth` 3 and lengths of 3, 4, and 5, then evaluate all `PulseTexture` pairs to discover whether some close relationship existed between the pair and, if so, to document that relationship. This plan was wrecked by combinatorial explosion. Testing for `depth` 3 produced 120 `PulseTexture` instances of `length` 2 and 1,771 `PulseTexture` instances of length 3. This meant evaluating $(120 + 1{,}771)^2 = 18{,}912 = 3{,}575{,}881$ `PulseTexture` pairs. The test ran through to completion, but it took my laptop 3 days

to run it. As things turned out I was able to speed things up substantially by caching pattern maps into in-memory collections. This allowed me to include textures of `length` 4, of which there were 29,260 instances. The number of required pairwise evaluations thus increased to $(120 + 1{,}771 + 29{,}260)^2 = 311{,}512 = 970{,}384{,}801$; however in-memory caching allowed my laptop to crunch these through in 2 days rather than 3. However, there were bugs. Once those were (mostly) remedied it required over 4 days to build the file.

Relations are defined between `PulsePattern` instances. A `Relation` instance assigns a unique long-integer ID, to the combination of a `RelationCategory` (defined through an `Enum`) and an offset. Thus *IDENTITY*(0) indicates the identity relation while *REVERSE*(0) indicates the retrograde relation. 0 is the only offset permitted for these two categories. Here is a summary of `RelationCategory` items:

- *IDENTITY* — Target same as source. This relation only happens when the compared `PulsePattern` instances have the same ID.
- *REVERSE* — Target retrograde of source. This relation preserves durations, for examples, *REVERSE*(0) for [▶ − −] gives [▶ − −] while *REVERSE*(0) for [▶ − ▶] gives [▶ − ▶].
- *EXTEND* — Target same as source except for pulse inserted in N-th position. If an inserted *REST* would precede a *TIE*, then the *TIE* converts to an *ATTACK*. For example, *EXTEND*(1) for [▶ − ▶] gives [▶ ▶ − ▶], [▶ − − ▶], and [▶ ▶ ▶].
- *TRUNCATE* — Target same as source except for pulse removed from N-th position. For example *TRUNCATE*(1) for [▶ − − ▶] gives [▶ − ▶]. If the *ATTACK* is truncated from the succession *REST*, *ATTACK*, *TIE* then the *TIE* converts to an *ATTACK*. For example, *TRUNCATE*(2) for [▶ ▶ −] gives [▶ ▶].
- *ROTATE* — Target derived from source by right shifting N positions, with $N \neq 0$. For example *ROTATE*(1) for [▶ ▶ ▶] gives [▶ ▶ ▶]. If right-shifting a *REST* places it in front of a *TIE*, then the *TIE* converts to an *ATTACK*; for example *ROTATE*(1) for [− ▶] gives [▶ ▶] rather than the invalid [▶ −].
- *MASK* — Target derived from source by resting in N-th pulse. For example, *MASK*(1) for [▶ ▶ ▶ ▶] gives [▶ ▶ ▶]. If pulse $N + 1$ has a *TIE*, then the *TIE* converts to an *ATTACK*; for example, *MASK*(0) for [▶ − −] gives [▶ −]rather than the invalid [− −].
- *EXCHANGE* — Target derived from source by swapping pulse position N with pulse position $N + 1$. For example, *EXCHANGE*(1) for [▶ ▶ ▶] gives [▶ ▶ ▶]. Exchanges preserve durations; for example *EXCHANGE*(0) for [▶ −] gives [▶ −]. If the exchange would move a *REST* in front of a *TIE*, then the *TIE* converts to an *ATTACK*; for example *EXCHANGE*(0) for [−] gives [▶] rather than the invalid [−].

There is no exclusivity to relations: [▶ ▶ ▶ ▶]bears the *IDENTITY*(0) relation to itself, but it also bears the relations *REVERSE*(0), *ROTATE*(N) (for every N), *EXCHANGE*(N) (for every N) and so forth.

Coding the procedures which discover valid relations between patterns was a challenging exercise combining asymmetric iterations with conditional branching. The special circumstances introduced by ties greatly complicate things. This is an exercise I would recommend for beginning programmers, especially those who wish to pursue composing programs.

To document all pattern-to-pattern relations I implemented a `BTreeMap` whose *key* consisted of three long integers: the source-pattern ID, the relation ID, and the target-pattern ID. Placing the relation ID in the middle allowed map queries of the form: given a a reference instance, which `PulsePattern` instances bear any sort of relation? Also map queries of the form: given a a reference instance, which `PulsePattern` instances bear a specific relation? (The `BTreeMap` value repeated the relation ID.) For the record, the number of pattern-to-pattern relations discovered was 6889.

Two `PulseTexture` instances are defined to bear a `Relation` if all their layered `PulsePattern` instances bear the same relation. Given any two `PulseTexture` instances (a source and a target), the comparison algorithm iterated through all the different ways the target layers could align with the source layers. (With the knowledge base `depth` set to three layers per texture, this amounted to six permutations.) The algorithm then identified all the different relations existing between source layer 0 and its corresponding target layer. For each layer-0 relation, the remaining layers were compared. If both remaining source layers bore the same relation to their corresponding target layers, then the two textures were determined to have that relation.

