PARTITURA: A PYTHON PACKAGE FOR SYMBOLIC MUSIC PROCESSING

Carlos Cancino-Chacón*
Johannes Kepler University
carlos_eduardo.
cancino_chacon@jku.at

Silvan David Peter*
Johannes Kepler University
silvan.peter@jku.at

Emmanouil Karystinaios*
Johannes Kepler University
emmanouil.karystinaios@jku.at

Francesco Foscarin*
Johannes Kepler University
francesco.foscarin@jku.at

Maarten Grachten
Independent Researcher
maarten.grachten@gmail.com

Gerhard Widmer
Johannes Kepler University
gerhard.widmer@jku.at

Abstract

Partitura is a lightweight Python package for handling symbolic musical information. It provides easy access to features commonly used in music information retrieval tasks, like note arrays (lists of timed pitched events) and 2D piano roll matrices, as well as other score elements such as time and key signatures, performance directives, and repeat structures. Partitura can load musical scores (in MEI, MusicXML, Humdrum **kern, and MIDI formats), MIDI performances, and score-to-performance alignments. The package includes some tools for music analysis, such as automatic pitch spelling, key signature identification, and voice separation. Partitura is an open-source project and is available at `https://github.com/CPJKU/partitura/`.

Introduction

In the past few years, symbolic music processing has been gaining increasing attention in the Music Information Research (MIR) community, with several music datasets of symbolic formats recently released, e.g. (Foscarin, McLeod, Rigaux, Jacquemard, & Sakai, 2020; Kong, Li, Chen, & Wang, 2020; Micchi, Gotham, & Giraud, 2020). Systems that target symbolic data are usually more efficient and easier to interpret than systems that target lower-level representation of music, such as audio files. This is not surprising, as sequences of notes are more compact and interpretable than sequences of amplitudes over time. Symbolic formats can encode much more than a sequential note representation. Symbolically encoded musical scores arrange those

---

*equal contribution.

notes in temporal and organizational structures such as measures, beats, parts, and voices. They can also explicitly represent dynamics and temporal directives and other high-level musical features such as time signature, pitch spelling, and key signatures.

While this rich set of musical elements adds useful information that can be leveraged by MIR systems, it also drastically increases the complexity of encoding and processing symbolic musical formats. Common formats for storage such as MEI, MusicXML, Humdrum **kern and MIDI are not ideally suited to be directly used as input in MIR tasks[1]. Therefore, the typical data processing pipeline starts with parsing the relevant information from those files and putting it into a convenient data structure (*e.g.,* numerical arrays that can be used directly as input for machine learning or signal processing methods). Both operations require musical knowledge and can be very time-consuming, thus constituting a major barrier, especially for data-driven approaches that require a large dataset to be trained, and for researchers with limited musical background.

These problems have motivated us to develop Partitura. Our goal is to simplify, as much as possible, all steps from the symbolic encoding to a convenient input data structure for a MIR system. Partitura can straightforwardly produce standard data structures while still handling a complete set of symbolic music elements to create a customized one. Partitura can parse symbolic representations of musical scores and performances from multiple file encodings (MEI, MusicXML, Humdrum **kern, and MIDI) into Python objects to easily access their content. Moreover, it can produce commonly used data structures such as piano rolls and note arrays at different time resolutions.

The rest of this paper is structured as follows: Section 1 highlights the differences between Partitura and other python packages for processing symbolic musical formats. The package functionalities are detailed in Section 2, and in Section 3 we provide a short usage example. Finally, in Section 4 we draw some conclusions on this paper and discuss possible future work.

1 Related Work

Among the available Python packages for parsing and processing music in symbolic formats, there are two that stand out in terms of popularity and usability, *Pretty MIDI* and *music21*.

Pretty MIDI (Raffel & Ellis, 2014) is a Python package that focuses on the analysis, modification, and generation of MIDI data in a fast and straightforward way. A strong feature of PrettyMIDI is its ability to easily extract MIDI properties such as the position of beats and downbeats, key and time signatures, and to produce piano roll representation with a specific sample frequency. Partitura follows the PrettyMIDI philosophy of speed and simplicity but extends it to other symbolic formats of musical scores, and to the other notation elements they contain. Moreover, while PrettyMIDI only represents time in seconds, Partitura can work with other time units such as beats and quarter notes.

