# 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].

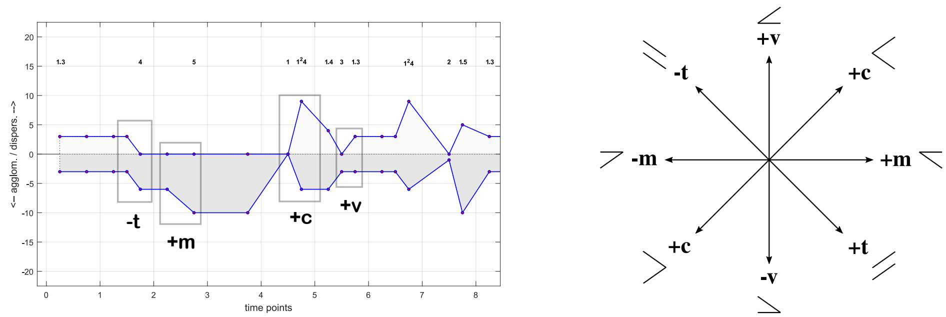
Partitions are representations of integers by the sum of other integers. Since each integer has a finite set of partitions, it is possible to establish an exhaustive taxonomy and map their relationships. One can, too, establish a biunivocal correspondence between partitions and textural configurations. 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.

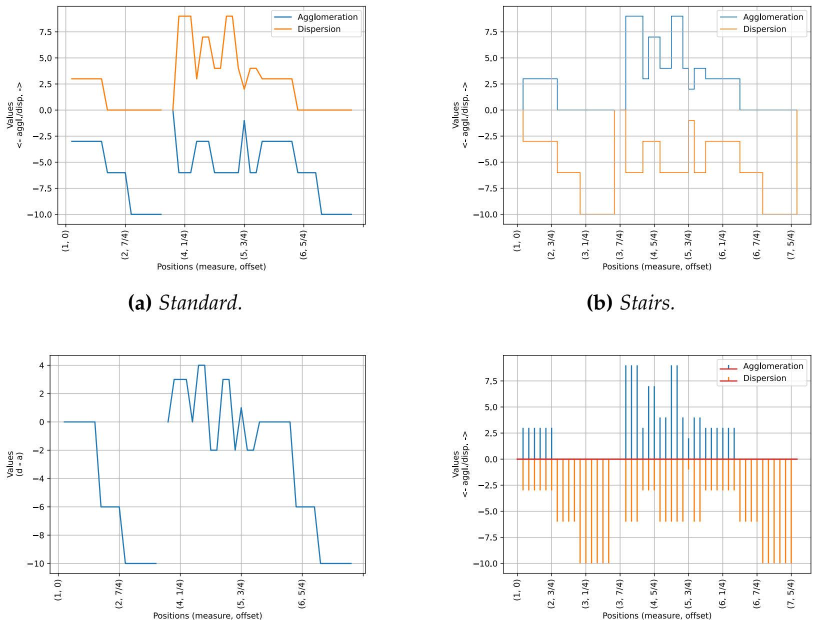
Regarding partitions notation, George Andrews [2] and the mathematicians that work with the Theory of Integer Partitions use to abbreviate them with indexes that express the multiplicity of the parts. When there are successive unique parts, they are separated by dots. For instance, the abbreviated notation 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 partitiogram [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.

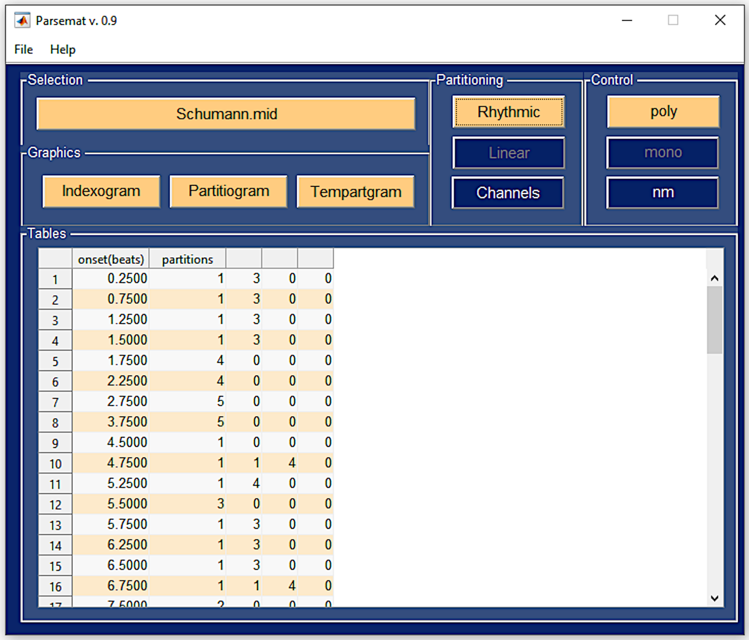
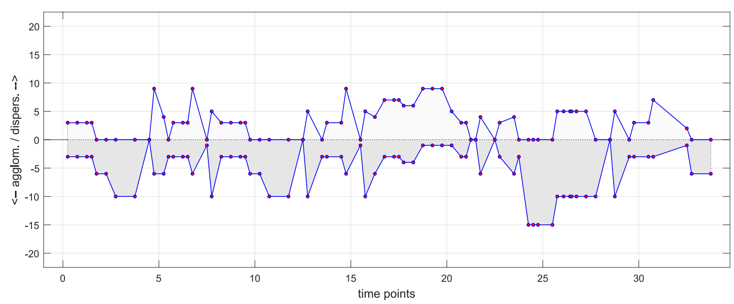
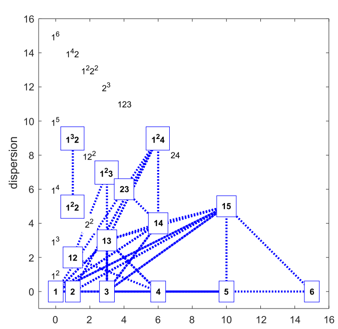