To document the close relationships discovered by these pairwise comparisons, I created a `BTreeMap` whose *key* combined four long integers: the source-texture ID, the relation ID, the permutation ID, and the target-texture ID. (The `BTreeMap` value repeated the relation ID and the permutation ID.)

Specifying the source-texture ID as 16520 and allowing the remaining key fields to range freely queries all texture-to-texture relations with `PulseTexture` #16520 as the source. This query fetched back 51 instances in all. Here once again is `PulseTexture` #16520:

`PulseTexture` #16520

Layer	PatternID	Content
0	40	[▶ – ▶]
1	54	[▶ ▶ – –]
2	78	[– – ▶]

And here is a random sampling of 6 from the 51 instances fetched:

MASK(2)

`PulseTexture` #9688

Layer	PatternID	Content
0	34	[▶]
1	48	[▶ ▶ ▶]
2	76	[– –]

MASK(1)

`PulseTexture` #7003

Layer	PatternID	Content
0	32	[▶ ▶]
1	46	[▶ ▶ –]
2	65	[– ▶]

ROTATE(2)

`PulseTexture` #21902

Layer	PatternID	Content
0	69	[– ▶ ▶]
1	79	[– – ▶ ▶]
2	46	[▶ ▶ –]

ROTATE(3)

`PulseTexture` #29016

Layer	PatternID	Content
0	78	[– – ▶]
1	61	[▶ – – ▶]
2	69	[– ▶ ▶]

NOT(0)

`PulseTexture` #1958

Layer	PatternID	Content
0	42	[▶]
1	29	[ ]
2	30	[▶]

TRUNCATE(0)

`PulseTexture` #1684

Layer	PatternID	Content
0	19	[▶ – ▶]
1	20	[▶ – –]
2	23	[– ▶]

(Notice that `PulseTexture` #9688 is missing an attack in position 3 of layer 0. The `RelationCategory` code still needs work.)

VIII. Application: Foundational Textures

The rhythmic knowledge base described here takes a neutral attitude toward rhythmic material. With applications this changes. Here the user actively expresses some sort of preference. He or she first uses a `BTreeMap` to *query* for candidates meeting some desired criterion. The resulting list can then be *filtered* by iterating through them and discarding those which fail to meet additional criteria. The pared-down list should then be shuffled randomly to eliminate enumeration biases. If all criteria have been applied, then the first candidate in the shuffled list becomes the selection. However sometimes additional criteria remain which impose additional overhead — like trying out different permutations. In this scenario a second iteration may be necessary. The selected candidate will be the first one in the shuffled list which has a permutation that works.

i. Silence

A texture is *concerted* when for any given pulse position, all layers have the same event type. This very first application will demonstrate how to identify concerted textures which are entirely silent; that is, where the event type is *`REST`* for all layers and all positions.

There is no lookup map which specifically identifies silent textures. The best route available is the attacks-profile lookup map. Querying this map for the attacks profile [0,0] returns 10 candidates (from the 120 textures of length 2). The first 5 of these are:

`PulseTexture` #0

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	0	[ ]

`PulseTexture` #7

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	7	[– –]

`PulseTexture` #5

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	5	[–]

`PulseTexture` #30

Layer	PatternID	Content
0	0	[ ]
1	5	[–]
2	5	[–]

`PulseTexture` #30

Layer	PatternID	Content
0	0	[ ]
1	5	[–]
2	7	[– –]

While it happens that `PulseTexture` #0 is the exactly the texture sought, the end user can't be expected to know that the texture-enumeration algorithm would have produced this first. However, with only 10 candidates it is not unreasonable to iterate through the candidates to filter out those with event type *`TIE`* in any layer or position. Once coverage profiles have been captured within the stored `PulseTexture` instances, it will be a simple matter to exclude textures with coverage profiles other than [0,0].

To identify silent textures of length 3 involves first querying for textures with attacks profile [0,0,0]. This returns 20 candidates (from the 1771 textures of length 3), which can then be filtered for coverage profile [0,0,0].

Likewise, identifying silent textures of length 4 involves first querying for textures with attacks profile [0,0,0,0]. This returns 35 candidates (from the 29260 textures of length 4), which can then be filtered for coverage profile [0,0,0,0].

In summary, the query-and-filter operations just described produce exactly three examples of entirely silent textures, one for each texture length in the knowledge base:

`PulseTexture` #0

Layer	PatternID	Content
0	0	[ ]
1	0	[ ]
2	0	[ ]

`PulseTexture` #120

Layer	PatternID	Content
0	8	[ ]
1	8	[ ]
2	8	[ ]

`PulseTexture` #1891

Layer	PatternID	Content
0	29	[ ]
1	29	[ ]
2	29	[ ]

ii. Onbeats

This second application will demonstrate how to identify concerted textures where the event type is *`ATTACK`* for all layers in position 0 and not *`ATTACK`* elsewhere. Candidates will be identified using the attacks-profile lookup map, for profiles containing the number of simultaneous events per pulse in pulse position 0 and 0 in all other positions. The number of simultaneous events per pulse is determined by the knowledge-base depth, which is 3.