Another well-known python package for handling both MIDI and richer symbolic encodings of musical scores is *music21* (Cuthbert & Ariza, 2010). Indeed, music21 has been developed and supported for many years now. It offers a robust parser for many file formats, and support for many "advanced" score elements such as nested tuples and beamings. Among the package goals, there are advanced modifications of musical scores, such as transpositions, pitch

---

[1]Though a few works exist that target directly Humdrum **kern files, *e.g.,* (Román, Pertusa, & Calvo-Zaragoza, 2019), but some preprocessing is always required.

respelling, insertion, and deletion of voices and measures. All this is supported by an internal representation based on nested containers called Streams that model the hierarchical temporal and organizational structure of the score in measures, voices, and parts. Partitura does not aim at rebuild such a complete and complex framework, instead, it focuses on a different goal: lightweight extraction of features that are relevant for MIR research, typically sequential representations of score elements such as piano rolls or note arrays. To efficiently target this objective, Partitura uses a much simpler, but nonetheless complete, sequential representation of musical scores, with musical elements arranged in a timeline.

2 Partitura

Partitura can handle three symbolic data types: musical scores, performances, and score-to-performance alignments. The score contains a representation of music, highly structured in staves, measures, beats, and voices, and express durations in *musical units* quantized to fractions of quarter notes and beats. The performance is a sequential representation of musical events expressed on a continuous timeline and not quantized to fixed values. Alignments between the two formats can be done at note- or time-level (*e.g.,* beat and measure). Different file formats can be parsed into dedicated internal representations to offer easy access to the file content. A set of functions creates data structures that are often used in MIR research. Finally, Partitura offers some music analysis tools.

2.1 Internal Data Structures

Different internal data types are built to represent scores, performances, and score-to-performance alignments.

For a score, Partitura uses three main classes: *TimePoints*, *TimedObjects*, and *Parts*. At the highest level, there are one or more *Part* objects, possibly grouped by *PartGroup* objects. *Parts* are typically associated with instruments, and each *Part* may have one or more staves. Each *Part* contains a timeline that encapsulates a sequence of *TimePoint* objects, each denoting a temporal position in the score. *TimePoints* encode score time in non-negative integer units. The relation of this unit to a quarter note is chosen such that any temporal position present in the score can be represented in integer values.

Musical elements (for example, a *Note*) are added to the timeline by registering them with the *TimePoints* corresponding to their start and end positions. Any element registered with two *TimePoints* is a *TimedObject*. Partitura represents a large set of score elements as subclasses of the *TimedObject*, *e.g.,* notes, rests, time signatures, key signatures, slurs, measures, tempo and loudness directives. Figure 1 shows a schematic representation of a *Part* object and its components.

In contrast to scores, the performance is inherently sequential and can be represented in a simpler structure. Partitura uses *PerformedPart* objects that consist of two ordered containers which store notes and MIDI control information. A note object of a *PerformedPart* is a dictionary encoding MIDI note parameters (onset, offset, velocity, pitch, channel, and track) as well as a deterministically generated unique note identifier.