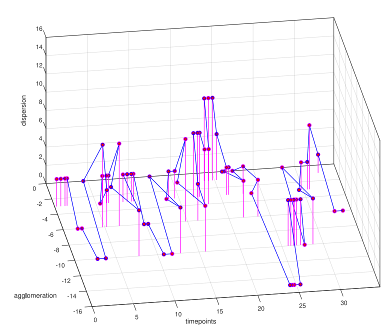

**Figure 3:** *Parsemat's interface.*




**(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).

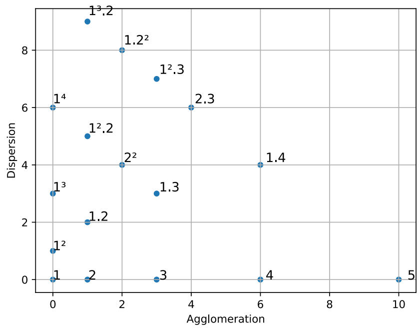
In Mozart's case (Figure 7), the quartet uses a repertoire of partitions very similar to Monteverdi's one—that is, the 11 partitions of the lexical-set of $dn = 4$, and some accessory partitions—in Monteverdi's case, coming from the fifth voice, and in the case of Mozart, the strings' double stops.

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.

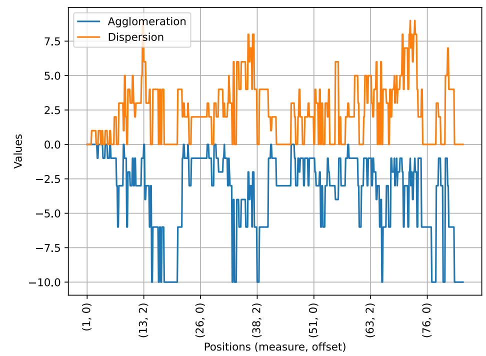
Mozart's partitions for $dn = 6$ and $dn = 8$ occur in the piece's final section and are in a purely cadential context. The arrangement in pairs of dispersion and agglomeration peaks also occurs in the Mozart excerpt. For example, in mm. 100–107.2 (Figure 7c), the alternation occurs between antecedent 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,

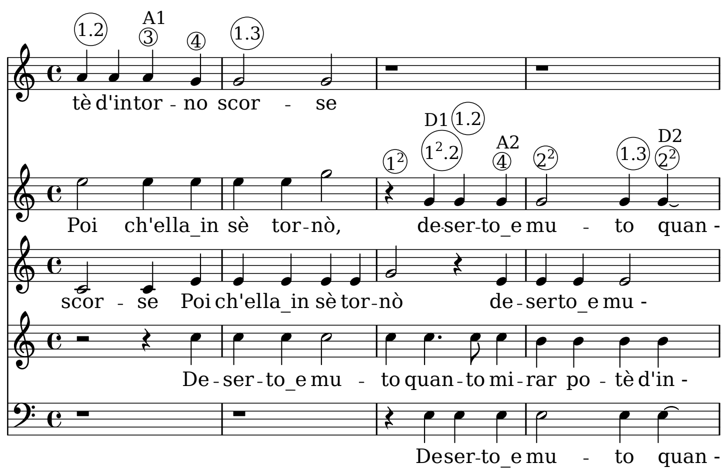
[8] We manually edited Monteverdi's digital score to add note tie endings for RPC's processing. See more information in Section VIII.