The patterns will be articulated in three ways:

- Staccato onbeats will be identified using a coverage profile that is the same as the attacks profile.
- Sustained onbeats will be identified using a coverage profile with 3 in all positions.
- Detached onbeats will be identified using a coverage profile with 0 in the rightmost position and with 3 in all other positions.

The query phases of these operations produced 5 candidates of `length` 2, 10 candidates of `length` 3, and 20 candidates of `length` 4.

Filtering by the staccato coverage profile produced exactly three examples of staccato-onbeat textures, one for each texture `length` in the knowledge base:

`PulseTexture` #64

Layer	PatternID	Content
0	2	[▶]
1	2	[▶]
2	2	[▶]

`PulseTexture` #1075

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	13	[▶]

`PulseTexture` #17907

Layer	PatternID	Content
0	29	[▶]
1	29	[▶]
2	29	[▶]

Filtering by the sustained coverage profile produced exactly three examples of sustained-onbeat textures, one for each texture length in the knowledge base:

`PulseTexture` #100

Layer	PatternID	Content
0	4	[▶–]
1	4	[▶–]
2	4	[▶–]

`PulseTexture` #1726

Layer	PatternID	Content
0	20	[▶ ––]
1	20	[▶ ––]
2	20	[▶ ––]

`PulseTexture` #29127

Layer	PatternID	Content
0	62	[▶ –––]
1	62	[▶ –––]
2	62	[▶ –––]

Filtering by the detached coverage profile produced exactly three examples of detached-onbeat textures, one for each texture length in the knowledge base:

`PulseTexture` #64

Layer	PatternID	Content
0	2	[▶–]
1	2	[▶–]
2	2	[▶–]

`PulseTexture` #1605

Layer	PatternID	Content
0	18	[▶ –]
1	18	[▶ –]
2	18	[▶ –]

`PulseTexture` #28551

Layer	PatternID	Content
0	60	[▶ ––]
1	60	[▶ ––]
2	60	[▶ ––]

IX. Application: Additive Rhythm

What I'm looking to do first with this knowledge base is additive rhythm. The most basic thing that happens with additive rhythm is elongating musical ideas one pulse at a time.

This next application will generate a `TextureStatement` compounding six simple texture-statements with this structure:

i. XXO XX'O

Texture-statement **X**, being first, cannot contain *TIE* events in pulse position 0. Its activity profile will be [3,1,2,1]. This profile satisfies the no-starting *TIE* constraint and also establishes a $2+2$ beat structure. Texture-statement **O** will be the staccato onbeat texture of `length` 4, identified previously under "Onbeats" (Subsection ii of Section VIII) as `PulseTexture` #17907. Item **X'** will extend texture-statement **X** by one pulse, added to the end. The first four pulses will have the same content as texture-statement **X**. The fifth pulse will have an *ATTACK* in one layer. This establishes a beat structure of $2+3$.

But wait: The knowledge base does not support `PulseTexture` instances of `length` 5 or greater. That means dividing texture-statement **X** into texture-statement **A** with activity profile [3,1] and texture-statement **B** with activity profile [2,1], then extending texture-statement B into texture-statement B′ with activity profile [2,1,1]. The structure now becomes:

ii. XXO XAB'O

Step 1: Texture-statement **X** is selected by querying the `PulseTexture` supply by attacks profile [3,1,2,1]. This query discovered 100 candidates. Here are the first 6:

`PulseTexture` #17996

Layer	PatternID	Content
0	42	[▶]
1	44	[▶ ▶]
2	50	[▶▶▶▶]

`PulseTexture` #18073

Layer	PatternID	Content
0	42	[▶]
1	46	[▶ ▶–]
2	50	[▶▶▶▶]

`PulseTexture` #18034

Layer	PatternID	Content
0	42	[▶]
1	45	[▶ ▶▶]
2	49	[▶▶▶]

`PulseTexture` #18189

Layer	PatternID	Content
0	42	[▶]
1	49	[▶▶▶]
2	58	[▶–▶▶]

`PulseTexture` #18036

Layer	PatternID	Content
0	42	[▶]
1	45	[▶ ▶▶]
2	51	[▶▶▶–]

`PulseTexture` #18222

Layer	PatternID	Content
0	42	[▶]
1	50	[▶▶▶▶]
2	57	[▶–▶]

The query fetches results in their enumeration order, which seems to disfavor *TIE* events. Since the only criterion prescribed is the attacks profile, any bias introduced by the enumeration algorithm should be overcome by choosing one candidate at random. So I did that (and ended up tweaking the random seed until the result had a few ties). The selected candidate was #19999:

`PulseTexture` #19999

Layer	PatternID	Content
0	44	[▶ ▶]
1	53	[▶▶–▶]
2	57	[▶–▶]

Step 2: PulseTexture #19999 embraces six different actual passages depending upon which musical part plays which layer. The selected permutation orders layers from most to least active, using the *beauty contest* described above under "Order of Patterns in Textures" (Section V):

`PulseTexture` #19999

Layer	PatternID	Content
0	53	[▶▶–▶]
1	57	[▶–▶]
2	44	[▶ ▶]

Step 3: How to divide **X** into **A** and **B**, given how the knowledge base documents relations between `PulseTexture` instances? There is no direct way to extract two out of four pulses from a texture. However, texture-statement **A** can be obtained from texture-statement **X** by looking up an intermediate texture-statement **Q** bearing the relation *TRUNCATE*(3) to **X**, then looking up a `PulseTexture` bearing the relation *TRUNCATE*(2) to **Q**. Both of these lookups are supported by a `BTreeMap`, making them efficient.