Score-to-performance alignments are represented with a *Part*, a *PerformedPart*, and a sequence of alignment pairs. Each alignment pair encodes a link between a note ID (or time position) in
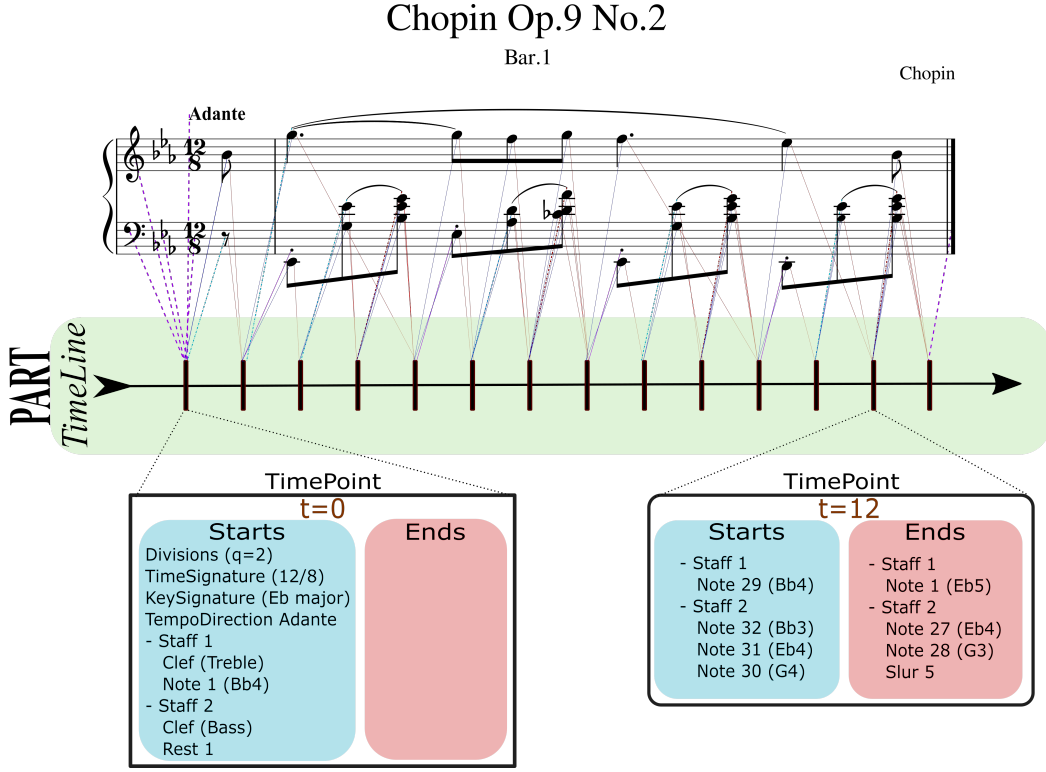
Figure 1: Schematic representation of a Part. The blue lines represent the starting times of the objects in the score and the red lines represent the end times.

the score and a note ID (or time position) in the performance.

2.2 Supported File Formats

Partitura can parse score formats such as MEI, MusicXML, Humdrum **kern and produce *Part* objects. The case of MIDI files is more complex, as they can encode either a performance or a bare-bones scores representation (Back, 1999). Partitura loads MIDI scores and MIDI performances into *Part* and *PerformedPart* objects respectively. As far as output file formats are concerned, Partitura can produce MusicXML and MIDI files from *Parts* and MIDI files from *PerformedParts*.

Partitura supports import and export functionality for match files, a format for encoding symbolic score-to-performance music alignments (Foscarin et al., 2022). Furthermore, Partitura parses simpler alignment file formats such as the .match and .corresp files proposed by Nakamura, Yoshii, and Katayose (2017).

2.3 Generated Data Structures

Although convenient for lossless representation of score time, the internal representation of time points and durations as integers is not particularly meaningful from a musical perspective. For this reason, Partitura can output temporal positions and durations in two other units: quarter notes and beats. For example, the upper-staff notes of the score in Figure 1 would have a temporal position of $[0, 1, 5, 6, 7, \ldots]$ if we are considering quarter notes, $[-1, 0, 4, 5, 6 \ldots]$ if we are considering "slow-tempo beats" (12 beats for the measure) or $[-0.333, 0, 1, 0.333, 0.666, \ldots]$