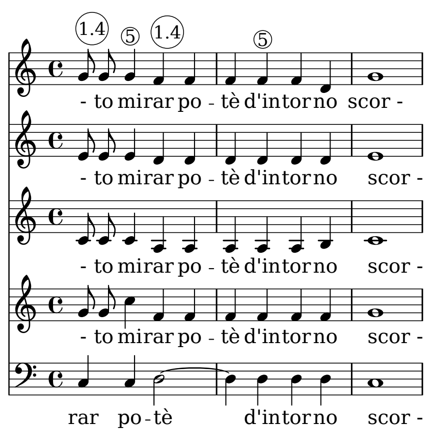

**(a)** *Partitiogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*




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

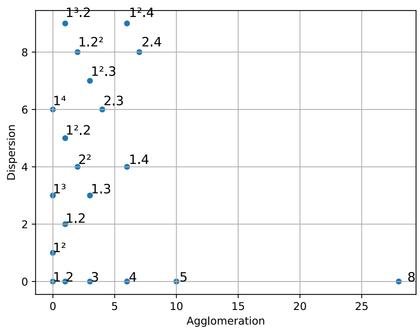



**(c)** *Mm. 7–10. Generated by Music21 [6] (see Section VIII) and Lilypond [21].*

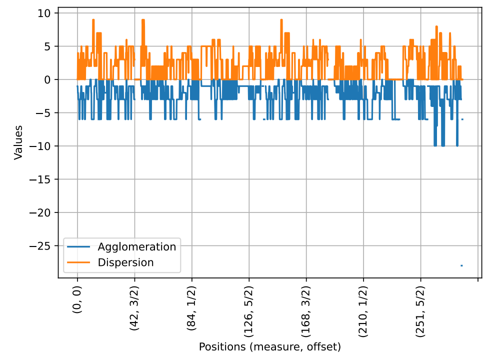



**(d)** *Mm. 17–19. Generated by Music21 [6] (see Section VIII) and Lilypond [21].*

**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()
```

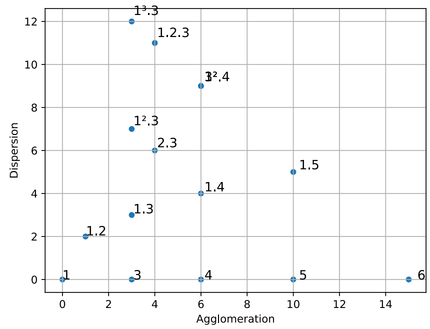
Another RP Scripts' highlight is the possibility of processing large corpora. Since they are standalone scripts, a concatenation in a shell script is possible. For instance, the single line below calls RPC and RPP to create CSV, indexogram, and partitiogram files from all the MusicXML files in a directory:

```
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
```

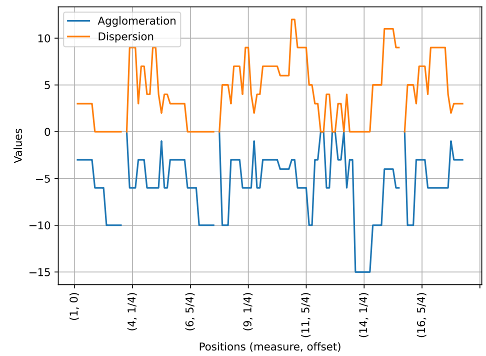
Parsemat's and RPC's outputs differ in two aspects: voice and rest handling. The voice processing is different due to the particularities of the MIDI and MusicXML data parsing. Given two equal MIDI notes coded in two different voices, if they are in the same channel, Parsemat processes them as a single 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.

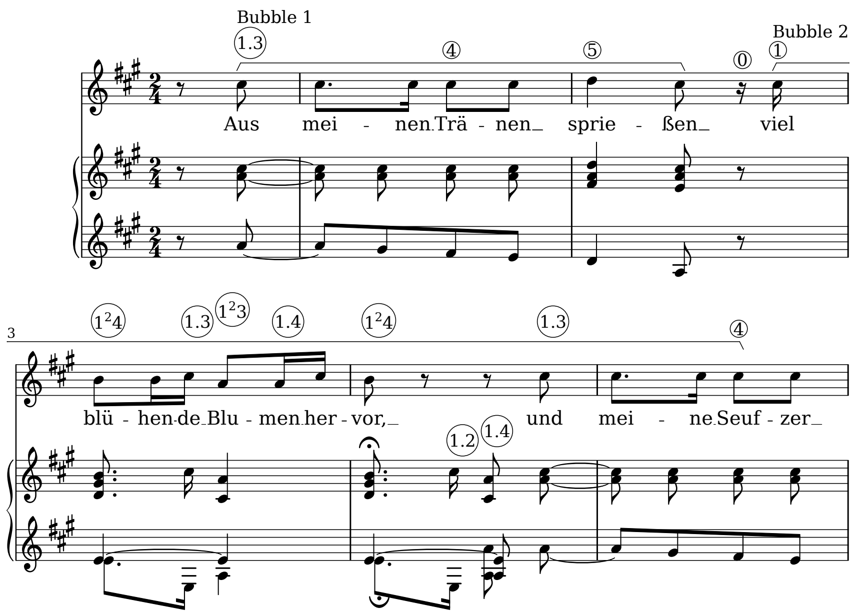
[10] See a complete software list with MusicXML export support at https://www.musicxml.com/software/.




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



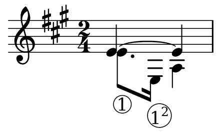

**(b)** *Indexogram. Generated by RPC and RPP Scripts (see Appendixes A and B).*



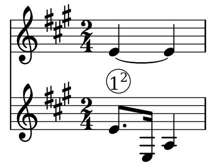

**(c)** *Mm. 1–5. Generated by Music21 [6] (see Section VIII) and Lilypond [21], edited manually to add the partitions and bubbles annotations.*


**(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?).*

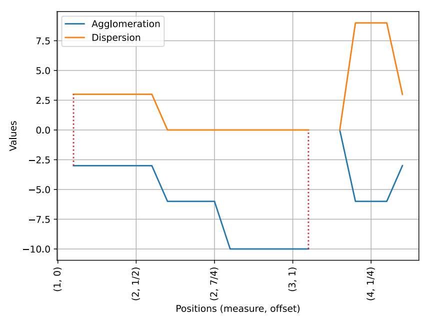



**(a)** *Original.*

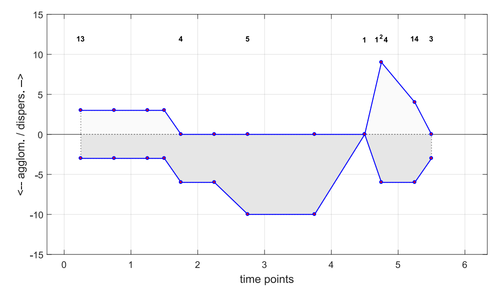



**(b)** *Voices splitting into parts.*

**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.*

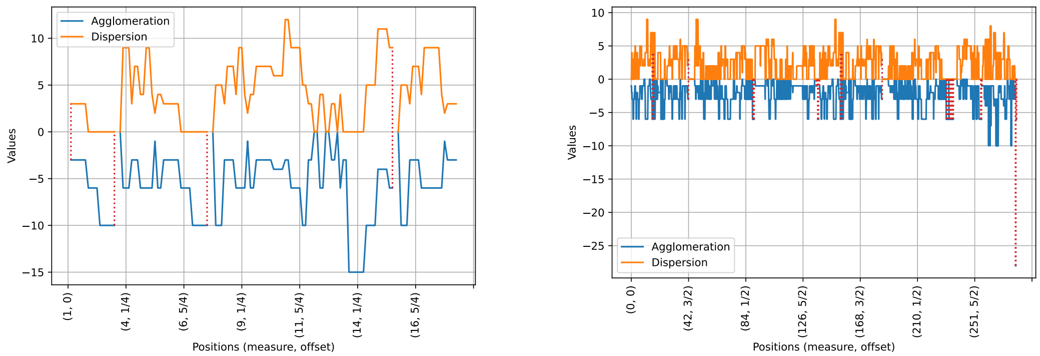
is only $(1^2)$. This algorithm does not merge different voices in this situation. The present authors consider the inclusion of these possibilities as interface options in future releases of Parsemat and RPC.

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