Querying the knowledge base for textures with the *TRUNCATE*(3) relation to `PulseTexture` #19999 fetches back:

`PulseTexture` #1255

Layer	PatternID	Content
0	17	[▶▶–]
1	19	[▶–▶]
2	14	[▶ ▶]

Querying the knowledge base for textures with the *TRUNCATE*(2) relation to `PulseTexture` #1255 gives the result desired for texture-statement **A**:

`PulseTexture` #71

Layer	PatternID	Content
0	17	[▶▶]
1	19	[▶–]
2	14	[▶]

Step 4: Extracting the final two pulses out of texture-statement **X** can be accomplished by looking up an intermediate texture-statement R bearing the relation *TRUNCATE*(0) to **X**, then looking up a `PulseTexture` bearing the relation *TRUNCATE*(0) to **R**.

Querying the knowledge base for textures with the *TRUNCATE*(0) relation to `PulseTexture` #19999 fetches back:

`PulseTexture` #700

Layer	PatternID	Content
0	19	[▶–▶]
1	23	[–▶]
2	10	[▶]

Querying the knowledge base for textures with the *TRUNCATE*(0) relation to `PulseTexture` #700 gives the result desired for texture-statement **B**:

`PulseTexture` #68

Layer	PatternID	Content
0	2	[▶]
1	2	[▶]
2	6	[–▶]

Step 5: Querying the knowledge base for textures with the *EXTEND*(2) relation to texture-statement **B** brought back 18 textures of `length` 3. Filtering these down to attacks profile [2,1,1] produced 6 candidates:

`PulseTexture` #1086

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	24	[– ▶▶]

`PulseTexture` #1086

Layer	PatternID	Content
0	13	[▶]
1	13	[▶]
2	24	[– ▶▶]

`PulseTexture` #1100

Layer	PatternID	Content
0	13	[▶]
1	14	[▶ ▶]
2	23	[– ▶]

`PulseTexture` #1100

Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	23	[– ▶]

`PulseTexture` #1102

Layer	PatternID	Content
0	13	[▶]
1	14	[▶ ▶]
2	25	[– ▶ –]

`PulseTexture` #1102

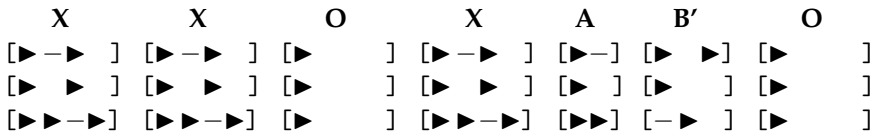
Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	25	[– ▶ –]

Of these `PulseTexture` #1100 was selected at random, then permuted to align with texture-statement **X**:

`PulseTexture` #1100

Layer	PatternID	Content
0	14	[▶ ▶]
1	13	[▶]
2	23	[– ▶]

All the component texture statements have been identified. It just remains to join these simple statements into a compound `TextureStatement` instance. Here is the result:



X. Application: Vertical Masking

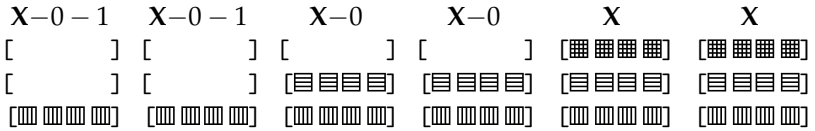
This application gives an imagined nod to the *Schillinger System* [4] or at least something that John Myhill[6] presented as coming from the Schillinger System during a spring 1979 course on composing programs at the State University of New York at Buffalo.

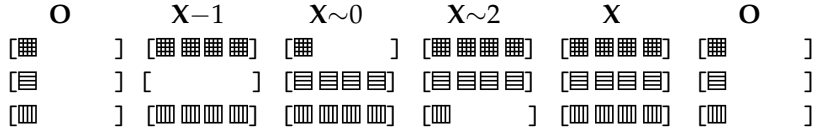
I have mixed feelings about Schillinger. I have no sympathy for his aesthetic, which to me seems to hold that music is somehow 'good' if derivable mathematically and 'poor' otherwise. I am also intimidated both by the sheer size of his tome and by its opaqueness. I acquired the *Schillinger System*'s two massive volumes when still in high school, but never personally got much past the "Theory of Rhythm" (Book I) and the "Theory of Melody" (Book IV). Yet the Schillinger System is no more arbitrary than serialism, and it potentially offers techniques far richer than retrograde, inversion, and transposition. If one can get through those many pages of mathematical ciphers.

The technique of *vertical masking*, employed here, was something John Myhill talked about while presenting Schillinger in the above-mentioned course. The thrust of Myhill's presentation was that one can take a musical passage with several voices and vary it by rotating which instrument plays which part — rotation being Schillinger's go-to permutational device. Vertical masking provides further variation by selectively blanking out some parts while the others continue to play. Since this is exactly what happens in fugal expositions, the processes of part-permutation and vertical masking naturally go together. However I am presently writing this with Schillinger's volumes beside me. Thumbing through these books (which I hadn't cracked in decades), I have not been able to locate the masking scenario Myhill presented.