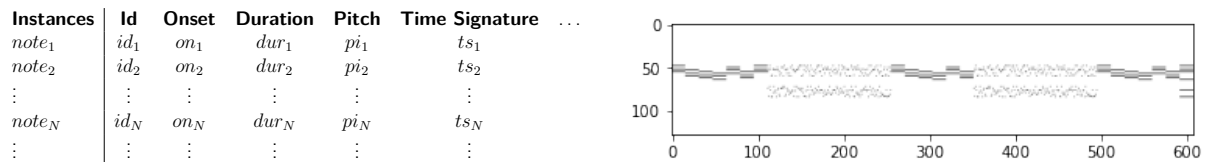| Instances	Id	Onset	Duration	Pitch	Time Signature	...
$note_1$	id_1	on_1	dur_1	pi_1	ts_1	
$note_2$	id_2	on_2	dur_2	pi_2	ts_2	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$note_N$	id_N	on_N	dur_N	pi_N	ts_N	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Figure 2: An abstract example of a note array (left) and a piano roll(right).

if we are considering "fast-tempo beats" (4 beats for the measure). Mappings between various time units are readily available as *Part* methods.

Partitura can automatically generate two data structures that are commonly used in MIR tasks: *note arrays* and *piano rolls matrices* (see Figure 2). A note array is an ordered sequence of note features. Such features can include note descriptors (*e.g.,* midi-pitch, pitch-spelling, onset position, voice, and duration), but also context information like metrical position, time signature, and key signature. Users can choose among these features the ones that are related to their application. A use case is demonstrated in Figure 4 From this representation, we can build a dedicated word encoding of the musical score, as done by Hawthorne et al. (2019). A piano roll is a matrix of shape (number of pitches × number of time frames) where the length of a time frame can be set as fractions of beats or quarter notes. For example, if we consider 4 frames per quarter note, and the piano range, the score of Figure 1 would produce a piano roll of shape (88×28). This representation is widely used in the MIR community, for example, by Huang et al. (2019). Built-in methods to create *note arrays* and *piano rolls matrices* are available for both *Part* and *PerformedPart* objects. For efficient processing in Python, note arrays and score piano rolls are numpy arrays.[2]

2.4 Music Analysis and Repetition Unfolding Tools

Partitura includes some tools for music analysis that are intended to fill in missing information with plausible values, for instance, when loading a score from a MIDI file. The list of available tools includes the Krumhansl–Shepard algorithm (Krumhansl, 1990) for key signature estimation, the *ps13s1* algorithm (Meredith, 2006) for pitch spelling, and *VoSA* (Chew & Wu, 2004) for polyphonic voice estimation. To our knowledge, this is the first publicly available Python implementation of ps13s1 and VoSA.

Musical scores often encode repetition structures with repeat signs, Volta brackets, and navigation directions such as al Coda, dal Segno, da Capo, or al Fine. On the other side, music performances are "unfolded" and multiple possible unfoldings can exist for a piece, as players often decide to skip some repetitions. Partitura supports the generation of such unfoldings from a score's repetition structure and their conversion into a new *Part* object.

3 Getting Started

In this section, we present a quick introduction to the usage of the Partitura package. For more examples of use cases, as well as a more detailed description of the elements of the package,

---

[2]https://numpy.org/

```
1  import partitura as pt
2
3  # Load score
4  score = pt.load_score("chopin_op9_no2.mei")
5  # Load MIDI as a performance
6  performance = pt.load_performance("chopin_op9_no2_perf.mid")
7  # Load Alignment
8  performance, alignment = pt.load_match('chopin_op9_no2.match')
```

Figure 3: Importing Files.

please refer to the online documentation.[3] A hands-on tutorial can be found on Google Colab.[4]

The Partitura package can be installed from Python Package Index[5] with the command `<pip install partitura>` or directly from the source code available on Github.

3.1 Importing Files

As mentioned in Section 2.1, Partitura treats musical scores and performances differently, and this is reflected in how scores and performances are imported. Partitura includes a generic `load_score` method for loading files as scores (*i.e.,* as *Parts*), as well as a `load_performance` method for loading files as performances (*i.e.,* as *PerformedParts*). These generic methods infer the format of the input file automatically. Additionally, there are individual methods for loading supported formats (*e.g.,* `load_musicxml`, `load_mei`, `load_kern` for MusicXML, MEI, Humdrum **kern files, respectively). For MIDI files, Partitura provides both `load_score_midi` and `load_performance_midi` methods. By doing so, we expect users to know what kind of information they would like to extract from MIDI files. As mentioned in Section 2.4, we use the included music analysis tools to infer plausible values for the missing information (especially pitch spelling and voice information) in MIDI files imported as scores. Figure 3 shows an example of loading files.