```
import argparse
import copy
import csv
import fractions
import math
import music21
import numpy

def get_number_combinations_pairs(n):
    return n * (n - 1) / 2

def make_fraction(value):
    if isinstance(value, fractions.Fraction):
        return fractions.Fraction(int(value.numerator), int(value.
            denominator))
    else:
        a, b = value.as_integer_ratio()
        return fractions.Fraction(int(a), int(b))

def get_common_fractions_denominator(fractions_lst):
    denominators = [fr.denominator for fr in fractions_lst]
    return numpy.lcm.reduce(denominators)
```

```python

def get_common_denominator_from_list(seq):
    diffs = [b - a for a, b in zip(seq, seq[1:])]
    values = map(make_fraction, sorted(list(set(diffs))))
    return fractions.Fraction(1, get_common_fractions_denominator(
        values))

def find_nearest_smaller(value, seq):
    if value < seq[0]:
        return -1

    if value > seq[-1]:
        return seq[-1]

    size = len(seq)
    if size == 1 and value >= seq[0]:
        return seq[0]

    middle_pointer = math.floor(size/2)
    left = seq[:middle_pointer]
    right = seq[middle_pointer:]

    if value < right[0]:
        return find_nearest_smaller(value, left)
    else:
        return find_nearest_smaller(value, right)

def auxiliary_find_interval(value, dic, i=0):
    size = len(dic.keys())

    if i > size - 1:
        raise IndexError('Given index is out of dic')

    keys = list(dic.keys())
    while i < size - 1 and value >= dic[keys[i + 1]]:
        i += 1

    return keys[i], i

def aux_make_events_from_part(m21_part):
    '''Return a dictionary with location and Musical Events
    from a given Music21 part object.
    '''

    measures = m21_part.getElementsByClass(music21.stream.Measure)

    events = {}

```

```
    for m21_measure in measures:
        notes_and_rests = m21_measure.notesAndRests
        for m21_obj in notes_and_rests:
            m_event = MusicalEvent()
            m_event.set_data_from_m21_obj(m21_obj, m21_measure.number,
                m21_measure.offset)
            events.update({
                m_event.global_offset: m_event
            })

    return events


def aux_join_music_events(events):

    # Add null event at the end
    last_location = list(events.keys())[-1]
    last_event = events[last_location]
    last_location += last_event.duration + 1
    current_event = MusicalEvent()
    current_event.is_null = True
    events.update({
        last_location: MusicalEvent()
    })

    # Start with null
    last_event = None
    last_location = None
    joined_events = {}

    for location, current_event in events.items():
        if current_event.is_null: # any - null
            joined_events.update({last_location: last_event})
        else:
            if not last_event: # null - any
                last_event = current_event
                last_location = location
            else:
                if current_event.is_rest():
                    if last_event.is_rest(): # rest - rest
                        last_event.duration += current_event.duration
                    else: # note - rest
                        joined_events.update({last_location: last_event
                            })
                        last_event = current_event
                        last_location = location
                else:
                    if last_event.is_rest(): # rest - note
```