I had introduced `ROTATE` and `MASK` (the horizontal kind) as `RelationCategory` items with Schillinger in mind. Now I discover that masking is not actually present in Schillinger's technique. I had also been thinking that there was a fair amount of overlap between what I am doing here and what Schillinger did back when. This experience suggests not.

This present application starts with a texture-statement **X**, deconstructs it into layers, then arranges the layers according to the plan presented below:





[6] John Myhill was a close friend of Lejaren Hiller who taught mathematics at SUNY/Buffalo. His field was recursive function theory but he was also keenly interested in composing programs.

O indicates the staccato onbeat texture of `length` 4 identified under "Onbeats" as `PulseTexture` #17907 (Subsection ii of Section VIII). Since the content of **X** has yet to be determined beyond its `length` (4), the above plan uses the symbols "▦", "▤" and "▥" to indicate different layers. Variations upon **X** will be derived by blanking out one or two of the three layers. If "blank" means rest in all pulse positions, then are 3 ways of blanking 1 out of 3 layers and also 3 ways of blanking 2 out of 3 layers. However metric considerations suggest that that for two consecutive statements (a leader and a trailer), then a part which is active in the leader but blank in the trailer should be allowed 'resolution' to the trailer downbeat. So the **X**∼0 and **X**∼1 items in the above plan blank out all bit the first pulse position.

The plan indicates layer-specific blanking options by appending the blanked layer ID, prefixed either by a hyphen ("−") or a tilde ("∼"). The hyphen indicates full silence (all pulses resting), while the tilde indicates an isolated down beat of (just one attack, then rests). For example **X**−0 − 1 indicates the `PulseTexture` which carries over from texture-statement **X** in layer 2 but which is fully silent in layers 0 and 1.

Readers will notice that in spite of my backstory, the plan takes no steps to rotate layers. I originally intended to include rotations but decided it would unnecessarily complicate the plan. Layer permutation is basic functionality in `TextureStatement` instances. It is no reach at all to rotate layers if one wishes to do so.

For this present application, texture-statement **X** will be metric, with full coverage (no part rests during any pulse). It should be equally active in all parts; that is, the `imbalanceCount` (Subsection iv of Section IV) should be 0 (this did not prove attainable).

Step 1: Selecting texture-statement **X** began by querying the `PulseTexture` supply by attacks profile [3,1,2,1]. As reported earlier, this query discovered 100 candidates. Filtering out candidates with coverage profiles other than [3,3,3,3] whittled this number down to 5. None of these candidates had `imbalanceCount` scores of 0, which is what I was hoping for, but three candidates had `imbalanceCount` scores of 1:

`PulseTexture` #24844

Layer	PatternID	Content
0	51	[▶▶▶−]
1	59	[▶−▶−]
2	61	[▶−−▶]

`PulseTexture` #25866

Layer	PatternID	Content
0	53	[▶▶−▶]
1	59	[▶−▶−]
2	59	[▶−▶−]

`PulseTexture` #26306

Layer	PatternID	Content
0	54	[▶▶−−]
1	58	[▶−▶▶]
2	59	[▶−▶−]

The application randomly selected `PulseTexture` #26306, which is fortunate because this texture conforms least slavishly to the meter. (Demonstrating once again that while randomness is necessary for *unbiased* selection, positive criteria should override.) The selected permutation of `PulseTexture` #26306 orders its layers from most to least active, using the *beauty contest* described above under "Order of Patterns in Textures" (Section V):

`PulseTexture` #26306

Layer	PatternID	Content
1	58	[▶ − ▶▶]
2	59	[▶ − ▶ −]
0	54	[▶ ▶ − −]

Statement **X**

The earlier "Silence" application (Subsection i of Section VIII) identified `PulseTexture` #1891 as the silent texture of length 4. The supply of `PulseTexture` instances has a lookup map by pattern IDs, so querying this map with patterns #29 (from all layers of `PulseTexture` #1891), #29 again, and #54 (bottom layer of **X**) is very efficient. This query fetched back `PulsePattern` #1916:

`PulseTexture` #1916

Layer	PatternID	Content
1	29	[ ]
2	29	[ ]
0	54	[▶ ▶ − −]

Statement **X**$-0-1$

The remaining statements are identified by similar lookup queries. The earlier "Onbeats" application (Subsection ii of Section VIII) identified `PulseTexture` #17907 as the staccato onbeat texture of `length` 4. All three layers of `PulseTexture` #17907 employ `PulsePattern` #42, therefore #42 is the 'blanked' pattern ID used in for the **X**~ 0 and **X**~ 2 queries:

`PulseTexture` #2971

Layer	PatternID	Content
0	29	[ ]
1	59	[▶ − ▶ −]
2	54	[▶ ▶ − −]

Statement **X**-0

`PulseTexture` #18350

Layer	PatternID	Content
0	42	[▶]
1	59	[▶ − ▶ −]
2	54	[▶ ▶ − −]

Statement **X**~ 1

`PulseTexture` #2970

Layer	PatternID	Content
0	58	[▶ − ▶▶]
1	29	[ ]
2	54	[▶ ▶ − −]