3.2 Computing Note Arrays and Piano Rolls

Figure 4 shows an example of extracting note arrays and piano rolls from a score (the syntax is the same for performances). This example illustrates the philosophy of Partitura of reducing the most common operations on symbolic music to one-line Python commands.

In Partitura, note arrays are implemented using Numpy structured arrays,[6] arrays in which each column can have a different datatype. By default, note arrays generated from scores include onset and duration information (in beats, quarters and divs), MIDI pitch, voice and note ID. Note arrays generated from performances include onset and duration in seconds, MIDI pitch, velocity track and channel; and Note ID. The `note_array` method in *Parts* and

---

[3]https://partitura.readthedocs.io/
[4]https://tinyurl.com/partituratutorial
[5]https://pypi.org
[6]https://numpy.org/doc/stable/user/basics.rec.html

```
1  import partitura as pt
2  import matplotlib.pyplot as plt
3
4  # Import score
5  part = pt.load_score("chopin_op9_no2.mei")
6  # Get note array
7  note_array = part.note_array(include_time_signature=True)
8  # get piano roll
9  pianoroll = pt.utils.compute_pianoroll(part,
10                                         time_div=8,
11                                         piano_range=True)
12 # plot piano roll
13 plt.spy(pianoroll)
```
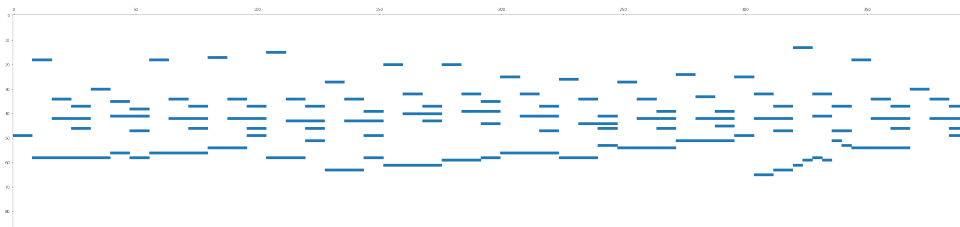
Output:

Figure 4: Extracting note arrays and piano rolls (score of Figure 1).

PerformedParts can receive a list of user defined callable methods which can compute other features at the note level (*e.g.,* scale degree, etc.).

Since computing piano rolls results in very sparse matrices, Partitura computes piano rolls as Scipy sparse matrices.[7] The method `compute_pianoroll` can also specify the desired resolution of the piano roll by specifying the number of sub-divisions for each time unit with the `time_div` argument.

3.3 Music Analysis Tools

As mentioned in Section 2.4, Partitura includes tools for estimating key signature, pitch spelling, voice information, and tonal tension, which can be found in the `partitura.musicanalysis` module. The methods in this module accept *Parts*, *PerformedParts* or note arrays as input and return a Numpy structured array with the estimated information, except for `estimate_key` which returns the estimated key signature as a string (*e.g.,* `'Cm'`, `'F#m'`, etc.). Figure 5 shows an example of how to compute this information from a score.

---

[7]https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

```
1  import partitura as pt
2  # Load score
3  part = pt.load_score("chopin_op9_no2.mei")
4  # Estimate key signature
5  key_name = pt.musicanalysis.estimate_key(part)
6  # Estimate pitch spelling
7  pitch_spelling = pt.musicanalysis.estimate_spelling(part)
8  # Estimate voice information
9  voices = pt.musicanalysis.estimate_voices(part)
10 # Compute tonal tension
11 tonal_tension = pt.musicanalysis.estimate_tonaltension(part)
```

Figure 5: Music analysis tools in Partitura.