```
                        joined_events.update({last_location: last_event})
                        last_event = current_event
                        last_location = location
                    else: # note - note
                        if current_event.tie:
                            if current_event.tie == 'start': # note - note.start
                                joined_events.update({last_location: last_event})
                                last_event = current_event
                                last_location = location
                            else: # note - note.continue or note.stop
                                last_event.duration += current_event.duration
                                pass
                        else:
                                joined_events.update({last_location: last_event})
                                last_event = current_event
                                last_location = location

    return joined_events

def make_music_events_from_part(m21_part):
    events = aux_make_events_from_part(m21_part)
    return aux_join_music_events(events)

def pretty_partition_from_list(seq):
    if not seq:
        return '0'
    dic = {}
    for el in seq:
        if el not in dic.keys():
            dic[el] = 0
        dic[el] += 1
    partition = '.'.join([str(k) if v < 2 else '{}^{}'.format(k, v)
        for k, v in sorted(dic.items())
    ])

    return partition

class CustomException(Exception):
    pass

class MusicalEvent(object):
    def __init__(self):
        self.offset = 0
```

```
        self.global_offset = 0
        self.number_of_pitches = 0
        self.duration = 0
        self.tie = None
        self.m21_class = None
        self.is_null = False

    def __str__(self) -> str:
        return ' '.join(list(map(str, [self.number_of_pitches, self.
            duration, self.tie])))

    def __repr__(self):
        return '<E {}>'.format(self.__str__())

    def is_rest(self):
        return self.m21_class == music21.note.Rest

    def set_data_from_m21_obj(self, m21_obj, measure_number,
            measure_offset):
        self.measure_number = measure_number
        self.offset = make_fraction(m21_obj.offset)
        self.global_offset = self.offset + make_fraction(measure_offset
            )
        self.duration = make_fraction(m21_obj.duration.quarterLength)
        self.m21_class = m21_obj.__class__

        if self.is_rest():
            self.number_of_pitches = 0
        else:
            if m21_obj.isNote:
                self.number_of_pitches = 1
            else:
                self.number_of_pitches = len(m21_obj.pitches)
            if m21_obj.tie:
                if m21_obj.tie.type in ['start', 'continue', 'stop']:
                    self.tie = m21_obj.tie.type

class SingleEvent(object):
    def __init__(self):
        self.number_of_pitches = 0
        self.duration = 0
        self.measure_number = 0
        self.offset = 0
        self.sounding = False
        self.partition_info = []

class Parsema(object):
    def __init__(self):
```

```
        self.measure_number = None
        self.offset = None
        self.global_offset = None
        self.duration = 0
        self.single_events = []
        self.partition_info = []
        self.partition_pretty = ''

    def __repr__(self):
        return '<P: {} ({}, {}), dur {}>'.format(self.partition_pretty, self.measure_number, self.offset, self.duration)

    def add_single_events(self, single_events):
        self.single_events = single_events
        durations = [event.duration for event in single_events if event]
        if durations:
            self.duration = min(durations)

        self.set_partition_info()
        self.partition_pretty = pretty_partition_from_list(self.partition_info)

    def set_partition_info(self):
        partitions = {}
        number_of_pitches_set = set([
            s_event.number_of_pitches
            for s_event in self.single_events
        ])
        if list(number_of_pitches_set) == [0]:
            return [0]
        for s_event in self.single_events:
            key = (s_event.sounding, s_event.duration)
            if key not in partitions.keys() and s_event.number_of_pitches > 0:
                partitions[key] = 0
            if s_event.number_of_pitches > 0:
                partitions[key] += s_event.number_of_pitches
        self.partition_info = sorted(partitions.values())

    def get_density_number(self):
        return int(sum(self.partition_info))

    def count_binary_relations(self):
        density_number = self.get_density_number()
        return get_number_combinations_pairs(density_number)

    def get_agglomeration_index(self):
```

```
        if self.partition_info == []:
            return None
        return float(sum([get_number_combinations_pairs(n) for n in self.partition_info]))

    def get_dispersion_index(self):
        if self.partition_info == []:
            return None
        return float(self.count_binary_relations() - self.get_agglomeration_index())

class PartSoundingMap(object):
    def __init__(self):
        self.single_events = None
        self.attack_global_offsets = []

    def __str__(self):
        return len(self.single_events.keys())

    def __repr__(self):
        return '<PSM: {} events>'.format(self.__str__())

    def set_from_m21_part(self, m21_part):
        music_events = make_music_events_from_part(m21_part)
        self.single_events = {}
        for global_offset, m_event in music_events.items():
            # interval: closed start and open end.
            closed_beginning = global_offset
            open_ending = closed_beginning + m_event.duration

            single_event = SingleEvent()
            single_event.number_of_pitches = m_event.number_of_pitches
            single_event.duration = m_event.duration
            single_event.measure_number = m_event.measure_number
            single_event.offset = m_event.offset

            self.single_events.update({
                (closed_beginning, open_ending): single_event
            })
            self.attack_global_offsets.append(closed_beginning)

    def get_single_event_by_location(self, global_offset):
        beginning = find_nearest_smaller(global_offset, self.attack_global_offsets)

        if beginning == -1: # No event to return
            return

```

```
        ind = self.attack_global_offsets.index(beginning)
        _, ending = list(self.single_events.keys())[ind]
        s_event = None
        if global_offset >= beginning and global_offset < ending:
            s_event = copy.deepcopy(self.single_events[(beginning,
                ending)])
            duration_diff = global_offset - beginning
            duration = s_event.duration
            duration = duration - duration_diff
            sounding = duration_diff > 0
            s_event.duration = duration
            if s_event.number_of_pitches > 0:
                s_event.sounding = sounding
            else:
                s_event.sounding = False
        return s_event

class ScoreSoundingMap(object):
    def __init__(self):
        self.sounding_maps = []
        self.attacks = []
        self.measure_offsets = {}

    def __repr__(self):
        return '<SSM: {} maps, {} attacks>'.format(len(self.
            sounding_maps), len(self.attacks))

    def add_part_sounding_map(self, m21_part):
        psm = PartSoundingMap()
        psm.set_from_m21_part(m21_part)
        if psm.single_events:
            self.sounding_maps.append(psm)
            self.attacks.extend(psm.attack_global_offsets)
            self.attacks = sorted(set(self.attacks))

    def add_score_sounding_maps(self, m21_score):
        # Get and fill measure offsets
        offset_map = m21_score.parts[0].offsetMap()
        self.measure_offsets = {
            om.element.number: make_fraction(om.element.offset)
            for om in offset_map
            if isinstance(om.element, music21.stream.Measure)
        }