Statement **X**-1

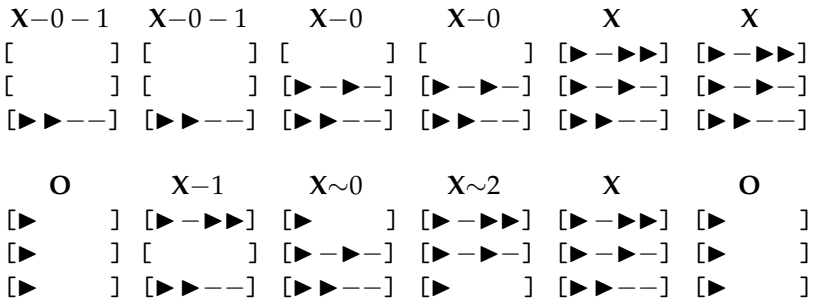
`PulseTexture` #18460

Layer	PatternID	Content
0	58	[▶ − ▶▶]
1	59	[▶ − ▶ −]
2	42	[▶]

Statement **X**~ 2

Understand that the lookup map lists pattern ID's in all possible permutations. Thus looking up the pattern-ID sequence [29,59,54] will fetch back `PulsePattern` #2971 even though this instance actually lists its component patterns in ascending order: [29,54,59]. The code surrounding the lookup request also determines what `Permutation` is necessary to present the patterns in their requested order.

Compounding these seven simple statements together according to the plan graphed above completes the result:



XI. Application: Counter Rhythm

My coined term *counter rhythm* abstracts pitch away from *counter melody*, which according to *Wikipedia* is "a sequence of notes ... written to be played simultaneously with a more prominent lead melody". This next application seeks to write *two* counter rhythms, to be played simultaneously with a more prominent lead rhythm. The lead rhythm is given as an input.

My own take is that a good counter rhythm compliments the lead with respect to the meter. That means that when the lead part syncopates over a strong beat, the counter part fills in the beat. And when the lead part attacks a weak beat, the counter part either ties over or rests. (Remember the premise, stated earlier under "Profiles" (Section VI), that "musical meter is established through the convergence of polyphonic attacks on strong beats and divergence of attacks on weak beats.")

Another desirable feature of counter rhythms is that they actively contribute. This additional proviso was added late after initial attempts produced solutions where one counter rhythm simply rested.

Here is the rhythm for the lead part:

[▶ ▶▶] [− ▶▶ −] [▶]

The meter is described using the attacks profile [2,1,2,1], which repeats. Interpreted with respect to this 2+2 beat structure, the lead rhythm features a syncopation over beat 2. Placing 2, rather than 3, in position 0 of the attacks profile ensures that one counter part will play a pickup rhythm during beat 1. It also ensures that both counter parts will attack the second onbeat (since the lead syncopates there).

The solution involves the following steps:

1. Using the knowledge base to find `PulseTexture` instances which conform to attacks profile [2,1,2,1].

2. Filtering out those `PulseTexture` instances which do not contain the lead rhythm or which have `imbalanceCount` properties greater than 2 (no successful candidates actually had fewer). The results of this step are then randomly shuffled to eliminate enumeration bias.

3. Iterating through the shuffled instances. For each instance, a `Permutation` is sought which brings the lead pattern to part 0 and which also excludes unanticipated ties. If such a permutation is discovered, the `Pulse` and its associated `Permutation` are selected. Otherwise iteration proceeds.

Step 1: Querying for textures with attacks profile [2,1,2,1] returns 476 `PulseTexture` instances.
Step 2: (Filtering)

- Looking up [▶ ▶ ▶] returned `PulsePattern` #45. 58 of the textures returned in Step 1 included `PulsePattern` #45 as one layer while also possessing an `imbalanceCount` of 2.
- Looking up [– ▶ ▶ –] returned `PulsePattern` #72. 20 of the textures returned in Step 1 included `PulsePattern` 4 #72 as one layer while also possessing an `imbalanceCount` of 2.

Step 3: (Iteration and 2nd Filtering)
Here are the first 6 of the 58 shuffled `PulseTexture` instances obtained for [▶ ▶▶] (`PulsePattern` #45).

`PulseTexture` #20853

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	55	[▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #11962

Layer	PatternID	Content
0	36	[▶▶]
1	45	[▶ ▶▶]
2	60	[▶ – –]

`PulseTexture` #21028

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	62	[▶ – – –]
2	72	[– ▶▶ –]

`PulseTexture` #20962

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	59	[▶ – ▶ –]
2	75	[– ▶ – –]

`PulseTexture` #20601

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	47	[▶▶]
2	80	[– – ▶ –]

`PulseTexture` #9507

Layer	PatternID	Content
0	34	[▶]
1	44	[▶ ▶]
2	45	[▶ ▶▶]

The first 3 of these options begin with ties; therefore the specific texture chosen was #11962. This texture lists pattern #45 as layer 1:

`PulseTexture` #11962

Layer	PatternID	Content
1	45	[▶ ▶▶]
0	36	[▶▶]
2	60	[▶ – –]

Here are the first 6 of the 20 shuffled `PulseTexture` instances obtained for [– ▶▶ –] (`PulsePattern` #72).