4 Conclusions and Future Work

In this paper, we presented Partitura, a Python package for handling symbolic music information that requires minimal music expertise. This package can parse common symbolic music formats, like MusicXML, MEI, and MIDI, and conveniently represent them as Python objects that are easy to manipulate in automatic data pipelines. Moreover, it can straightforwardly produce the most commonly used data structures for MIR tasks. To the best of our knowledge, Partitura is the only Python library that can handle alignments between scores and corresponding performances.

Future work will be in the direction of making Partitura file parsers more robust to bad encoding practices that are unfortunately very frequent in symbolic musical scores. Moreover, support for more score elements and different data structures will be added to keep Partitura on track with new needs and demands from the research community. We are working on making Partitura faster and more efficient by optimizing existing methods and including support for parallel data processing. Another functionality that will be added is the automatic score unfolding to match the repetition structure of a given performance. Finally, we will develop more analysis tools to infer high-level score elements, with the final goal of being able to "scorify" a performance or an incomplete score representation.

Acknowledgements

References

Back, D. (1999). *Standard MIDI-file format specifications.* Retrieved from http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html (Accessed

September 23, 2020)

Chew, E., & Wu, X. (2004). Separating Voices in Polyphonic Music: A Contig Mapping Approach. In *Proceedings of the International Symposium on Computer Music Multidisciplinary Research (CMMR).* Esbjerg, Denmark.

Cuthbert, M. S., & Ariza, C. (2010). music21: A toolkit for computer-aided musicology and symbolic music data. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR).* Utrecht, Netherlands.

Foscarin, F., Karystinaios, E., Peter, S. D., Cancino-Chacón, C. E., Grachten, M., & Widmer, G. (2022). The match file format: Encoding Alignments between Scores and Performances. In *Proceedings of the music encoding conference.* Halifax, Canada.

Foscarin, F., McLeod, A., Rigaux, P., Jacquemard, F., & Sakai, M. (2020). ASAP: a dataset of aligned scores and performances for piano transcription. In *International Society for Music Information Retrieval Conference (ISMIR)* (pp. 534–541). Montréal, Canada.

Hawthorne, C., Stasyuk, A., Roberts, A., Simon, I., Huang, C.-Z. A., Dieleman, S., . . . Eck, D. (2019). Enabling Factorized Piano Music Modeling and Generation with the MAESTRO Dataset. In *International Conference on Learning Representations.* New Orleans, USA. Retrieved from `https://openreview.net/forum?id=r1lYRjC9F7`

Huang, A., Hawthorne, C., Roberts, A., Dinculescu, M., Wexler, J., Hong, L., & Howcroft, J. (2019). Bach Doodle: Approachable music composition with machine learning at scale. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR).* Delft, Netherlands. Retrieved from `https://arxiv.org/abs/1907.06637`

Kong, Q., Li, B., Chen, J., & Wang, Y. (2020). Giantmidi-piano: A large-scale midi dataset for classical piano music. *arXiv preprint arXiv:2010.07061.*

Krumhansl, C. L. (1990). *Cognitive foundations of musical pitch.* New York: Oxford University Press.

Meredith, D. (2006). The *ps13* Pitch Spelling Algorithm. *Journal of New Music Research*, *35*(2), 121–159.

Micchi, G., Gotham, M., & Giraud, M. (2020). Not all roads lead to rome: Pitch representation and model architecture for automatic harmonic analysis. *Transactions of the International Society for Music Information Retrieval (TISMIR)*, *3*(1), 42–54.

Nakamura, E., Yoshii, K., & Katayose, H. (2017). Performance Error Detection and Post-Processing for Fast and Accurate Symbolic Music Alignment. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)* (pp. 347–353). Suzhou, China.

Raffel, C., & Ellis, D. P. (2014). Intuitive analysis, creation and manipulation of midi data with pretty midi. In *Late-breaking demo session of the international society for music information retrieval conference* (pp. 84–93). Taipei, Taiwan.

Román, M. A., Pertusa, A., & Calvo-Zaragoza, J. (2019). A Holistic Approach to Polyphonic Music Transcription with Neural Netwoks. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)* (pp. 731–737). Delft, Netherlands.