        # Get and fill sounding parts
        parts = m21_score.voicesToParts()

        for m21_part in parts:
```

```
            self.add_part_sounding_map(m21_part)

    def get_single_events_by_location(self, global_offset):
        single_events = []
        for sounding_map in self.sounding_maps:
            s_event = sounding_map.get_single_event_by_location(
                global_offset)
            if s_event:
                single_events.append(s_event)
        return single_events

    def make_parsemae(self):
        parsemae = []

        offset_map = {ofs: ms for ms, ofs in self.measure_offsets.items
            ()}
        all_offsets = list(offset_map.keys())

        for attack in self.attacks:
            measure_offset = find_nearest_smaller(attack, all_offsets)
            measure_number = offset_map[measure_offset]
            offset = make_fraction(attack) - make_fraction(
                measure_offset)

            parsema = Parsema()
            parsema.add_single_events(self.
                get_single_events_by_location(attack))
            parsema.global_offset = attack
            parsema.measure_number = measure_number
            parsema.offset = offset
            parsemae.append(parsema)

        if not parsemae:
            return

        # Merge parsemae
        merged_parsemae = []
        first_parsema = parsemae[0]
        for parsema in parsemae[1:]:
            if parsema.partition_info == first_parsema.partition_info:
                first_parsema.duration += parsema.duration
            else:
                merged_parsemae.append(first_parsema)
                first_parsema = parsema

        merged_parsemae.append(first_parsema)

        return merged_parsemae
```

```python


class Texture(object):
    def __init__(self):
        self.parsemae = []
        self._measure_offsets = {}

    def __repr__(self):
        return '<T: {} parsemae>'.format(len(self.parsemae))

    def make_from_music21_score(self, m21_score):
        ssm = ScoreSoundingMap()
        ssm.add_score_sounding_maps(m21_score)
        self.parsemae = ssm.make_parsemae()
        self._measure_offsets = ssm.measure_offsets

    def _auxiliary_get_data(self):
        columns = [
            'Index', # 0
            'Measure number', # 1
            'Offset', # 2
            'Global offset', # 3
            'Duration', # 4
            'Partition', # 5
            'Density-number', # 6
            'Agglomeration', # 7
            'Dispersion', # 8
        ]
        data = []
        for parsema in self.parsemae:
            ind = tuple([parsema.measure_number, parsema.offset])
            data.append([
                ind,
                parsema.measure_number,
                parsema.offset,
                parsema.global_offset,
                parsema.duration,
                parsema.partition_pretty,
                parsema.get_density_number(),
                parsema.get_agglomeration_index(),
                parsema.get_dispersion_index(),
            ])
        dic = {
            'header': columns,
            'data': data
        }
        return dic

```

```
    def _auxiliary_get_data_complete(self):
        # check indexes
        auxiliary_dic = self._auxiliary_get_data()
        data = auxiliary_dic['data']
        data_map = {row[3]: row for row in data}
        global_offsets = [row[3] for row in data]
        common = make_fraction(get_common_denominator_from_list(
            global_offsets))
        size = global_offsets[-1] + data[-1][4]

        new_data = []
        current_global_offset = global_offsets[0]
        last_row = data[0]

        measure_index = 0
        while current_global_offset < size:
            current_measure, measure_index = auxiliary_find_interval(
                current_global_offset, self._measure_offsets,
                measure_index)

            if current_global_offset in data_map:
                row = copy.deepcopy(data_map[current_global_offset])
                last_row = copy.deepcopy(row)
            else:
                row = copy.deepcopy(last_row)
                row[2] = current_global_offset - self._measure_offsets[
                    current_measure]
                row[3] = current_global_offset

            row[0] = '{}+{}'.format(str(current_measure), str(row[2]))
            row[1] = current_measure
            new_data.append(row)

            last_row = row
            current_global_offset = make_fraction(current_global_offset
                + common)

        dic = {
            'header': auxiliary_dic['header'],
            'data': new_data,
        }

        return dic

    def get_data(self, equal_duration_events=True):
        '''Get parsemae data as dictionary with data and index. If
            only_parsema_list attribute is False, the data is filled
            with equal duration events.'''
```

```
        if equal_duration_events:
            return self._auxiliary_get_data_complete()
        else:
            return self._auxiliary_get_data()


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
                prog = 'rpc',
                description = 'Rhythmic Partitioning Calculator',
                epilog = 'Rhythmic Partitioning Calculator')
    parser.add_argument('filename')

    args = parser.parse_args()
    fname = args.filename

    print('Running script on {} filename...'.format(fname))
    try:
        sco = music21.converter.parse(fname)
    except:
        raise CustomException('File must be XML or KRN.')