`PulseTexture` #18956

Layer	PatternID	Content
0	43	[▶ ▶]
1	46	[▶ ▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #27610

Layer	PatternID	Content
0	57	[▶ – ▶]
1	61	[▶ – – ▶]
2	72	[– ▶▶ –]

`PulseTexture` #19308

Layer	PatternID	Content
0	43	[▶ ▶]
1	57	[▶ – ▶]
2	72	[– ▶▶ –]

`PulseTexture` #26754

Layer	PatternID	Content
0	55	[▶ –]
1	58	[▶ – ▶▶]
2	72	[– ▶▶ –]

`PulseTexture` #27185

Layer	PatternID	Content
0	56	[▶ – ▶]
1	69	[▶ – ▶ –]
2	72	[– ▶▶ –]

`PulseTexture` #20853

Layer	PatternID	Content
0	45	[▶ ▶▶]
1	55	[▶ –]
2	72	[– ▶▶ –]

The first of these options is `PulseTexture` #18956. Layer 2 is pattern #72, which is the lead rhythm. This pattern [– ▶▶ –] begins with a *`TIE`*, but that's okay because it follows on after pattern #45 [▶ ▶▶] which does not end with a *`REST`*. So #18956 is the selection:

`PulseTexture` #18956

Layer	PatternID	Content
1	72	[– ▶▶ –]
0	43	[▶ ▶]
2	46	[▶ ▶ –]

And here is the assembled TextureStatement instance:

```
[▶  ▶▶] [– ▶▶ –] [▶    ]
[  ▶▶  ] [▶   ▶] [▶    ]
[▶ – –  ] [▶  ▶ –] [▶    ]
```

XII. Application: Cross Rhythm

The term *cross rhythm* here refers to simultaneous musical parts playing patterns which share a common pulse but which do not share the same length. This is distinguished from *polyrhythm*, where pulses happen at different speeds, and also from *hemiola*, which is a full metric modulation between, say, 3/4 time and 6/8 time. The knowledge base described here copes with hemiola very easily. It does not cope with polyrhythm at all. The present exercise will demonstrate that cross-rhythm is doable. However whether it is worth the trouble depends upon whether one finds it needful to express a cross-rhythm in the `TextureStatement` format.

Cross rhythm and polyrhythm (but not hemiola) are both examples of what Steve Reich calls "phase music". Another example is the selection principle I call "statistical feedback"[7] when applied to nonuniform weights. The compositional attraction here is that you have a period of time over which individual parts proceed inexorably but which the tension between parts destabilizes until everything comes together at the moment of convergence. Like a cadence only different.

This exercise will cross the pattern [▶▶▶▶] (5 pulses) in one musical part with the pattern [▶ – ▶▶] (4 pulses) in a second musical part. Understand that these components need not

[7] https://charlesames.net/feedback/index.html

be literal — for example, one could dynamically swap out [▶▶▶▶] for [▶▶ – ▶] or even [– ▶▶▶▶].

Often just one part 'crosses' the rest of the ensemble; that is, the remaining parts hold to an established beat. The present exercise will not play favorites in that way. Rather the third part will present the rhythm derived by Joseph *Schillinger* in "Interferences of Periodicities", Chapter 2 of Book 1, the Schillinger System's "Theory of Rhythm". In this case the 'periodicities' are 5 and 4 and the resultant sequence of durations (pulse counts) is {4, 1, 3, 2, 2, 3, 1, 4}. The present exercise will articulate these durations in a detached manner, meaning that pulse positions between *ATTACK* events will be *TIE* events up to the position just before the next *ATTACK*; this will be a *REST* event.

What will be produced here is a succession of texture-statement instances, and the first thing to decide is the sequence of statement-instance lengths. One alternative would be employ Schillinger's interference durations; however, the knowledge base does not support `PulseTexture` instances of `length` 1. A reasonable alternative is to employ the shorter pattern `length` (4), since `PulseTexture` instances of `length` 5 are also not supported.

Next follows a collation algorithm which works out what sequence of pulse events each part will play during a statement, which looks up the corresponding `PulsePattern` in the knowledge base, and which in turn uses all three patterns to look up a `PulseTexture` and an associated `Permutation`. I explained how `PulseTexture` lookups work under the "Vertical Masking" application (Section X). The point here is that the knowledge base doesn't really help out with these collation tasks. Cross rhythm is not what the knowledge base is about. The present exercise only makes sense if performed within a larger context that makes active use of the `TextureStatement` representation.

Here is the assembled result:

```
[  ▶▶▶] [▶  ▶▶] [▶▶  ▶] [▶▶▶  ] [▶▶▶▶] [▶     ]
 5          5         5         5
[▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶  ▶▶] [▶     ]
 4         4         4         4         4
[▶––  ] [▶▶–  ] [▶  ▶  ] [▶–  ▶] [▶––  ] [▶     ]
 4         1 3       2   2       3   1    4
```

XIII. Reflections

This article describes a personal effort to carry through ideas I've had percolating in response to earlier projects of mine. Whether anybody (other than Schillinger) has done this before, all or in part, I wouldn't know. I have long been out of academics. Having limited cognitively productive hours, I feel no obligation to divert them into scholarship.