    texture = Texture()
    texture.make_from_music21_score(sco)
    dic = texture.get_data(equal_duration_events=True)

    # Filename
    split_name = fname.split('.')
    if len(split_name) > 2:
        base = '.'.join(split_name[:-1])
    else:
        base = split_name[0]
    dest = base + '.csv'

    with open(dest, 'w') as fp:
        csv_writer = csv.writer(fp, quoting=csv.QUOTE_NONNUMERIC)
        csv_writer.writerow(dic['header'])
        csv_writer.writerows(dic['data'])
```

## B. RPP's source-code

```
from fractions import Fraction
from matplotlib import pyplot as plt
import argparse
import pandas

```

```python
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)
```

```
    df.index = df['Index'].apply(parse_index).values
    df['Partition'] = df['Partition'].apply(parse_pow)
    df = df.drop('Index', axis=1)

    return df

def invert_dataframe(df):
    inverted = pandas.DataFrame([
        df.Agglomeration * -1,
        df.Dispersion,
    ], index=['Agglomeration', 'Dispersion'], columns=df.index).T
    return inverted

def plot_simple_partitiogram(df, img_format='svg', with_labels=True,
   outfile=None):
    seq = [
        [partition, len(_df), _df.Agglomeration.iloc[0], _df.Dispersion
            .iloc[0]]
        for partition, _df in df.groupby('Partition')
    ]
    columns=['Partition', 'Quantity', 'Agglomeration', 'Dispersion']
    df = pandas.DataFrame(seq, columns=columns)

    plt.clf()
    ax = df.plot(
        grid=True,
        kind='scatter',
        x='Agglomeration',
        y='Dispersion',
    )

    if with_labels:
        factor = 1.025
        fontsize = 12

        for _, s in df.iterrows():
            x = s['Agglomeration']
            y = s['Dispersion']
            v = s['Partition']
            plt.text(x * factor, y * factor , v, fontsize=fontsize)

    if img_format == 'svg':
        plt.savefig(outfile)
    else:
        plt.savefig(outfile, dpi=RESOLUTION)
    plt.close()

```

```
def plot_simple_indexogram(df, img_format='svg', outfile=None, close_bubbles=False):
    def draw_vertical_line(row, x):
        ymin = row[1].Agglomeration * -1
        ymax = row[1].Dispersion
        plt.vlines(x=x, ymin=ymin, ymax=ymax, linestyles='dotted', colors='C3')

    inverted = invert_dataframe(df)

    plt.clf()

    ax = inverted.plot(grid=True)
    ax.set_ylabel('Values\n<- aggl./disp. ->')
    ax.set_xlabel('Positions (measure, offset)')

    # draw vertical lines to close the bubbles
    if close_bubbles:
        rest_segment = False
        last_row = None
        for i, row in enumerate(df.iterrows()):
            _agg = row[1].Agglomeration
            if pandas.isnull(_agg):
                if not rest_segment:
                    if last_row:
                        x = i - 1
                        draw_vertical_line(last_row, x)
                    rest_segment = True
            else:
                if rest_segment:
                    x = i
                    draw_vertical_line(row, x)
                    rest_segment = False
            last_row = row

    plt.xticks(rotation=90)
    plt.tight_layout()

    if img_format == 'svg':
        plt.savefig(outfile)
    else:
        plt.savefig(outfile, dpi=RESOLUTION)
    plt.close()

def plot_stem_indexogram(df, img_format='svg', outfile=None):
    inverted = invert_dataframe(df)

    ind = ['({}, {})'.format(a, b) for a, b in inverted.index.values]
```

```
    size = len(ind)
    step = int(size / 8)

    plt.clf()
    plt.stem(ind, inverted.Dispersion.values, markerfmt=' ')
    plt.stem(ind, inverted.Agglomeration.values, markerfmt=' ', linefmt='C1-')
    plt.xticks(range(0, size, step))
    plt.xlabel('Positions (measure, offset)')
    plt.ylabel('Values\n<- aggl./disp. ->')
    plt.grid()
    plt.legend(inverted.columns)
    plt.xticks(rotation=90)
    plt.tight_layout()

    if img_format == 'svg':
        plt.savefig(outfile)
    else:
        plt.savefig(outfile, dpi=RESOLUTION)
    plt.close()

def plot_stairs_indexogram(df, img_format='svg', outfile=None):
    inverted = invert_dataframe(df)

    ind = ['({}, {})'.format(a, b) for a, b in inverted.index.values]
    size = len(ind)
    step = int(size / 8)

    plt.clf()
    plt.stairs(inverted.Dispersion.values[:-1], ind)
    plt.stairs(inverted.Agglomeration.values[:-1], ind)
    plt.xticks(range(0, size, step))
    plt.xlabel('Positions (measure, offset)')
    plt.ylabel('Values\n<- aggl./disp. ->')
    plt.grid()
    plt.legend(inverted.columns)
    plt.xticks(rotation=90)
    plt.tight_layout()

    if img_format == 'svg':
        plt.savefig(outfile)
    else:
        plt.savefig(outfile, dpi=RESOLUTION)
    plt.close()

def plot_combined_indexogram(df, img_format='svg', outfile=None):
    inverted = invert_dataframe(df)
    series = inverted.Dispersion + inverted.Agglomeration
```

```python

    plt.clf()
    ax = series.plot(grid=True)
    ax.set_ylabel('Values\n(d - a)')
    ax.set_xlabel('Positions (measure, offset)')

    plt.xticks(rotation=90)
    plt.tight_layout()

    if img_format == 'svg':
        plt.savefig(outfile)
    else:
        plt.savefig(outfile, dpi=RESOLUTION)
    plt.close()

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
                    prog = 'rpp',
                    description = "Plot Partitiogram and Indexogram from RPC's output",
                    epilog = 'Rhythmic Partitioning Plotter')

    parser.add_argument('filename')
    parser.add_argument("-f", "--img_format", help = "Image format (svg, jpg or png). Default=svg", default='svg')
    parser.add_argument("-r", "--resolution", help = "PNG image resolution. Default=300", default=300)
    parser.add_argument("-a", "--all", help = "Plot all available charts", action='store_true')
    parser.add_argument("-c", "--close_bubbles", help = "Close indexogram bubbles.", default=False, action='store_true')
    parser.add_argument("-e", "--stem", help = "Indexogram as a stem chart", action='store_true')
    parser.add_argument("-t", "--stairs", help = "Indexogram as a stair chart", action='store_true')
    parser.add_argument("-b", "--combined", help = "Indexogram as a combination of aglomeration and dispersion", action='store_true')
    args = parser.parse_args()

    try:
        RESOLUTION = int(args.resolution)
    except:
        raise CustomException('Resolution must be an integer from 0 to 1200')

    close_bubbles = args.close_bubbles
    if close_bubbles:
```

```
        close_bubbles = True

    img_format = args.img_format.lower()
    if img_format not in ['svg', 'jpg', 'png']:
        raise CustomException('Image format must be svg, jpg or png.')

    fname = args.filename

    print('Running script on {} filename...'.format(fname))

    indexogram_choices = {
        'simple': plot_simple_indexogram,
        'stem': plot_stem_indexogram,
        'stairs': plot_stairs_indexogram,
        'combined': plot_combined_indexogram,
    }

    try:
        df = make_dataframe(fname)
        bname = fname.rstrip('.csv')

        partitiogram_name = bname + '-partitiogram.' + img_format
        plot_simple_partitiogram(df, img_format, outfile=partitiogram_name)

        if args.all:
            for k, fn in indexogram_choices.items():
                outfile = bname + '-indexogram-{}.'.format(k) + img_format
                fn(df, img_format, outfile=outfile)
        elif args.stem:
            k = 'stem'
            fn = indexogram_choices[k]
            outfile = bname + '-indexogram-{}.'.format(k) + img_format
            fn(df, img_format, outfile=outfile)
        elif args.stairs:
            k = 'stairs'
            fn = indexogram_choices[k]
            outfile = bname + '-indexogram-{}.'.format(k) + img_format
            fn(df, img_format, outfile=outfile)
        elif args.combined:
            k = 'combined'
            fn = indexogram_choices[k]
            outfile = bname + '-indexogram-{}.'.format(k) + img_format
            fn(df, img_format, outfile=outfile)
        else:
            k = 'simple'
            fn = indexogram_choices[k]
```

```
            outfile = bname + '-indexogram.' + img_format
            fn(df, img_format, outfile=outfile, close_bubbles=close_bubbles)

    except:
        raise CustomException('Something wrong with given csv file.')
```

## C. RPA's source-code

```
import argparse
import csv
import fractions
import math
import music21

def find_nearest_smaller(value, seq):
    if value < seq[0]:
        return -1

    if value > seq[-1]:
        return seq[-1]

    size = len(seq)

    if size == 1 and value >= seq[0]:
        return seq[0]

    middle_pointer = math.floor(size/2)
    left = seq[:middle_pointer]
    right = seq[middle_pointer:]

    if value < right[0]:
        return find_nearest_smaller(value, left)
    else:
        return find_nearest_smaller(value, right)

def simplify_csv(csv_fname):
    seq = []
    last_row = None
    with open(csv_fname, 'r') as fp:
        i = 0
        for row in csv.reader(fp):
            if i > 0:
                if i == 1:
                    last_row = row
                if row[5] != last_row[5]:
                    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
```

```
            if m.number in events_location.keys():
                for _offset, partition in events_location[m.number]:
                    rest = music21.note.Rest(quarterLength=1/256)
                    rest.offset = _offset
                    rest.addLyric(partition)
                    new_measure.insert(_offset, rest)
                new_measure = new_measure.makeRests(fillGaps=True)
                for el in new_measure:
                    el.style.color = 'white'
                    el.style.hideObjectOnPrint = True
            measures.update({m.number: new_measure})

    for m in measures.values():
        new_part.append(m)

    new_part = new_part.makeRests(fillGaps=True)

    sco.insert(0, new_part)
    sco.write(fmt='xml', fp=outfile)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
                prog = 'rpa',
                description = 'Rhythmic Partitioning Annotator',
                epilog = 'Rhythmic Partitioning Annotator')
    parser.add_argument("-s", "--score", help = "Score filename.")
    parser.add_argument("-c", "--csv", help = "CSV filename.")

    args = parser.parse_args()
    sco_fname = args.score
    csv_fname = args.csv

    print('Running script on {} filename...'.format(sco_fname))

    sco = music21.converter.parse(sco_fname)
    outfile = csv_fname.rstrip('.csv') + '-annotated.xml'

    main(sco, csv_fname, outfile)
```