I am grateful to Hugo Carvalho, to the MusMat Research Group, and to the *Brazilian Journal of Music and Mathematics* for inviting me to contribute. The software project described here did not exist prior to that invitation, so it can legitimately said to have been created under MusMat auspices.

If I suddenly had a massively parallel supercomputer at my disposal (plus a full-time professional programmer to adapt my existing inline code to multi-threading), I would increase the number of overlaid patterns in a texture (its `depth`) from 3 to 4. That would raise the number of textures by a power of 4 and the number of texture-pair evaluations by a power of 8 (?). I believe

the examples provided by this article demonstrate how the length `limit` can be worked around using compound texture statements. There is no corresponding workaround for the `depth` limit.

Limited changes of scale are imaginable. *Parts* can easily scale vertically to homophonic *choirs*. Creative interpretation of the `PulseEventType` set (`REST`, `ATTACK`, `TIE`) can be used to scale *pulses* horizontally into longer durations. Here context may exert influence. Thus the first duration in the succession [`REST`, `ATTACK`] might be filled with some sort of pickup rhythm.

Something "Concurrence" had which presently does not exist here are primitive pattern alterations like these:

- Promoting a rest/demoting a tie: [►] ⟺ [►–]
- Promoting a tie/demoting an attack: [►–] ⟺ [►►]
- Anticipating/delaying an attack: [►►] ⟺ [►–►]

Such alterations don't readily generalize to textures. Still, they ought to be explored.

References

[1] Ames, Charles (1988). Concurrence. *Journal of New Music Research*, v. 17, n. 1, pp. 3–24.

[2] Ames, Charles (1992). Quantifying Musical Merit. *Journal of New Music Research*, v. 21, n. 1, pp. 53–94.

[3] Kohn, Karl (1981). The Renotation of Polyphonic Music. *The Musical Quarterly*, v. 67, n. 1, pp. 29–49.

[4] Schillinger, Joseph (1941, 1942, 1946). *The Schillinger System of Musical Composition*. New York: Carl Fischer.

FEDERAL UNIVERSITY OF RIO DE JANEIRO

Denise Pires de Carvalho
Rector

Carlos Frederico Leão Rocha
Vice Rector

Denise Maria Guimarães Freire
Pro-Rector of Graduate and Research

CENTER OF LITERATURE AND ARTS

Cristina Grafanassi Tranjan
Dean

SCHOOL OF MUSIC

Ronal Silveira
Director

Marcelo Jardim
Vice-director

David Alves
Associate Director of Undergraduate Education

Marcelo Jardim
Director of the Cultural Artistic Sector

Maria José Bernardes Di Cavalcanti
Director of Public Outreach

João Vidal
Coordinator of the Graduate Program in Music

Patricia Michelini Aguilar
Coordinator of the Graduate Program Professional in Music

MusMat Research Group

COORDINATORS

Carlos Almada (leader)
Carlos Mathias
Cecília Saraiva
Daniel Moreira
Hugo Carvalho
Liduino Pitombeira

CURRENT MEMBERS

Ana Miccolis (Doctoral student)
Ariane Petri (Doctoral student)
Claudia Usai (Master student)
Filipe de Matos Rocha (Doctoral student)
Gabriel Barbosa (Undergraduate Research Project)
Igor Chagas (Master student)
Natanael Luciano de Matos (Master student)
Pedro Proença (Doctoral student)
Pedro Zizels Ramos (Doctoral student)
Rui Aldé Lopes (Undergraduate Research Project)
Sebastião Rodrigo (Undergraduate Research Project)
Tatiana Thays Davalos Alves (Undergraduate Research Project)
Vinicius Braga (Undergraduate Research Project)
Vinícius Rodrigues (Undergraduate Research Project)

ALUMNI

Adriel Viturino (Undergraduate Research Project)
Alexandre Avellar (Undergraduate Research Project)
Alexandre Ferreira (Doctoral student)
André Codeço (Doctoral Student)
André Pinto (Undergraduate Research Project)
Desirée Mayr (Doctoral Student)
Eduardo Cabral (Undergraduate Research Project)
Érico Bonfim (Doctoral student)
Fabio Monteiro (Master student)
Gabriel Mesquita (Master student)

Helder Oliveira (Doctoral student)
João Penchel (Undergraduate Research Project)
Jorge Santos (Master student)
Leandro Chrispim (Master student)
Marcel Castro Lima (Master student)
Marco Antônio Ramos Feitosa (Post-doc)
Mariane Batista dos Santos (Undergraduate)
Max Kühn (Master student)
Nathalie Deziderio (Undergraduate)
Rafael Fortes (Master student)
Rafael Soares Bezerra (Undergraduate Research Project)
Roberto Macedo (Doctoral student)
Rodrigo Furman (Master student)
Rodrigo Pascale (Undergraduate Research Project)

COLLABORATORS

Alexandre Schubert (Prof. UFRJ)
Fábio Adour (Prof. UFRJ)
Francisco Erivelton Aragão (Prof. UFC)
Pauxy Gentil-Nunes (Prof. UFRJ)
Nei Rocha (Prof. UFRJ)
Petrucio Viana (Prof. UFF)
Stefanella Boatto (Prof. UFRJ)
Raphael Sousa Santos

FRONT COVER ART

Watercolor painting: Hugo Carvalho
Editing: Daniel Moreira